

EXPERT INSIGHT

Time Series Analysis with Python Cookbook

Practical recipes for the complete time series workflow,
from modern data engineering to advanced forecasting
and anomaly detection

Second Edition

Bonus content on GitHub



<packt>

Tarek A. Atwan

Time Series Analysis with Python Cookbook

Second Edition

Practical recipes for the complete time series workflow, from modern data engineering to advanced forecasting and anomaly detection

Tarek A. Atwan



Time Series Analysis with Python Cookbook

Second Edition

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty

Relationship Lead: Apeksha Shetty

Project Manager: Shashank Desai

Content Engineer: Gowri Rekha

Technical Editor: Seemanjay Ameriya

Copy Editor: Safis Editing

Indexer: Hemangini Bari

Proofreader: Gowri Rekha

Production Designer: Ajay Patule

Growth Lead: Abhishek Kaushik

First published: June 2023

Second edition: January 2026

Production reference: 1191225

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80512-428-3

www.packtpub.com

To my mother, who never gets tired of providing unconditional love and support; everything I am, and the person I will become, I owe to you.

– Tarek Atwan

Contributors

About the author

Tarek A. Atwan is a seasoned analytics and technology leader with over 20 years of international experience in data science, machine learning, and data engineering. As an independent consultant and corporate trainer, he advises large organizations on their strategic initiatives in AI, cloud data architecture, and **Machine Learning Operations (MLOps)**. Tarek is also a passionate educator and author, recognized as a Pluralsight Elite Instructor Award recipient (2023, 2024, and 2025). He has created and taught numerous courses on advanced topics—including deep learning, Generative AI, MLOps, LLMOps, anomaly detection, and time-series analysis—at prestigious institutions and platforms. He is the author of *Time Series Analysis with Python Cookbook* (Packt Publishing), now in its second edition and also translated into Japanese. He has achieved multiple certifications across AWS, Microsoft Azure, Google Cloud, and Databricks.

I would like to thank my amazing wife, Fella, for her unwavering support, patience, and understanding throughout this long journey.

I owe a big thank you to Packt's fantastic editorial team and reviewers for this project—Shashank Desai, Gowri Rekha, Juan Villanueva, and the rest of the Packt team—for their guidance and hard work in bringing this second edition to life.

About the reviewer

Juan Hurtado is a senior data scientist and engineer. He started his data science career seven years ago in the oil and gas industry, working for several start-ups mainly in time-series analysis. This work included time-series aggregations, decomposition, forecasting, and event detection using statistical and machine learning techniques. He has three research papers presented at different oil and gas conferences. He holds a Bachelor of Science in mechanical engineering from Universidad de los Andes (Colombia), a Master of Engineering in petroleum engineering, and a Master of Science in engineering and technology management from Colorado School of Mines, and he is currently finishing his master's in computer science (machine learning specialization) from Georgia Institute of Technology.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



Table of Contents

Preface	xi
Free Benefits with Your Book	xviii
Chapter 1: Reading Time Series Data from Files	1
Technical requirements	2
Reading data from CSV and other delimited files	3
Reading data from an Excel file	9
Reading data from URLs	14
Reading data from Parquet files	25
Chapter 2: Reading Time Series Data from Databases	33
Reading data from a relational database	36
Reading data from Snowflake	50
Reading data from MongoDB	61
Chapter 3: Persisting Time Series Data to Files	75
Technical requirements	75
Serializing time series data with pickle	76
Writing to CSV and other delimited files	83
Writing data to an Excel file	88
Storing data in cloud storage (AWS, GCP, and Azure)	95
Chapter 4: Persisting Time Series Data to Databases	103
Technical requirements	104
Writing time series data to a relational database	105
Writing time series data to MongoDB	117
Writing time series data to Snowflake	127

Chapter 5: Working with Date and Time in Python	137
Technical requirements	138
Working with DatetimeIndex	138
Providing a format argument to DateTime	148
Working with Unix epoch timestamps	152
Working with time deltas	158
Converting DateTime with time zone information	165
Working with date offsets	170
Working with custom business days	177
Chapter 6: Handling Missing Data	185
Technical requirements	186
Understanding missing data	187
Performing data quality checks	189
Handling missing data with univariate imputation using pandas	199
Handling missing data with univariate imputation using scikit-learn	208
Handling missing data with multivariate imputation	215
Handling missing data with interpolation	223
Chapter 7: Outlier Detection Using Statistical Methods	233
Technical requirements	234
The nature of outliers in time-series data	236
Resampling time-series data	237
Detecting outliers using visualizations	244
Detecting outliers using the Tukey method	253
Detecting outliers using a z-score	259
Detecting outliers using a modified z-score	264
Detecting outliers using the Hampel filter	270
Chapter 8: Exploratory Data Analysis and Diagnosis	283
Technical requirements	284
Decomposing time series data	285
Detecting time series stationarity	297
Applying power transformations	306
Testing for autocorrelation	322

Chapter 9: Building Univariate Time Series Models Using Statistical Methods	329
Technical requirements	330
Plotting ACF and PACF	335
Forecasting univariate time series data with exponential smoothing	347
Forecasting univariate time series data with non-seasonal ARIMA	365
Forecasting univariate time series data with seasonal ARIMA	380
Forecasting univariate time series with auto_arima	391
Chapter 10: Additional Statistical Modeling Techniques for Time Series	401
Technical requirements	402
Forecasting univariate time series data using Prophet	402
Forecasting univariate time series data with Theta	427
Forecasting multivariate time series data using VAR	433
Evaluating VAR models	449
Forecasting volatility in financial time series data with GARCH	460
Automated univariate forecasting using the StatsForecast library	472
Chapter 11: Forecasting Using Supervised Machine Learning	485
Technical requirements	486
Understanding supervised ML	488
Preparing time-series data for supervised learning	489
One-step forecasting with scikit-learn	496
Multi-step forecasting with scikit-learn	511
Forecasting with sktime	522
Hyperparameter tuning with sktime	534
Forecasting with exogenous variables and ensemble learning in sktime	544
Forecasting with XGBoost	555
Chapter 12: Deep Learning for Time Series Forecasting	565
Technical requirements	566
Understanding artificial neural networks	567
Forecasting with LSTM (Keras, Darts, and NeuralForecast)	572
Handling multivariate time series and exogenous variables	606
Hyperparameter tuning for deep learning forecasters	621
Forecasting with TCNs	635
Forecasting with Transformers (N-HiTS)	647

Chapter 13: Outlier Detection Using Unsupervised Machine Learning	661
Technical requirements	663
Understanding considerations for time series outlier detection	665
Detecting outliers using distance-based algorithms	668
Detecting outliers using clustering-based algorithms	695
Detecting outliers using probabilistic and statistical algorithms	703
Detecting outliers using kernel-based algorithms (PyOD)	708
Detecting outliers using ensemble method algorithms	714
Detecting outliers using deep learning-based algorithms (AutoEncoder)	721
Chapter 14: Advanced Techniques for Complex Time Series	731
Technical requirements	732
Understanding state-space models	733
Decomposing time series with multiple seasonal patterns using MSTL	734
Forecasting with multiple seasonal patterns using the Unobserved Components Model (UCM)	743
Forecasting time series with multiple seasonal patterns using AutoTBATS	752
Forecasting time series with multiple seasonal patterns using NeuralProphet	757
Unlock Your Exclusive Benefits	767
Other Books You May Enjoy	771
Index	775

Preface

A familiar association people make is between forecasting and financial data (such as predicting stock prices or market movements). In reality, forecasting is used in more industries than finance, leveraging historical data to make future predictions. More specifically, this book is about time-series analysis, a process to gain better insight from historical data, capture trends and cyclical patterns, and build production-ready forecasting models.

When working with data that contains observations that change over time and is recorded at specific intervals, you are dealing with time-series data. You will find time-series data in many domains, and the discipline of time-series analysis covers various use cases. For example, time-series analysis is used in science (forecasting weather, earthquakes, air quality, or species growth), finance (forecasting stock return, budget, sales, or volatility), government (forecasting inflation, unemployment rates, GDP, or population birth rate), medicine (tracking infectious disease transmission, monitoring electrocardiogram or blood glucose, or forecasting healthcare costs), engineering (predictive maintenance, production decline analysis, or traffic volume forecasting), **business** (inventory management, product demand planning, or resource planning), and much more. Time-series data is pretty much all around us, and you will most definitely encounter it in your work.

In this book, you will find practical end-to-end recipes that you can apply and use—less theory and more actionable results. This *second edition* will take you through the complete journey of time-series analysis. We cover the entire process, from environment setup to acquiring and ingesting time-series data from diverse sources (files, SQL, NoSQL, cloud storage, and data warehouses). You will learn how to explore the data through **exploratory data analysis (EDA)** techniques specific to time series, transform and manipulate the data, and train and evaluate a wide variety of models for forecasting.

This book and its corresponding GitHub repository covers concepts, techniques, and algorithms ranging from commonly used to more advanced and modern approaches. For example, you will learn how to train and validate models across a full spectrum of techniques covering statistical methods (e.g., ARIMA, SARIMA, Theta, Prophet, VAR, and GARCH), machine learning algorithms (e.g., scikit-learn, sktime, and ensemble methods such as XGBoost), deep learning architectures (e.g., RNN, LSTM, TCN, and N-HiTS), probabilistic modeling (e.g., quantile regression, Monte Carlo dropout, and conformal prediction) for forecasting, spectral (or frequency-domain) analysis (e.g., **Fast Fourier transform (FFT)**, periodograms, and Welch's method), and comprehensive outlier (anomaly) detection (using statistical, machine learning, and deep learning techniques).

Most importantly, the variety of datasets used in this book will give you a better insight into how these different models work and how you can pick the most appropriate approach to solve your specific problem. You will be introduced to a vast number of libraries, with enough depth and comparison for you to understand why and when to use them and how they work. The book covers statsmodels, scikit-learn, sktime, Darts, StatsForecast, NeuralForecast, TensorFlow, PyTorch, aeon, PyOD, Prophet, NeuralProphet, XGBoost, LightGBM, Optuna, arch, pmdarima, SciPy, TSFEL, Modin, Dask, Polars, pandas, hvPlot, Seaborn, and much more.

Who this book is for

This book is for data analysts, business analysts, data scientists, data engineers, or Python developers who want practical Python recipes for time-series analysis and forecasting techniques. Fundamental knowledge of Python programming is required. Although having a basic math and statistics background will be beneficial, it is not necessary. Prior experience working with time-series data to solve business problems will also help you to better utilize and apply the different recipes in this book.

What this book covers

This book's GitHub repository includes three bonus chapters not in the print edition: *Chapter 0* covers Python environment setup with Pip and Conda; Bonus *Chapter 15* explores advanced deep learning with probabilistic forecasting; and Bonus *Chapter 16* introduces frequency domain analysis for cyclical patterns. There are also additional recipes: *Ch1-1: Working with large data files*, *Ch2-1: Reading data from a time series database*, *Ch3-1: Writing large datasets*, *Ch4-1: Writing time series data to InfluxDB*, *Ch8-1: Plotting time series data using pandas*, *Ch8-2: Plotting time series data with interactive visualizations using hvPlot* which you can find on GitHub in the respective chapter folders. These resources provide additional foundational and advanced techniques for time series analysis.

Chapter 1, Reading Time Series Data from Files, is an introduction to time-series data. This chapter shows you how to read data from various and commonly used file types, whether stored locally or on the cloud (e.g., AWS S3). You will also learn how to handle large datasets using Dask, Polars, and Modin. The recipes will highlight advanced options for ingesting, preparing, and transforming data into a time-series DataFrame for later analysis.

Chapter 2, Reading Time Series Data from Databases, picks up from *Chapter 1* and focuses on reading data from various database systems. This includes relational (PostgreSQL and MySQL) and non-relational (MongoDB), as well as time-series (InfluxDB, TimescaleDB, and TDEngine) databases, whether on-premises or on a cloud service such as Amazon Redshift or Snowflake. For Snowflake, you will explore multiple APIs, including Snowpark for modern data engineering workflows. The recipes will highlight different methods and techniques to offer flexibility on how data can be ingested, prepared, and transformed into a time-series DataFrame for later analysis.

Chapter 3, Persisting Time Series Data to Files, covers different options and use cases to store time-series data for later retrieval. The techniques will cover various methods and file types, whether on-premises or in the cloud (AWS, GCP, and Azure).

Chapter 4, Persisting Time Series Data to Databases, builds on *Chapter 3*, focusing on writing data for scale. This covers different techniques for writing data to relational and non-relational database systems such as those discussed in *Chapter 2*, including on-premises and cloud services, such as Snowflake.

Chapter 5, Working with Date and Time in Python, takes a practical and intuitive approach to an intimidating topic. You will learn how to deal with the complexity of dates and time in your time-series data. The chapter illustrates practical use cases for handling time zones, custom holidays, and business days, working with Unix epoch and UTC.

Chapter 6, Handling Missing Data, explores different methods for identifying and handling missing data. You will learn different imputation and interpolation techniques. The chapter starts with simple statistical methods and then explores more advanced multivariate imputation, such as MICE and IterativeImputer.

Chapter 7, Outlier Detection Using Statistical Methods, covers statistical methods for outlier and anomaly detection. These techniques are easy to interpret and implement. The chapter uses data from the Numenta Anomaly Benchmark (NAB) to evaluate different anomaly detection algorithms. Beyond point outliers, the chapter covers contextual outliers using techniques such as the Hampel filter, STRAY, and Matrix Profile.

Chapter 8, Exploratory Data Analysis and Diagnosis, dives into visualization techniques for effective EDA, including interactive visualizations with hvPlot. You will learn how to investigate and diagnose your time-series data to test for assumptions such as stationarity and autocorrelation, and apply transformations such as power decomposition, differencing, and power transforms such as the Box-Cox transformation.

Chapter 9, Building Univariate Time Series Models Using Statistical Methods, kicks off the journey into modeling. The chapter intuitively explains the **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF)** plots, and then moves on to training, diagnosing, and comparing different statistical models, including exponential smoothing, **Autoregressive Integrated Moving Average (ARIMA)**, and **Seasonal ARIMA (SARIMA)**. You'll learn how to implement these using the statsmodels, Darts, and StatsForecast libraries and learn how to tune them with grid search.

Chapter 10, Additional Statistical Modeling Techniques for Time Series, dives into more advanced and practical models. This includes the Theta method, **Vector Autoregressive (VAR)** for multivariate time series, **Generalized Autoregressive Conditional Heteroskedasticity (GARCH)** for forecasting volatility, and an introduction to the Prophet algorithm for forecasting and changepoint. Finally, it demonstrates automated univariate forecasting using the StatsForecast library.

Chapter 11, Forecasting Using Supervised Machine Learning, will take you from classical techniques to more machine learning algorithms. The chapter shows how to transform time-series data for supervised learning. You will explore a variety of machine learning algorithms and implement multi-step forecasting, using scikit-learn, Darts, XGBoost, and the sktime framework. Other concepts include forecasting with exogenous variables, using different ensemble learning methods, and hyperparameter tuning.

Chapter 12, Deep Learning for Time Series Forecasting, explores modern deep learning architectures. You will learn how to implement and tune a variety of models, including **Long Short-Term Memory (LSTM)** networks, **Temporal Convolutional Network (TCN)**, and **Neural Hierarchical Interpolation for Time Series (N-HITS)**. The recipes demonstrate how to build these forecasters using multiple powerful libraries, including TensorFlow/Keras, Darts, and NeuralForecast. You will learn how to handle multivariate time series with exogenous variables and perform hyperparameter tuning (using Optuna) for deep learning forecasters.

Chapter 13, Outlier Detection Using Unsupervised Machine Learning, continues from *Chapter 7*, but focuses on more advanced unsupervised machine learning methods. The recipes cover distance-based, clustering-based, probabilistic, kernel-based, ensemble, and deep learning approaches, including autoencoders and Variational Autoencoders (VAEs). You will use the same NAB datasets to compare techniques and learn how to expand from point to contextual outlier detection using the aeon library and your own custom functions.

Chapter 14, Advanced Techniques for Complex Time Series, will introduce more complex time-series data that contains multiple overlapping seasonal patterns. The chapter also looks at how such data can be decomposed using MSTL, and explores different modeling techniques, such as **Unobserved Components Model (UCM)**, TBATS, and NeuralProphet.

Please refer to the GitHub repository for additional information.

To get the most out of this book

You should be comfortable coding in Python, with some familiarity with Matplotlib, NumPy, and pandas. Working knowledge of the Python programming language will assist with understanding the key concepts covered in this book.

The book covers a vast and diverse set of libraries, many of which have complex and potentially conflicting dependencies. To ensure a smooth, reproducible experience and avoid dependency issues, we strongly recommend creating a separate virtual environment for each chapter.

Chapter 0 will show you how to create and manage these environments. To make this process seamless, the book's official GitHub repository supports UV, Pip, or Conda, with setup instructions included in the main README. Each chapter includes the necessary configuration files (`pyproject.toml`, `requirements.txt`, `.python-version`, and `uv.lock`), allowing you to install the exact, tested libraries needed for that set of recipes

It is recommended, but not required, to install either Anaconda, Miniconda, or Miniforge. Throughout the chapters, you will see instructions using either `pip` or `conda`. Alternatively, you can use cloud-based environments such as **Google Colab** or **GitHub Codespaces**, for which you will only need a browser.

Software/hardware covered in the book	Operating system requirements
Python 3.10+	Windows, macOS, or Linux
JupyterLab, Visual Studio Code (VS Code), or a browser for cloud environments such as Colab or Codespaces	Windows, macOS, or Linux

In *Chapter 2* and *Chapter 4*, you will be working with different databases, including PostgreSQL, MySQL, InfluxDB, MongoDB, TimescaleDB, and TDengine. If you do not have access to such databases, you can install them locally on your machine or use Docker. Instructions are provided in the repository to pull the official images from Docker Hub (e.g., `docker pull influxdb`). You can download Docker from <https://docs.docker.com/get-docker/>.

For the recipes using AWS Redshift and Snowflake, you will need to have an account. You can subscribe to the AWS free tier here: <https://aws.amazon.com/free>. You can subscribe to a 30-day Snowflake trial here: <https://signup.snowflake.com>.

Similarly, in *Chapter 1* and *Chapter 3*, recipes for AWS S3, Google Cloud Storage, and Azure Blob Storage will require a valid cloud subscription, which should be covered under their respective free tiers.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

To get the most value out of this book, it is important that you continue to experiment with the recipes using different time-series data. Throughout the book, you will see a recurring theme in which multiple datasets are used. This is deliberate, so you can observe how results vary. You are encouraged to continue with that theme on your own.

If you are looking for additional datasets, in addition to those provided in the GitHub repository, you can check out some of the following links:

- <https://ourworldindata.org>
- <https://www.kaggle.com/datasets?search=time+series>
- <https://github.com/numenta/NAB> (specific to anomaly and outlier detection)
- <https://fred.stlouisfed.org>
- <https://datasetsearch.research.google.com>

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781805124283>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter/X handles. For example: “`pip freeze` allows you to export all pip-installed libraries.”

A block of code is set as follows:

```
data = client.get_object(Bucket=bucket, Key='AirQuality.csv')
df = pd.read_csv(data['Body'],
                 decimal=',',
                 index_col='Date',
                 parse_dates=True)

print(df.iloc[:5, 0:5])
```

Any command-line input or output is written as follows:

```
pip install optuna
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “We will explore a variety of techniques, including Monte Carlo dropout, quantile regression, conformal prediction, and conformalized quantile regression.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at customer@packt.com and mention the book’s title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Share your thoughts

Once you've read *Time Series Analysis with Python Cookbook, Second Edition*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free Benefits with Your Book

This book comes with free benefits to support your learning. Activate them now for instant access (see the “*How to Unlock*” section for instructions).

Here’s a quick overview of what you can instantly unlock with your purchase:

PDF and ePub Copies



Free PDF and ePub versions



Access a DRM-free PDF copy of this book to read anywhere, on any device.



Use a DRM-free ePub version with your favorite e-reader.

Next-Gen Web-Based Reader



Next-Gen Reader



Multi-device progress sync: Pick up where you left off, on any device.



Highlighting and notetaking: Capture ideas and turn reading into lasting knowledge.



Bookmarking: Save and revisit key sections whenever you need them.



Dark mode: Reduce eye strain by switching to dark or sepia themes.

How to Unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

1

Reading Time Series Data from Files

Time series analysis often begins with data preparation, a crucial and time-consuming stage. To effectively analyze time series, you first need to access the data, often stored in various file formats or database systems.

Time series data is complex and can appear in various shapes and formats. For instance, it may come as regular timestamped records, such as hourly temperature readings, or irregular events, such as transaction logs with varying time intervals. Data might also include multiple time series combined into a single dataset, such as sales and inventory levels for several stores.

In this chapter, we will use **pandas**, a popular Python library with a rich set of I/O tools, data wrangling, and date/time handling capabilities that streamline the process of working with time series data. You will explore several of the reader functions available in pandas to ingest data from different file types, including **comma-separated values (CSV)**, **Excel**, and **Parquet**. Additionally, you will explore ingesting data from files stored locally or remotely on the cloud, such as an **Amazon S3 bucket**.

The **pandas** library provides two fundamental data structures for working with time series data: **Series** and **DataFrame**. A **DataFrame** is a distinct data structure for working with tabular data (think rows and columns in a spreadsheet). The main difference between the two data structures is that a **Series** is one-dimensional (single column), while a **DataFrame** is two-dimensional (multiple columns). The relationship between the two is that you get a **Series** when you slice out a column from a **DataFrame**. You can think of a **DataFrame** as a side-by-side concatenation of two or more **Series** objects.

A particular feature of the **Series** and **DataFrames** data structures is that they both have a labeled axis called an **index**. A specific type of index that you will often see with time series data is **DatetimeIndex**, which you will explore further in this chapter. Generally, the index makes slicing and dicing operations very intuitive. For example, to make a **DataFrame** ready for time series analysis, you will learn how to create **DataFrames** with an index of the **DatetimeIndex** type.

The recipes in this chapter will guide you through key techniques to load time series data into pandas DataFrames:

- Reading data from CSV and other delimited files
- Reading data from an Excel file
- Reading data from URLs
- Reading data from Parquet files



There's an exciting GitHub-only bonus—*Chapter 0: Getting Started with Time Series Analysis*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Don't miss the GitHub-exclusive recipe—*Chapter 1-1: Working with large data files*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>



Why DateTimeIndex?

A pandas DataFrame with an index of the `DatetimeIndex` type unlocks a large set of features and useful functions needed when working with time series data. You can think of it as adding a layer of intelligence or awareness to pandas to treat the DataFrame as a time series DataFrame.

Free Benefits with Your Book

Your purchase includes a free PDF copy of this book along with other exclusive benefits. Check the *Free Benefits with Your Book* section in the *Preface* to unlock them instantly and maximize your learning experience.

Technical requirements

In this chapter and forward, you will extensively use pandas 2.3.3 (released September 29, 2025).

Throughout our journey, you will be installing additional Python libraries to use in conjunction with pandas. You can download the Jupyter notebooks from the GitHub repository (https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch1/Chapter_1.ipynb) to follow along.

You can download the datasets used in this chapter from the GitHub repository using this link: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch1>.

Reading data from CSV and other delimited files

In this recipe, you will use the `pandas.read_csv()` function, which offers a large set of parameters that you will explore to ensure that the data is properly read into a time series `DataFrame`. In addition, you will learn how to specify an index column, parse the index to be of the `DatetimeIndex` type, and parse string columns that contain dates into `datetime` objects. A properly parsed `DatetimeIndex` allows for efficient slicing, resampling, and analysis of time-based data.

CSV files store data in plain text, and when using Python to read data from a CSV file (for example, using the `csv` Python module), it will be treated as a string (text) by default. In `pandas`, when using the `read_csv()` method, it will try to infer the appropriate data types (`dtype`), and, in most cases, it does a great job at that. However, there are situations where you will need to explicitly indicate which columns to cast to a specific data type. For example, you will specify which column(s) to parse as dates using the `parse_dates` parameter in this recipe.

Getting ready

You will read a CSV file containing hypothetical box office numbers for a movie. The file is provided in the GitHub repository for this book. The data file is in `datasets/Ch1/movieboxoffice.csv`.

How to do it...

You will ingest our CSV file using `pandas` and leverage some of the available parameters in `read_csv`:

1. Import `pandas` and `Path` from `pathlib` to handle file paths :

```
import pandas as pd
from pathlib import Path
```

2. Create a `Path` object for the file location:

```
filepath = Path('../..../datasets/Ch1/movieboxoffice.csv')
```

Using `Path` objects is a best practice for cross-platform compatibility, ensuring that your code runs smoothly on Windows, macOS, or Linux.

3. Read the CSV file into a `DataFrame` using the `read_csv` function and pass the `filepath` with additional parameters.

The first column in the CSV file contains movie release dates, and it needs to be set as an index of the `DatetimeIndex` type (using `index_col=0` and `parse_dates=['Date']`). Specify which columns you want to include by providing a list of column names to `usecols`. The default behavior is that the first row includes the header (`header=0`):

```
ts = pd.read_csv(filepath,
                  header=0,
                  parse_dates=['Date'],
                  index_col=0,
                  usecols=['Date'],
```

```

        'DOW',
        'Daily',
        'Forecast',
        'Percent Diff'])

ts.head(5)

```

This will output the following first five rows:

	DOW	Daily	Forecast	Percent Diff
Date				
2021-04-26	Friday	\$125,789.89	\$235,036.46	-46.48%
2021-04-27	Saturday	\$99,374.01	\$197,622.55	-49.72%
2021-04-28	Sunday	\$82,203.16	\$116,991.26	-29.74%
2021-04-29	Monday	\$33,530.26	\$66,652.65	-49.69%
2021-04-30	Tuesday	\$30,105.24	\$34,828.19	-13.56%

- Print a summary of the DataFrame to check the index and column data types. Alternatively, you can use raw string `r'^\d. - ]'` to avoid double backslashes.:

```

ts.info()
>> <class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 128 entries, 2021-04-26 to 2021-08-31
Data columns (total 4 columns):
 #   Column          Non-Null Count  Dtype
---  -
 0   DOW              128 non-null    object
 1   Daily            128 non-null    object
 2   Forecast         128 non-null    object
 3   Percent Diff     128 non-null    object
dtypes: object(4)
memory usage: 5.0+ KB

```

Notice that the Date column is now an index (not a column) of the `DatetimeIndex` type. Additionally, the `Daily`, `Forecast`, and `Percent Diff` columns have the wrong dtype inference. You would expect them to be of the `float` type. The issue is due to the source CSV file containing non-numeric characters, such as dollar signs (\$), percent signs (%), and thousand separators (,). The presence of these non-numeric characters can lead to incorrect dtype inference. A column with the dtype `object` indicates either a string column or a column with mixed dtypes (not homogeneous).

- Remove non-numeric characters such as the dollar sign (\$), percent sign (%), and thousand separators (,) using `str.replace()` with a **regular expression**. The regular expression will remove all non-numeric characters but exclude the decimal points (.) and negative sign (-). Removing these characters does not convert the dtype, so you will need to **cast** the columns to the `float` dtype using the `astype(float)` method:

```
clean = lambda x: x.str.replace('[^\d.-]', '', regex=True)
c_df = ts[['Daily', 'Forecast', 'Percent Diff']].apply(clean)
ts[['Daily', 'Forecast', 'Percent Diff']] = c_df.astype(float)
```

The `[^\d.-]` regex (written as `'[^\d.-]'` in Python strings) breaks down as follows:

- `^`: Matches characters **not** in the set that follows
- `\d`: Matches any digit (0–9)
- `.`: Matches the decimal point
- `-`: Matches negative signs for negative numbers



Note that we need to use double backslashes in the Python string (`'\\d'`) to escape the backslash character and properly represent a single backslash in the regex pattern (`'\d'`). Without proper escaping, Python would interpret `\d` as an invalid string escape sequence before it's processed as a regex.

Print a summary of the updated DataFrame:

```
ts.info()
>> <class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 128 entries, 2021-04-26 to 2021-08-31
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   DOW          128 non-null    object
1   Daily        128 non-null    float64
2   Forecast     128 non-null    float64
3   Percent Diff 128 non-null    float64
dtypes: float64(3), object(1)
memory usage: 5.0+ KB
```

Now, the `Daily`, `Forecast`, and `Percent Diff` columns are of the `float64` type, making them suitable for numerical operations and analysis. `DatetimeIndex` allows for efficient slicing and resampling of time-based data, while the cleaned numeric columns are ready for further calculations

How it works...

Using pandas for data transformation is fast since it loads the data into memory. For example, the `read_csv()` method reads and loads the entire data into a `DataFrame` in memory. When requesting a `DataFrame` summary with the `info()` method, the output will display memory usage for the entire `DataFrame` in addition to column data types and index information. To get the exact memory usage for each column, including the index, you can use the `memory_usage()` method.

The `memory_usage()` method returns the memory consumption in **bytes** for each column and the index. The total memory usage (in bytes) matches the memory usage reported in the DataFrame summary:

```
ts.memory_usage().sum()
>> 5120
```

So far, you have used a few of the available parameters when reading a CSV file using `read_csv()`. The more familiar you become with the different options available in any pandas reader functions, the more upfront **preprocessing** you can do during data ingestion (reading). Keep in mind, though, that while performing transformations during data ingestion reduces subsequent processing, it may slightly increase load time for very large datasets.

You used the built-in `parse_dates` argument, which accepts a list of columns specified by name or position. The combination of `index_col=0` and `parse_dates=[0]` produced a DataFrame with an index of the `DatetimeIndex` type. When working with the `parse_dates` parameter, it's good practice to verify that your dates were correctly parsed by checking the DataFrame's index type using `type(ts.index)`. If it returns `pandas.core.indexes.datetimes.DatetimeIndex`, your parsing was successful. If not, you may need to use `date_format` or `to_datetime()`, as described in the *There's more* section.

Let's inspect the parameters used in this recipe as defined in the official `pandas.read_csv()` documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html):

- **filepath_or_buffer**: This is the first positional argument and the only required field needed (at a minimum) to read a CSV file. Here, you passed the Python path object named `filepath`. This can also be a string that represents a valid file path, such as `'../datasets/Ch1/movieboxoffice.csv'` or a URL that points to a remote file location, such as an Amazon S3 bucket (we will examine this later in the *Reading data from URLs* recipe in this chapter).
- **sep**: This takes a string to specify which delimiter to use. The default is a comma delimiter (,) which assumes a CSV file. If the file is separated by another delimiter, such as a pipe (|) or semicolon (;), then the argument can be updated, such as `sep="|"` or `sep=";"`.
- **delimiter**: This is an alias to `sep`, which can be used as well as a parameter name.
- **header**: In this case, you specified that the first row (0) value contains the header information. The default value is `infer`, which usually works as-is in most cases. If the CSV does not contain a header, then you specify `header=None`. If the CSV has a header but you prefer to supply custom column names, then you need to specify `header=0` and overwrite it by providing a list of new column names to the `names` argument.
- **parse_dates**: In the recipe, you provided a list of column positions using `[0]`, which specified that only the first column (by position) should be parsed. The `parse_dates` argument can take a list of column names, such as `["Date"]`, or a list of column positions, such as `[0, 3]`, indicating the first and the fourth columns. If you only intend to parse the index column(s) specified in the `index_col` parameter, you only need to pass `True` (Boolean).
- **index_col**: You specified that the first column by position (`index_col=0`) will be used as the DataFrame index. Alternatively, you could provide the column name as a string (`index_col='Date'`). The parameter also takes in a list of integers (positional indices) or strings (column names), which would create a `MultiIndex` object.

- `usecols`: The default value is `None`, which includes all the columns in the dataset. Limiting the number of columns to only those that are required results in faster parsing and overall lower memory usage, since you only bring in what is needed. The `usecols` argument can take a list of column names, such as `['Date', 'DOW', 'Daily', 'Percent Diff', 'Forecast']` or a list of *positional indices*, such as `[0, 1, 3, 7, 6]`, which would produce the same result.

Recall that you specified which columns to include by passing a list of column names to the `usecols` parameter. These names are based on the file header (the first row of the CSV file). Generally, the combination of `usecols` and header is especially useful when working with large files.

If you provide custom header names, you cannot reference the original names in the `usecols` parameter; this will produce the following error: `ValueError: Usecols do not match columns`. Instead, you can use column positions or rename the columns in two steps. The following are two approaches to handle this:

```
col_names = ['Date', 'Day Name', 'Daily B0', 'Forecast B0']
df = pd.read_csv(filepath,
                 parse_dates=[0],
                 index_col=0,
                 header=0,
                 names=col_names,
                 usecols=[0, 1, 3, 6])
```

By using column positions in `usecols`, you bypass the need for matching column names, which allows you to apply custom headers without encountering errors

Alternatively, you can accomplish this in two steps using a Python dictionary:

```
col_names = {
    'Date': 'Date',
    'DOW': 'Day Name',
    'Daily': 'Daily B0',
    'Forecast': 'Forecast B0'
}

df = pd.read_csv(filepath,
                 parse_dates=[0],
                 index_col=0,
                 usecols=col_names.keys())
df.rename(columns=col_names, inplace=True)
```

Using a dictionary to map original column names to new ones allows flexibility and clarity, especially when dealing with large datasets or when you need to maintain a reference to original column names.

There's more...

There are situations where `parse_dates` may not work (it just cannot parse the date). In such cases, the column(s) will be returned unchanged, and no error will be thrown. This is where the `date_format` parameter can be helpful.

The following code shows how `date_format` can be used:

```
ts = pd.read_csv(filepath,
                  parse_dates=[0],
                  index_col=0,
                  date_format="%d-%b-%Y",
                  usecols=[0,1,3, 7, 6])

ts.head()
```

The preceding code will print out the first five rows of the `ts` DataFrame, displaying a correctly parsed **Date** index.

Let's break it down. In the preceding code, since the date is stored as a string in the form `26-Apr-2021`, you passed `"%d-%b-%Y"` to reflect that:

- `%d` represents the day of the month, such as `01` or `02`
- `%b` represents the abbreviated month name, such as `Apr` or `May`
- `%Y` represents the year as a four-digit number, such as `2020` or `2021`

Other common string codes include the following:

- `%y` represents a two-digit year, such as `19` or `20`
- `%B` represents the month's full name, such as **January** or **February**
- `%m` represents the month as a two-digit number, such as `01` or `02`

For more information on Python's string formats for representing dates, visit <https://strftime.org>.

When dealing with more complex date formats, an alternative option is to use the `to_datetime()` function. The `to_datetime()` function is used to convert a string, integer, or float into a datetime object.

Initially, you will read the CSV data as is, then apply the `to_datetime()` function to parse the specific column(s) as desired. This is demonstrated in the following code:

```
ts = pd.read_csv(filepath,
                  index_col=0,
                  usecols=[0,1,3, 7, 6])

ts.index = pd.to_datetime(ts.index, format="%d-%b-%Y")
```

The last line, `ts.index = pd.to_datetime(ts.index, format="%d-%b-%Y")`, converts the index of the `ts` DataFrame into a `DatetimeIndex` object. Notice how we specified the data string format similar to what we did with the `date_format` parameter in the `read_csv()` function.

When dealing with mixed or inconsistent date formats, you can use `pd.to_datetime()` with the `errors='coerce'` parameter:

```
ts.index = pd.to_datetime(ts.index, format="%d-%b-%Y", errors='coerce')
```

This will convert unparseable dates to `NaT` (not a time), letting you identify problematic entries without halting your workflow.

Generally, you should consider `to_datetime()` when dealing with multiple date formats within the same dataset or when initial parsing with `read_csv()` does not yield the desired results.

See also

- For more on parsing dates, refer to *Chapter 5*
- To learn more about `pandas.Series.str.replace()`, read the official documentation at <https://pandas.pydata.org/docs/reference/api/pandas.Series.str.replace.html>

Reading data from an Excel file

To read data from an Excel file, you will need to use a different reader function from `pandas`. Generally, working with Excel files can be challenging, as they often include formatted multi-line headers, merged header cells, and embedded images. Additionally, Excel files may contain multiple worksheets with custom labels. Therefore, it is vital that you always inspect the Excel file's structure beforehand. A common scenario involves reading Excel files that contain data partitioned into multiple sheets, which is the focus of this recipe.

In this recipe, you will be using the `pandas.read_excel()` function and exploring various parameters available to ensure that the data is properly ingested as a `DataFrame` with a `DatetimeIndex` for time series analysis. In addition, you will explore different options to read Excel files with multiple sheets.

Getting ready

To use `pandas.read_excel()`, you will need to install an additional library for reading and writing Excel files. The `engine` parameter in `read_excel()` allows you to specify a library (engine) to use to process the Excel file. Depending on the Excel file extension you are working with (for example, `.xls` or `.xlsx`), different engines may be required, which may result in installing additional libraries.

The supported libraries (engines) for reading and writing Excel include `openpyxl`, `xlrd`, `odf`, `calamine`, and `pyxlsb`. When working with Excel files, the two most common libraries are usually `xlrd` and `openpyxl`.

The `xlrd` library only supports `.xls` files. So, if you are working with an older Excel format, such as `.xls`, then `xlrd` will do just fine. For newer Excel formats, such as `.xlsx`, you will need a different engine, and in this case, `openpyxl` would be the recommended choice.

To install using `pip`, run the following command:

```
>>> pip install openpyxl
```

We will be using the `sales_trx_data.xlsx` file, which you can download from the book's GitHub repository (refer to the *Technical requirements* section of this chapter). The file contains sales data split across two sheets (one for 2017 and one for 2018).

How to do it...

You will ingest the Excel file (`.xlsx`) using pandas and the `openpyxl` engine, leveraging some of the available parameters in `read_excel()`:

1. Import pandas and Path from `pathlib` to handle file paths:

```
import pandas as pd
from pathlib import Path
```

Use Path to define a path to the Excel file:

```
filepath = Path('../datasets/Ch1/sales_trx_data.xlsx')
```

Use `filepath.exists()` if you want to make sure the file exists:

```
filepath.exists()
>>
True
```

2. You can read the Excel file (`.xlsx`) using the `read_excel()` function. By default, pandas will only read from the first sheet. This is specified under the `sheet_name` parameter, which is set to `0` as the default value (`sheet_name=0`).

Before specifying which sheets to read, you can use the `pandas.ExcelFile` class to inspect the file and determine the number of sheets available. The `ExcelFile` class will provide additional methods and properties, such as `sheet_name`, which returns a list of sheet names:

```
excelfile = pd.ExcelFile(filepath)
excelfile.sheet_names

>> ['2017', '2018']
```

This shows that the file contains two sheets: 2017 and 2018.

If you have multiple sheets, you can specify which sheets you want to ingest by passing a list to the `sheet_name` parameter in `read_excel()`. The list can either be **positional** arguments, such as first, second, and fifth sheets using `[0, 1, 4]`, **sheet names** using `["Sheet1", "Sheet2", "Sheet5"]`, or a **combination** of both, such as first sheet, second sheet, and a sheet named "Revenue" using `[0, 1, "Revenue"]`.

3. You can use sheet positions to read both the first and second sheets (0 and 1 indexes). This will return a Python dictionary object with two DataFrames. Note that the returned dictionary (key-value pair) has numeric keys (0 and 1) representing the first and second sheets (positional index), respectively:

```
ts = pd.read_excel(filepath,
                    engine='openpyxl',
                    index_col=1,
                    sheet_name=[0,1],
                    parse_dates=True)

ts.keys()

>> dict_keys([0, 1])
```

When specifying sheets by position, pandas uses zero-based indexing (the first sheet is at position 0). Here, `index_col=1` sets the second column (in our Excel file, it is the date column) as the DataFrame index. Adjust this value if your date column is in a different position.



If the required engine (such as `openpyxl`) is not installed, pandas will raise an `ImportError` error indicating the missing dependency. Similarly, specifying a sheet name that does not exist will raise a `ValueError` error. Always verify your file and sheet names before running your code

4. Alternatively, you can pass a list of sheet names. Notice that the returned dictionary keys are now strings and represent the sheet names, as shown in the following code:

```
ts = pd.read_excel(filepath,
                    engine='openpyxl',
                    index_col=1,
                    sheet_name=['2017', '2018'],
                    parse_dates=True)

ts.keys()

>> dict_keys(['2017', '2018'])
```

Using sheet names (e.g., `'2017'`) instead of positions (e.g., `0`) makes your code more robust to changes in the Excel file structure.

5. If you want to read from all the available sheets, you will pass `None` instead. The keys for the dictionary, in this case, will represent sheet names:

```
ts = pd.read_excel(filepath,
                    engine='openpyxl',
                    index_col=1,
                    sheet_name=None,
```

```

                                parse_dates=True)

ts.keys()

>> dict_keys(['2017', '2018'])

```

The two DataFrames within the dictionary are identical (homogeneous-typed) in terms of their schema (column names and data types). You can inspect each DataFrame with `ts['2017'].info()` and `ts['2018'].info()`.

They both have a `DatetimeIndex` object, which you specified in the `index_col` parameter. The 2017 DataFrame consists of 36,764 rows, and the 2018 DataFrame consists of 37,360. In this scenario, you want to stack (combine) the two (think `UNION` in SQL) into a single DataFrame that contains all 74,124 rows and a `DatetimeIndex` that spans from 2017-01-01 to 2018-12-31.

- To combine the two DataFrames along the index axis (stacked one on top of the other), you will use the `pandas.concat()` function. The default behavior of the `concat()` function is to concatenate along the index axis (`axis=0`). Since both DataFrames share the same structure (identical columns and data types), you can safely stack them together using `concat()`. This approach assumes that all sheets you want to combine follow the same schema. In the following code, you will explicitly specify which DataFrames to concatenate:

```

ts_combined = pd.concat([ts['2017'],ts['2018']])
ts_combined.info()

>> <class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 74124 entries, 2017-01-01 to 2018-12-31
Data columns (total 4 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Line_Item_ID          74124 non-null  int64
1   Credit_Card_Number    74124 non-null  int64
2   Quantity              74124 non-null  int64
3   Menu_Item             74124 non-null  object
dtypes: int64(3), object(1)
memory usage: 2.8+ MB

```

- When you have multiple DataFrames returned (think multiple sheets), you can use the `concat()` function on the returned dictionary. In other words, you can combine the `concat()` and `read_excel()` functions in one statement. In this case, you will end up with a `MultiIndex` DataFrame where the first level is the sheet name (or number) and the second level is the `DatetimeIndex`. For example, by using the `ts` dictionary, you will get a two-level index: `MultiIndex([('2017', '2017-01-01'), ..., ('2018', '2018-12-31')], names=[None, 'Date'], length=74124)`.

To reduce the number of levels, you can use the `droplevel(level=0)` method to drop the first level following `concat()`, as follows:

```
ts_combined = pd.concat(ts).droplevel(level=0)
```

This will simplify the index for the `ts_combined` DataFrame.

8. If you are only reading one sheet, the behavior is slightly different. By default, `sheet_name` is set to `0`, which means it reads the first sheet. You can modify this and pass a different value (single value), either the sheet name (string) or sheet position (integer). When passing a single value, the returned object will be a pandas **DataFrame** and not a dictionary:

```
ts = pd.read_excel(filepath,
                    index_col=1,
                    sheet_name='2018',
                    parse_dates=True)

type(ts)
>> pandas.core.frame.DataFrame
```

If you pass a single value inside two brackets (e.g., `[1]` or `['2018']`), then pandas will interpret this differently and will return a **dictionary** that contains one DataFrame.

Note that you did not need to specify the engine in the last example. The `read_excel()` function will determine which engine to use based on the file extension, given that the appropriate library is installed (e.g., `openpyxl`). For example, if the required library for that engine is not installed, an `ImportError` error will indicate the missing dependency.

How it works...

The `pandas.read_excel()` function has many common parameters with the `pandas.read_csv()` function that you used earlier. The `read_excel` function can either return a DataFrame object or a dictionary of DataFrames. The dependency here is whether you are passing a single value (scalar) or a list to the `sheet_name` parameter:

- **Single value** for `sheet_name`: Returns a single DataFrame
- **List of sheet names or indices** for `sheet_name`: Returns a dictionary of DataFrames
- `sheet_name=None`: Reads all sheets and returns a dictionary

In the `sales_trx_data.xlsx` file, both sheets had the same schema (homogeneous-typed). The sales data was partitioned (split) by year, where each sheet contained sales for a particular year. In this case, concatenating the two DataFrames was a natural choice. The `pandas.concat()` function is like the `DataFrame.append()` function, in which the second DataFrame was added (appended) to the end of the first DataFrame. This should be similar in behavior to the **UNION** clause for those coming from a SQL background.

There's more...

An alternative method to reading an Excel file is using the `pandas.ExcelFile()` class, which returns an `ExcelFile` object. Earlier in this recipe, you used `ExcelFile` to inspect the number of sheets in the Excel file using its `sheet_name` attribute.

The `ExcelFile` class includes useful methods, such as the `parse()` method, which can parse an Excel sheet into a `DataFrame`. This functionality is similar to the `pandas.read_excel()` function, but provides more flexibility when working with multiple sheets.

In the following example, you will use the `ExcelFile` class to parse the first sheet, set the first column as an index, and print the first five rows of the resulting `DataFrame`:

```
excelfile = pd.ExcelFile(filepath)
excelfile.parse(sheet_name='2017',
                index_col=1,
                parse_dates=True).head()
```

Upon inspecting the output you can conclude that the `ExcelFile.parse()` method is equivalent to the `pandas.read_excel()` function. Using `ExcelFile` is particularly useful when you need to read multiple sheets from the same Excel file repeatedly. It avoids rereading the file each time and can be more efficient for large workbooks.

See also

For more information on `pandas.read_excel()` and `pandas.ExcelFile()`, please refer to the official documentation:

- *pandas.read_excel*: https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html
- *pandas.ExcelFile.parse*: <https://pandas.pydata.org/docs/reference/api/pandas.ExcelFile.parse.html>

Reading data from URLs

Files can either be downloaded and stored locally on your machine or accessed directly from a remote server or cloud location. In the earlier two recipes, *Reading from CSV and other delimited files* and *Reading data from an Excel file*, the data files were stored locally.

Many of the `pandas` reader functions can read data directly from remote locations by passing a URL as the file path. For example, `read_csv()` and `read_excel()` can take a URL to read a file accessible via the internet. This capability is particularly useful for automating workflows, accessing large datasets stored in the cloud, or integrating with web-based APIs.

In this recipe, you will read a CSV file using `pandas.read_csv()` and Excel files using `pandas.read_excel()` from remote locations, such as GitHub and Amazon S3 buckets (private and public buckets). You will also read data directly from an HTML page into a pandas DataFrame.

When working with remote files, keep in mind that network connectivity, file permissions, and authentication requirements (e.g., for private Amazon S3 buckets) may impact the ability to access and load the data.

Getting ready

To read files from S3 buckets and HTML pages using pandas, you will need to install some additional libraries. This section outlines the required installations and their use cases:

- **AWS SDK for Python (Boto3):**
 - Required for accessing files from S3 buckets. Alternatively, pandas offers the `storage_options` parameter in its reader functions, allowing you to read from S3 without directly using the Boto3 library.
- **S3 URL support:**
 - To use S3 URLs (e.g., `s3://bucket_name/path-to-file`) in pandas, you need to install the `s3fs` library. This library enables seamless interaction with S3-compatible storage directly with pandas without requiring Boto3.
- **HTML parsing:**
 - To parse HTML content using `pandas.read_html()`, you need to install an HTML parser. pandas supports two parsing engines:
 - `lxml`: pandas will use this as the default if it is installed.
 - `html5lib`: If `lxml` is not installed, pandas will fall back to this engine. Using `html5lib` requires the `Beautiful Soup` library (`beautifulsoup4`).

To install using `pip`, you can use the following command:

```
>>> pip install boto3 s3fs lxml html5lib
```

How to do it...

This recipe demonstrates different scenarios for reading data from online (remote) sources using pandas. First, import the pandas library used throughout this recipe:

```
import pandas as pd
```


Reading data from GitHub

Sometimes, you may find useful public data on GitHub that you want to use and read directly (without downloading). One of the most common file formats on GitHub is CSV files. Let's start with the following steps:

1. To read a CSV file from GitHub, you will need the URL to the raw content. If you copy the file's GitHub URL from the browser and use it as the file path, you will get a URL that looks like this: `https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/datasets/Ch1/AirQualityUCI.csv`. This URL points to the file's web page on GitHub, not its raw content. If you use this URL directly with `pd.read_csv()`, it will throw an error:

```
url = 'https://github.com/PacktPublishing/Time-Series-Analysis-with-
Python-Cookbook-Second-Edition/blob/main/datasets/Ch1/AirQualityUCI.csv'
pd.read_csv(url)

>>
ParserError: Error tokenizing data. C error: Expected 1 fields in line ,
saw 24
```

Navigate to the file in GitHub and click the “raw” button to obtain the raw content URL, which should look like `https://raw.githubusercontent.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/refs/heads/main/datasets/Ch1/AirQualityUCI.csv`:

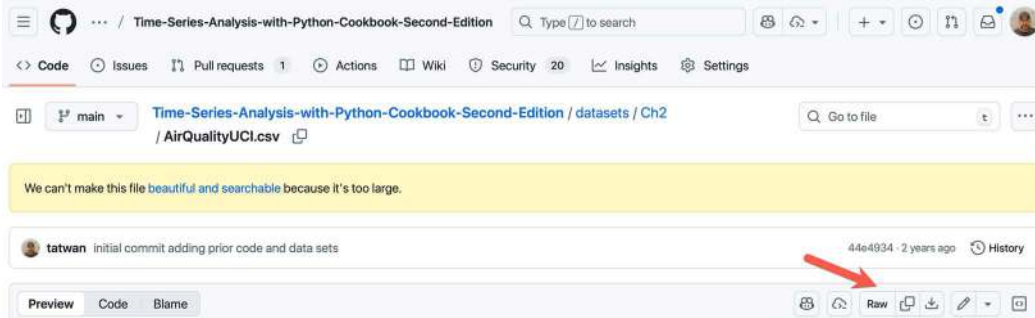


Figure 1.1: The GitHub page for the CSV file; note the View raw button

In Figure 1.1, notice that the values are not comma-separated (not a comma-delimited file); instead, the file uses a semicolon (;) as a delimiter to separate the values.

The first column in the file is the Date column, which you will need to parse as a `DatetimeIndex` using the `parse_date` and `index_col` parameters.

Pass the new URL to `pandas.read_csv()`:

```
url = 'https://raw.githubusercontent.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/refs/heads/main/datasets/Ch1/AirQualityUCI.csv'

df = pd.read_csv(url,
                  delimiter=';',
                  parse_dates=['Date'],
                  index_col='Date')
print(df.iloc[:5, 0:5])
>>
```

	Time	CO(GT)	PT08.S1(CO)	NMHC(GT)	C6H6(GT)
Date					
10/03/2004	18:00:00	2.6	1360.00	150	11.881723
10/03/2004	19:00:00	2.0	1292.25	112	9.397165
10/03/2004	20:00:00	2.2	1402.00	88	8.997817
10/03/2004	21:00:00	2.2	1375.50	80	9.228796
10/03/2004	22:00:00	1.6	1272.25	51	6.518224

You have successfully ingested the data from the CSV file from GitHub into a DataFrame.

Reading data from a public S3 bucket

AWS supports multiple URL formats for accessing files: **virtual-hosted-style** URLs such as `https://bucket-name.s3.Region.amazonaws.com/keyname`, **path-style** URLs such as `https://s3.Region.amazonaws.com/bucket-name/keyname`, and **S3 protocol** using `S3://bucket/keyname`. Here are examples of how these different URLs may look for our file:

- A **virtual-hosted-style** URL or an **object URL**: `https://tscookbook.s3.us-east-1.amazonaws.com/AirQualityUCI.xlsx`
- A **path-style** URL: `https://s3.us-east-1.amazonaws.com/tscookbook/AirQualityUCI.xlsx`
- An **S3 protocol**: `s3://tscookbook/AirQualityUCI.xlsx`

In this example, you will read the `AirQualityUCI.xlsx` file from S3, which has only one sheet. The file contains the same data as `AirQualityUCI.csv`, which you read earlier from GitHub.

Note that in the URL, you do not need to specify the region as `us-east-1`. The `us-east-1` region, which represents US East (Northern Virginia), is an *exception*. This is not the case for other regions.

In the following code, all three URLs will produce the same output:

```
# Virtual-hosted-style URL
url1 = 'https://tscookbook.s3.amazonaws.com/AirQualityUCI.xlsx'

# Path-style URL
url2 = 'https://s3.us-east-1.amazonaws.com/tscookbook/AirQualityUCI.xlsx'
```

```
# S3 protocol URL
url3 = 's3://tscookbook/AirQualityUCI.xlsx'

# Use any of these URLs to read the Excel file
df = pd.read_excel(url3,
                    index_col='Date',
                    parse_dates=True)
```

Ensure that the `s3fs` library is installed to enable seamless access to S3 files using the `s3://` protocol. If you get the `ImportError: Install s3fs to access S3 error`, it indicates that either you do not have the `s3fs` library installed, or possibly you are not using the right Python/Conda environment. Please check the *Getting ready* section on how to install these dependencies.

Reading data from a private S3 bucket

When accessing files in a private S3 bucket, you need to authenticate using your AWS credentials. This section demonstrates two methods for achieving this: using **pandas with storage_options** and **Boto3**.

Using pandas with storage_options

When reading files from a private S3 bucket, the `storage_options` parameter in pandas' I/O functions allows you to send additional information, such as credentials, to authenticate to a cloud service.

You will need to pass a dictionary (key-value pair) to provide the additional information along with the request, such as username, password, access keys, and secret keys to `storage_options` as in `{"username": username, "password": password}`.

Now, you will read the `AirQualityUCI.csv` file, located in a private S3 bucket:

1. Start by storing your AWS credentials in a **config** file (e.g., `aws.cfg`) to avoid hardcoding them in your script and getting them exposed. You will use the `configparser` module to read the credentials from the configuration file and store them in a Python variable:

```
# Example aws.cfg file
[AWS]
aws_access_key=your_access_key
aws_secret_key=your_secret_key
```

Always keep your `aws.cfg` file secure and never commit it to public repositories.

Load the `aws.cfg` file using `config.read()`:

```
import configparser
config = configparser.ConfigParser()
config.read('aws.cfg')

AWS_ACCESS_KEY = config['AWS']['aws_access_key']
AWS_SECRET_KEY = config['AWS']['aws_secret_key']
```

2. The AWS access key ID and secret access key are now stored in the `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` variables, respectively. Use `pandas.read_csv()` to read the CSV file and update the `storage_options` parameter by passing your credentials, as shown in the following code:

```
s3uri = "s3://tscookbook-private/AirQuality.csv"

df = pd.read_csv(s3uri,
                 index_col='Date',
                 parse_dates=True,
                 storage_options= {
                     'key': AWS_ACCESS_KEY,
                     'secret': AWS_SECRET_KEY
                 })

print(df.iloc[:5, 0:5])
```

If your data uses commas as decimal separators (e.g., 2,6 instead of 2.6), then consider using the `decimal=','` parameter in `read_csv()` to ensure correct numeric parsing:

```
s3uri = "s3://tscookbook-private/AirQuality.csv"

df = pd.read_csv(s3uri,
                 index_col='Date',
                 parse_dates=True,
                 decimal=',',
                 storage_options= {
                     'key': AWS_ACCESS_KEY,
                     'secret': AWS_SECRET_KEY
                 })

print(df.iloc[:5, 0:5])
```

Using Boto3

Alternatively, you can use the **AWS SDK for Python (Boto3)** to achieve similar results. The `boto3` Python library provides you with more control and additional capabilities (beyond just reading and writing to S3):

1. Pass your credentials stored earlier in `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` and pass them to AWS, using `boto3` to authenticate:

```
import boto3
bucket = "tscookbook-private"
client = boto3.client("s3",
                     aws_access_key_id =AWS_ACCESS_KEY,
                     aws_secret_access_key = AWS_SECRET_KEY)
```

The code creates a `client` object that provides access to various methods for interacting with the Amazon S3 service, such as creating, deleting, and retrieving bucket information, and more. In addition, Boto3 offers two levels of APIs: **client** and **resource**. In the preceding example, you used the **client** API. The **client** API is a low-level service access interface that gives you more granular control, for example, `boto3.client("s3")`. The **resource** API is a higher-level object-oriented interface (an abstraction layer), for example, `boto3.resource("s3")`.

In *Chapter 3*, you will explore the **resource** API interface when writing to S3. For now, you will use the client interface.

2. Use the `get_object` method to retrieve the data. Just provide the bucket name and a key. The key here is the actual filename:

```
data = client.get_object(Bucket=bucket, Key='AirQuality.csv')
df = pd.read_csv(data['Body'],
                 decimal=',',
                 index_col='Date',
                 parse_dates=True)

print(df.iloc[:5, 0:5])
```

The `get_object()` method returns a dictionary (key-value pair), and the content you are interested in is stored under the `Body` key. You then pass `data['Body']` to the `read_csv()` function, which loads the response stream (`StreamingBody`) into a `DataFrame`.

The `storage_options` method is simpler and integrates directly with **pandas**, making it ideal for quick data loading tasks. In contrast, **Boto3** provides more control over S3 interactions and is suitable for complex workflows requiring additional AWS functionality.

Reading data from HTML

This recipe demonstrates how to scrape and parse HTML tables from a web page into pandas `DataFrames`.

The **pandas** library offers an elegant way to read HTML tables and convert the content into a pandas `DataFrame` using the `pandas.read_html()` function:

1. You will extract HTML tables from Wikipedia's COVID-19 pandemic tracking page, which provides cases by country and territory (https://en.wikipedia.org/wiki/COVID-19_pandemic_by_country_and_territory):

```
import pandas as pd
from io import StringIO
import requests

url = "https://en.wikipedia.org/wiki/COVID-19_pandemic_by_country_and_territory"
headers = {'User-Agent': 'Mozilla/5.0'}
```

```

response = requests.get(url, headers=headers)
html = StringIO(response.text)

results = pd.read_html(html)
print(len(results))
>>
67

```

Wikipedia now blocks default Python User-Agents, requiring custom headers via requests. Additionally, pandas 2.1.0+ deprecated passing HTML strings directly to `pd.read_html()`; wrap the string in `StringIO()` instead.

2. The `pandas.read_html()` function returned a list of DataFrames, one for each HTML table found in the URL. Keep in mind that the website's content is dynamic and gets updated regularly, and the results may vary. In our case, it returned 68 DataFrames. The DataFrame at index 15 contains summary information on COVID-19 cases and deaths by region. Grab the DataFrame (at index 15) and assign it to the `df` variable, and print the returned columns:

```

df = results[15]
df.columns

>>
Index(['Region[30]', 'Total cases', 'Total deaths', 'Cases per million',
       'Deaths per million', 'Current weekly cases', 'Current weekly
       deaths',
       'Population millions', 'Vaccinated %[31]'],
      dtype='object')

```

3. Display the first three rows for the **Total cases**, **Total deaths**, and **Cases per million** columns:

```

cols = ['Region[30]', 'Total cases', 'Total deaths', 'Cases per million']
print(df[cols].head(3))

>>
   Region[30]  Total cases  Total deaths  Cases per million
0  European Union    179537758      1185108           401363
1   North America    103783777      1133607           281404
2    Other Europe     57721948       498259           247054

```

How it works...

Most of the pandas reader functions accept a URL as a path. Examples include the following:

- `pandas.read_csv()`
- `pandas.read_excel()`
- `pandas.read_parquet()`
- `pandas.read_table()`
- `pandas.read_pickle()`
- `pandas.read_orc()`
- `pandas.read_stata()`
- `pandas.read_sas()`
- `pandas.read_json()`

The URL needs to be one of the valid URL schemes that pandas supports, which includes `http` and `https`, `ftp`, `s3`, `gs`, or the `file` protocol.

The `read_html()` function is great for scraping websites that contain data in HTML tables. It inspects the HTML and searches for all the `<table>` elements within the HTML. In HTML, table rows are defined with the `<tr>` and `</tr>` tags and headers with the `<th>` and `</th>` tags. The actual data (cell) is contained within the `<td>` and `</td>` tags. The `read_html()` function looks for `<table>`, `<tr>`, `<th>`, and `<td>` tags, converts the content into a `DataFrame`, and assigns the columns and rows as they were defined in the HTML. If an HTML page contains more than one `<table>` or `</table>` tag, `read_html` will return them all, and you will get a list of `DataFrames`.

The following code demonstrates how `pandas.read_html()` works:

```
from io import StringIO
import pandas as pd

html = """
<table>
  <tr>
    <th>Ticker</th>
    <th>Price</th>
  </tr>
  <tr>
    <td>MSFT</td>
    <td>230</td>
  </tr>
  <tr>
    <td>APPL</td>
    <td>300</td>
  </tr>
  <tr>
```

```

        <td>MSTR</td>
        <td>120</td>
    </tr>
</table>
</body>
</html>
"""

df = pd.read_html(StringIO(html))
df[0]

>>
   Ticker  Price
0    MSFT    230
1    APPL    300
2    MSTR    120

```



Passing HTML literal strings

Starting from pandas version 2.1.0, you will need to wrap HTML code in `io.StringIO` when passing a literal HTML string to `read_html()`. The `StringIO(<HTML CODE>)` creates an in-memory file-like object from the HTML string that can be passed directly to the `read_html()` function.

In the preceding code, the `read_html()` function reads the HTML content from the file-like object and converts an HTML table, represented between the `<table>` and `</table>` tags, into a pandas DataFrame. The headers between the `<th>` and `</th>` tags represent the column names of the DataFrame, and the content between the `<tr>` and `<td>`, and `</td>` and `</tr>` tags represent the row data of the DataFrame. Note that if you go ahead and delete the `<table>` and `</table>` tags, you will get the `ValueError: No tables found error`.

There's more...

The `read_html()` function has an optional `attr` argument, which takes a dictionary of valid HTML `<table>` attributes, such as `id` or `class`. For example, you can use the `attr` parameter to narrow down the tables returned to those that match the `sortable` class attribute, as in `<table class="sortable">`. The `read_html` function will inspect the entire HTML page to ensure you target the right set of attributes.

In the previous exercise, you used the `read_html` function on the COVID-19 Wikipedia page, and it returned 68 tables (DataFrames). The number of tables will probably increase as time goes by, as Wikipedia gets updated. You can narrow down the result set and guarantee some consistency by using the `attr` option. First, start by inspecting the HTML code using your browser.

You will see that several of the `<table>` elements have multiple classes listed, such as `sortable`. You can look for other unique identifiers:

```
<table
  class="wikitable sortable mw-datatable covid19-countrynames jquery-
  tablesorter"
  id="thetable"
  style="text-align:right;">
```

Note, if you get an `html5lib not found` error, you will need to install both `html5lib` and `beautifulSoup4`.

To install using `conda`, use the following:

```
conda install html5lib beautifulSoup4 -y
```

To install using `pip`, use the following:

```
pip install html5lib beautifulSoup4
```

Now, let's use the `sortable` class and request the data again:

```
import pandas as pd
url = "https://en.wikipedia.org/wiki/COVID-19_pandemic_by_country_and_
territory"
headers = {'User-Agent': 'Mozilla/5.0'}
response = requests.get(url, headers=headers)
html = StringIO(response.text)

df = pd.read_html(html, attrs={'class': 'sortable'})
len(df)
>>
5

df[3].columns
>>
Index(['Region[30]', 'Total cases', 'Total deaths', 'Cases per million',
      'Deaths per million', 'Current weekly cases', 'Current weekly deaths',
      'Population millions', 'Vaccinated %[31]'],
      dtype='object')
```

The list returned a smaller subset of tables (from 68 down to 5).

When reading data from URLs, network-related issues can cause failures. A good practice is to implement error handling with try/except blocks. Here is an example:

```
# Error handling example
try:
    df = pd.read_csv(s3uri,
                     index_col='Date',
                     parse_dates=True,
                     storage_options= {
                         'key': AWS_ACCESS_KEY,
                         'secret': AWS_SECRET_KEY
                     })
except (pd.errors.ParserError, OSError, requests.exceptions.RequestException)
as e:
    print(f"Error reading URL: {e}")
    # Implement fallback strategy or retry logic
```

If you encounter a permissions or access error when reading from S3 (raised as `OSError`), double-check your AWS credentials and bucket policy. For network errors, verify your internet connection and that the URL is correct. Common error messages include “**Access Denied**” (credentials issue), “**NoSuchBucket**” (incorrect bucket name), or “**SignatureDoesNotMatch**” (credentials mismatch).

See also

- For more information, please refer to the official `pandas.read_html` documentation: https://pandas.pydata.org/docs/reference/api/pandas.read_html.html.

Reading data from Parquet files

Parquet files have emerged as a popular choice for storing and processing large datasets efficiently, especially in data engineering and big data analytics. Initially developed by Twitter and Cloudera, Parquet was later contributed to the **Apache Foundation** as an open source columnar file format. The focus of Parquet is to prioritize fast data retrieval and efficient compression. Its design specifically caters to analytical workloads and serves as an excellent option for partitioning data, which you can explore in the *Bonus Recipe* section on GitHub. As a result, Parquet has become the de facto standard for modern data architectures and cloud storage solutions.

In this recipe, you will learn how to read Parquet files using pandas and learn how to query a specific partition for efficient data retrieval.

Getting ready

You will be working with Parquet files containing weather data from the **Los Angeles Airport stations**, sourced from the **National Oceanic and Atmospheric Administration (NOAA)**. These files, stored in the `datasets/Ch1/LA_weather.parquet/` folder, contain weather readings from 2010 to 2023 and partitioned by year (14 subfolders).

You will use the `pandas.read_parquet()` function, which requires you to install a Parquet engine to process the files. You can install either **fastparquet** or **PyArrow**, with the latter being the default choice for pandas.

To install PyArrow using **pip**, run the following command:

```
pip install pyarrow
```

How to do it...

The PyArrow library allows you to pass additional arguments (`**kwargs`) to the `pandas.read_parquet()` function, thereby providing more options when reading files, as you will explore.

Reading all partitions

The following steps are for reading all the partitions in the `LA_weather.parquet` folder in one go:

1. Create a path to reference the Parquet folder, which contains the partitions, and pass it to the `read_parquet` function:

```
import pandas as pd
from pathlib import Path

file = Path('.././datasets/Ch1/LA_weather.parquet/')
df = pd.read_parquet(file, engine='pyarrow')
```

2. You can validate and check the schema using the `info()` method:

```
df.info()
```

This should produce the following output:

Reading specific partitions

The following steps explain how to read a specific partition or set of partitions using the `filters` argument and specify columns using the `columns` argument from the PyArrow library:

1. As the data is partitioned by year, you can utilize the `filters` argument to specify a particular partition. In the following, you will only read the partition for the year 2012:

```
filters = [('year', '==', 2012)]
df_2012 = pd.read_parquet(file,
                           engine='pyarrow',
                           filters=filters)
```

Note that partition filtering works when your Parquet dataset uses directory-based partitioning (e.g., `year=2021/`). If your folder structure doesn't match the partition column, the `filters` argument won't have any effect.

2. To read a set of partitions, such as for the years 2021, 2022, and 2023, you can utilize any of the following options, which will produce similar results:

```
# These filter options all select data from 2021-2023, using different approaches:

# Option 1: Select all years greater than 2020
filters = [('year', '>', 2020)]

# Option 2: Select all years greater than or equal to 2021
filters = [('year', '>=', 2021)]

# Option 3: Explicitly List the years we want to include
filters = [('year', 'in', [2021, 2022, 2023])]
```

After defining the `filters` object, you can assign it to the `filters` argument within the `read_parquet()` function, as demonstrated here:

```
df = pd.read_parquet(file,
                     engine='pyarrow',
                     filters=filters)
```

3. Another useful argument is `columns`, which allows you to specify the column names you want to retrieve as a Python list:

```
columns = ['DT', 'year', 'TMAX']
df = pd.read_parquet(file,
                     engine='pyarrow',
                     filters=filters,
                     columns=columns)
```

In the preceding code, the `read_parquet()` function will only retrieve the specified columns ('DT', 'year', and 'TMAX') from the Parquet file, using the defined filters. You can validate the results by running `df.head()` and `df.info()`.

Notice how the memory usage has significantly reduced when narrowing your selection to only the necessary data by specifying the partitions and columns for your data analysis.

How it works...

There are several advantages to working with the Parquet file format, especially when dealing with large data files. The columnar-oriented format of Parquet offers faster data retrieval and efficient compression, making it ideal for cloud storage and reducing storage costs. Parquet employs advanced techniques and algorithms for data encoding, leading to improved compression ratios.

Partitioning organizes Parquet data into subfolders based on specific column values, such as year. For example, a dataset partitioned by year will have subfolders named `year=2010`, `year=2011`, and so on. Each subfolder contains Parquet files that store the data for that specific partition.

Figure 1.2 shows an example of a folder structure for a dataset stored as a Parquet file partitioned by year. Each year has its own subfolder, and within each subfolder, there are individual files.

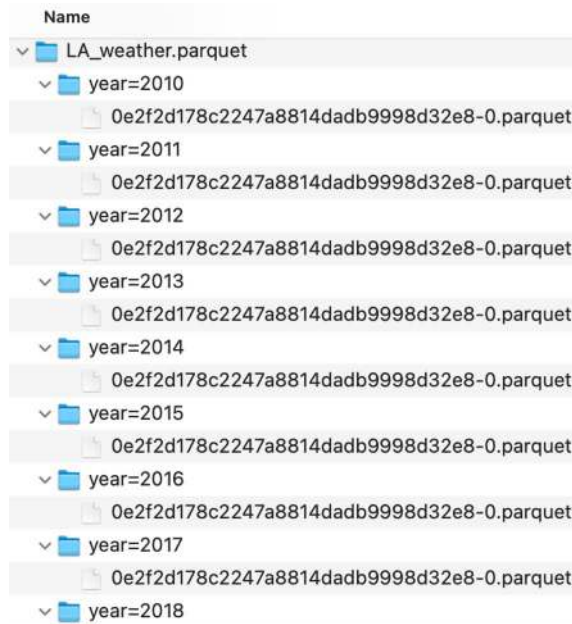


Figure 1.2: Example of a folder structure for a Parquet dataset partitioned by year

Partitioning allows for more efficient querying by scanning only the relevant subfolders, thereby reducing the amount of data read into memory. For instance, querying data for the year 2010 will only access files in the `year=2010` subfolder, skipping data from other years. This is particularly beneficial for time series data, where queries often filter by time periods.

Parquet files are referred to as “**self-described**” since each file contains the encoded data and additional **metadata** in the footer section. The metadata includes the version of the Parquet format, data schema, and structure (such as column types), and other statistical information, such as the minimum and maximum values for the columns. Consequently, when writing and reading Parquet datasets using pandas, you will notice that the DataFrame schema is preserved.

There are some key parameters you need to be familiar with based on the official pandas documentation for the `read_parquet()` reader function, which you can find on their official page (https://pandas.pydata.org/docs/reference/api/pandas.read_parquet.html). You already used some of these parameters in the previous *How to do it...* section. The following shows the key parameters you already used, as described in the list that follows:

```
read_parquet(  
    path: 'FilePath | ReadBuffer[bytes]',  
    engine: 'str' = 'auto',  
    columns: 'list[str] | None' = None,  
    **kwargs,  
)
```

- **path:** This is the first positional argument and the only required field needed (at a minimum) to read a Parquet file. In our example, you passed the Python path object named `file` as the argument. You can also pass a valid URL that points to a remote Parquet file location, such as an Amazon S3 bucket.
- **engine:** The default value is `auto` if you do not pass any arguments to the engine parameter. The other two valid options are `pyarrow` and `fastparquet`, depending on which engine you installed. In our example, you installed the `PyArrow` library (see the *Getting ready* section). The `auto` option will first attempt to load `pyarrow`, and then fall back to `fastparquet` if it's not available.
- **columns:** Here, you can specify the columns you would like to limit when reading. You will pass this as a Python list even if you select just one column. In our example, you defined a columns variable as `columns = ['DATE', 'year', 'TMAX']` and then passed it as an argument.
- ****kwargs:** This indicates that additional arguments can be passed to the engine. In our case, we used the `PyArrow` library; hence, the `read_parquet()` pandas function will pass these arguments to the `pyarrow.parquet.read_table()` function from the `PyArrow` library. In the previous examples, we passed a list with filtering criteria to the `filters` parameter in `pyarrow.parquet.read_table()`. For additional parameters that you can utilize, you can check the official documentation for `pyarrow.parquet.read_table()` here: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.read_table.html.

When working with extremely large Parquet files that exceed available RAM, you can use **memory mapping**:

```
# Memory-map large Parquet files  
df = pd.read_parquet(file, memory_map=True)
```

Memory mapping allows pandas (via `PyArrow`) to access Parquet files directly from disk, which can improve file reading performance by reducing I/O overhead for very large files. With memory mapping, pandas maps the file and loads data as needed during the read operation, instead of reading the entire file into memory all at once.

There's more...

Recall in the *Getting ready* section that you installed the PyArrow library as the backend engine for working with Parquet files with pandas. When you used the `read_parquet()` reader function in pandas, the `pyarrow` engine was the default.

Since you have already installed the library, you can utilize it directly to work with Parquet files in a similar manner as you did using pandas. Instead of the `pandas.read_parquet()` function, you will use the `pyarrow.parquet.read_table()` function, as shown in the following:

```
import pyarrow.parquet as pq
from pathlib import Path

file = Path('../..../datasets/Ch1/LA_weather.parquet/')
table = pq.read_table(file, filters=filters, columns=columns)
```

The `table` object is an instance of the `pyarrow.Table` class. You can validate this using the following code:

```
import pyarrow as pa
isinstance(table, pa.Table)
>> True
```

The `table` object contains many useful methods, including the `to_pandas()` method to convert the object into a pandas `DataFrame`:

```
df = table.to_pandas()
```

The following illustrates this further to show the similarities between the `read_table()` and `read_parquet()` functions:

```
columns = ['DATE', 'year', 'TMAX']
filters = [('year', 'in', [2021, 2022, 2023])]

tb = pq.read_table(file,
                  filters=filters,
                  columns=columns,
                  use_pandas_metadata=True)

df_pa = tb.to_pandas()
df_pd = pd.read_parquet(file,
                      filters=filters,
                      columns=columns,
                      use_pandas_metadata=True)
```

Both `df_pa` and `df_pd` are equivalent.

The `use_pandas_metadata=True` ensures that PyArrow preserves pandas dtypes (e.g., category for year).

The PyArrow library provides a low-level interface, while pandas provides a high-level interface that is built on top of PyArrow. The same applies if you decide to install the `fastparquet` library instead. Note that PyArrow is the Python implementation of **Apache Arrow**, an open source project for in-memory data (columnar memory). While Apache Parquet specifies the columnar file format for efficient storage and retrieval, Apache Arrow allows us to process such large columnar datasets in memory efficiently as well.

See also

- To learn more about the `pandas.read_parquet()` function, you can read the latest documentation here: https://pandas.pydata.org/docs/reference/api/pandas.read_parquet.html.

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



2

Reading Time Series Data from Databases

Time series data can be stored in a variety of formats, including flat files (such as CSV, Excel, or Parquet) or database systems. In *Chapter 1*, we explored reading data from different file types, whether stored locally (on-premise) or remotely in a cloud service such as AWS S3. In this chapter, we will focus on how to retrieve time series data from different database systems.

Databases extend storage capabilities beyond flat files, supporting structured, semi-structured, and unstructured data—including text, images, and media files. They are optimized for efficient read-and-write operations at scale, making them ideal for handling terabytes or even petabytes of data with efficient and optimized data retrieval capabilities.

Modern data architectures typically distinguish between several database paradigms:

- **Transactional (OLTP) databases** are optimized for handling day-to-day business operations with many small, concurrent transactions. Relational databases such as PostgreSQL, MySQL, and SQL Server traditionally serve this role.
- **Analytical (OLAP) databases** such as traditional data warehouses and cloud data warehouses (e.g., Snowflake, Amazon Redshift) are designed specifically for complex queries on large volumes of structured data, optimized for business intelligence, reporting, and analytics.
- **Data lakes** serve a different purpose than traditional OLAP systems. They store raw data in various formats (structured, semi-structured, or unstructured) and are primarily designed as flexible repositories rather than query engines. They often leverage object storage systems such as Azure Blob Storage, Google Cloud Storage, or AWS S3 (covered in *Chapter 1* and *Chapter 3*).
- **Data lakehouses** are an emerging hybrid architecture that combines the structure and query performance of data warehouses with the flexibility and cost-efficiency of data lakes, allowing analytical processing directly on data lake storage.

Specialized databases have emerged to handle specific data types and access patterns:

- **Time series databases** (e.g., InfluxDB, TimescaleDB) are optimized for high-frequency time-stamped data, offering efficient storage and retrieval for metrics, IoT, and financial data.

- Document databases (e.g., MongoDB) excel at storing semi-structured data where schema flexibility is important.

In this chapter, we will use **pandas** to read time series data from different database systems, including:

- Relational (SQL) databases (e.g., PostgreSQL)
- NoSQL databases (e.g., MongoDB, a document database)
- Cloud data platforms (e.g., Snowflake, Amazon Redshift)

Before diving into specific database techniques, let's understand how the chapters in this book connect to form a complete time series analysis workflow:

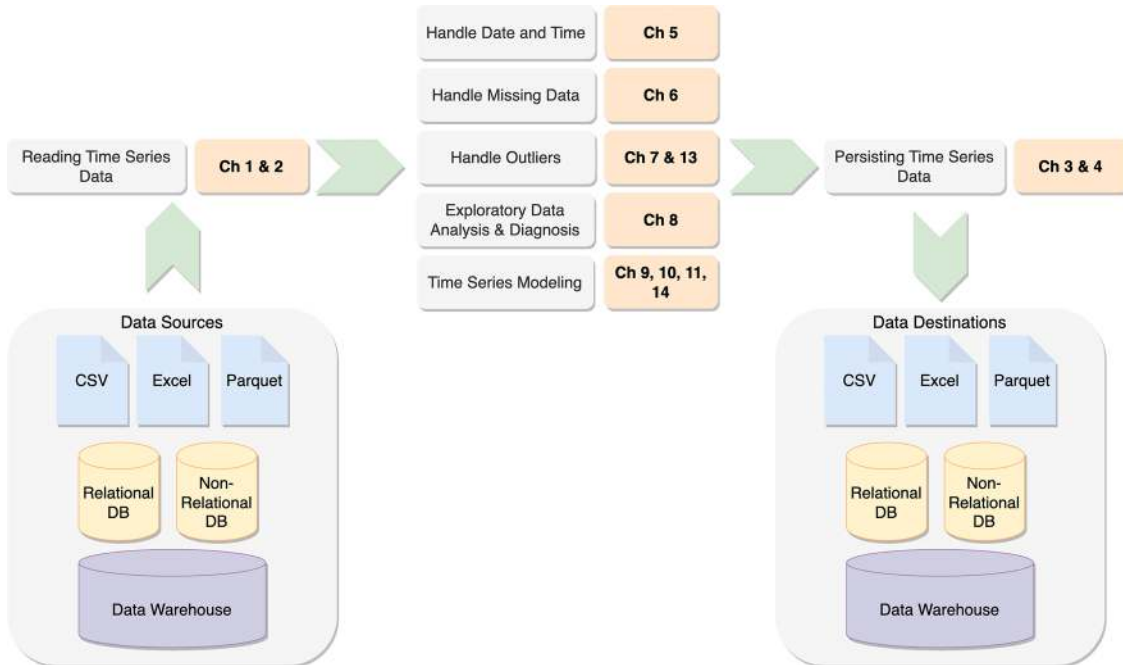


Figure 2.1: Time series data analysis workflow: from ingestion to persistence

As illustrated in *Figure 2.1*, the time series analysis process typically begins with **data acquisition** (*Chapters 1 and 2*) from various sources, continues through multiple **processing** and **modeling** steps (*Chapters 5-14*), and concludes with **persisting** results (*Chapters 3 and 4*). This chapter focuses on the left side of this workflow – extracting time series data from database systems ranging from traditional relational databases to specialized time series databases and cloud data platforms.

Throughout the recipes of this chapter, you will create pandas DataFrames with `DatetimeIndex` extracting time series data from various database systems:

- Reading data from a relational database
- Reading data from Snowflake
- Reading data from MongoDB



Don't miss the GitHub-exclusive recipe— *Chapter 2-1: Reading data from a time series database*— available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Technical requirements

In this chapter, we will be using pandas 2.3.3 (released September 29, 2025) extensively.

You will be working with different types of databases, such as **PostgreSQL**, **Amazon Redshift**, **MongoDB**, **InfluxDB**, and **Snowflake**. You will need to install additional Python libraries to connect to these databases. The specific libraries needed for each database will be introduced in the relevant recipes.

You can also download the Jupyter notebooks from this book's GitHub repository (<https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>) to follow along.

An important security practice when working with databases is to never hardcode credentials directly in your Python scripts. Instead, store your connection information in a separate configuration file, which has the following results:

- It can be excluded from version control (using `.gitignore`)
- It keeps sensitive information separate from your code
- It makes it easier to update connection details without modifying your scripts

For this chapter, it is recommended to create a file named `database.cfg` with sections for each database you'll connect to. Below is a template that you should customize with your own connection details:

```
# An example of configuration file "database.cfg file"
[SNOWFLAKE]
user=your_username
password=your_password
account=your_snowflake_account
warehouse=COMPUTE_WH
database=SNOWFLAKE_SAMPLE_DATA
schema=TPCH_SF1
role=your_role

[POSTGRESQL_PSYCOPG]
host: 127.0.0.1
dbname: postgres
user: postgres
password:your_password
port=5432

[AWS]
```

```
host=<your_end_point.your_region.redshift.amazonaws.com>
port=5439
database=dev
username=your_password
password=your_password
```

You can then load this configuration file in your Python scripts using the *configparser* module:

```
import configparser
config = configparser.ConfigParser()
config.read('database.cfg')

# Later, access credentials like this:
snowflake_user = config['SNOWFLAKE']['user']
postgres_password = config['POSTGRESQL']['password']
```

Reading data from a relational database

Relational databases such as PostgreSQL are widely used for structured time series data due to their ACID compliance, SQL support, and robust indexing capabilities. In this recipe, you'll extract time series data from PostgreSQL using two Python approaches:

1. **Low-level connection:** Use *psycopg2* (PostgreSQL's Python adapter) to execute SQL queries and load results into a pandas DataFrame.
2. **High-level ORM:** Leverage *SQLAlchemy*, an object-relational mapper (ORM) integrated with *pandas*, for more abstracted and maintainable queries.

Getting ready

In this recipe, we'll use PostgreSQL version 18.0, the latest stable version as of this writing. The code examples should be compatible with recent PostgreSQL releases.

To connect to and query the database in Python, you will need to install *psycopg*, a popular PostgreSQL database adapter for Python. You will also need to install *SQLAlchemy*, which provides flexibility regarding how you want to manage the database, whether for writing or reading data.

To install the libraries using *pip*, run the following command:

```
pip install sqlalchemy psycopg
```

This will install the latest version of **psycopg** (version 3). Note that although we're using **psycopg3** (the latest version), the package is named simply *psycopg* – unlike the previous version, which was installed as *psycopg2*.



If you ran into an issue when importing **psycopg**, run the following command:

```
pip install "psycopg[binary]"
```

If you cannot access a PostgreSQL database, the fastest way to get up and running is via Docker (https://hub.docker.com/_/postgres). The following is an example command:

```
docker run -d \  
  --name postgres-ch3 \  
  -p 5432:5432 \  
  -e POSTGRES_PASSWORD=password \  
  postgres:18.0
```

This command creates a container named `postgres-ch3` with the following connection details:

```
Host: localhost (127.0.0.1)  
Port: 5432  
Username: postgres (default)  
Password: password  
Database: postgres (default)
```

You can verify the container is running with: `docker ps`

Once the `postgres-ch3` container is up and running, you can connect to it using **DBeaver**, as shown:

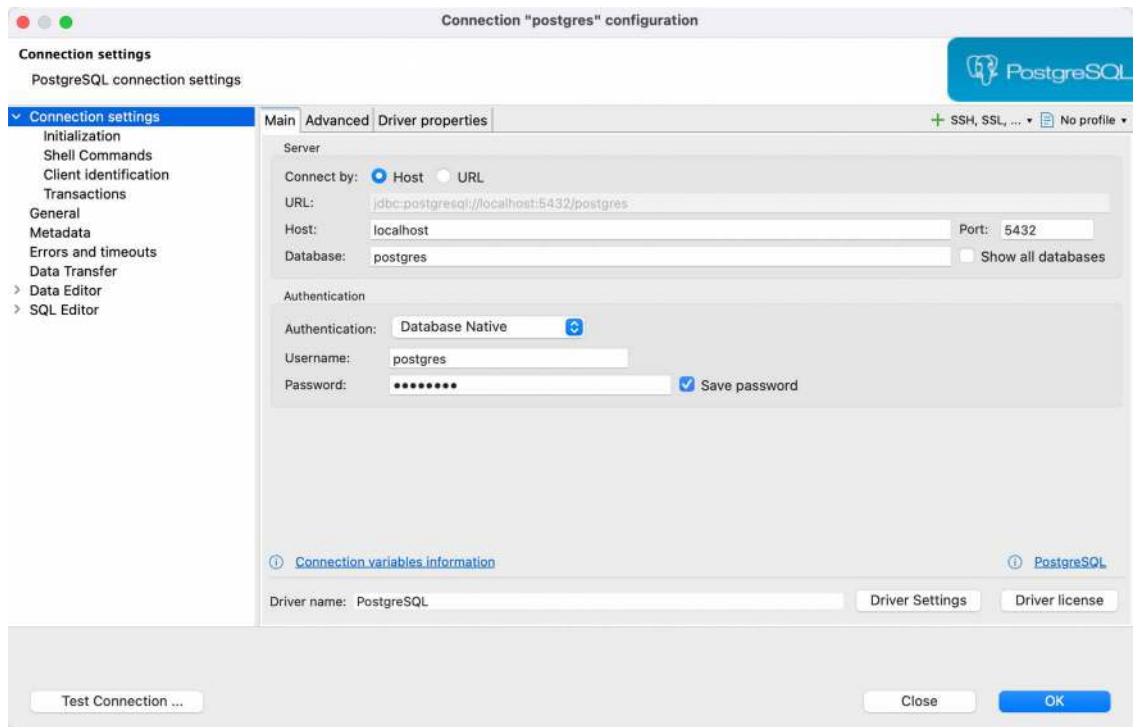


Figure 2.2: DBeaver PostgreSQL connection settings

You will be working with the MSFT stock dataset provided in the `datasets/Ch2/MSFT.csv` folder. I have uploaded the dataset into the database using **DBeaver Community Edition**, which you can download here: <https://dbeaver.io/download/>.

You can import the CSV file by right-clicking on Tables under the public schema, and selecting Import Data option.

You can confirm all records have been written to the `msft` table in the postgres database by running:

```
select count(*) from msft;
>>
1259
```

Update `database.cfg` to include a [POSTGRESQL] section with the correct database configurations. The following is an example based on the Docker command used earlier:

```
[POSTGRESQL]
host: 127.0.0.1
dbname: postgres
user: postgres
password:password
port=5432
```

How to do it...

We will start by connecting to the PostgreSQL instance, querying the database, loading the result set into memory, and parsing the data into a time series DataFrame.

In this recipe, I will connect to a PostgreSQL instance that runs locally, so my connection would be to localhost (127.0.0.1). You will need to adjust this for your own PostgreSQL database settings.

Using psycopg

psycopg is a Python library (and a database driver) that provides additional functionality and features when working with a PostgreSQL database. Follow these steps:

1. Start by importing the necessary libraries. You will import the required connection parameters from `database.cfg`, highlighted in the *Technical requirements* section. You will create a Python dictionary to store all the parameter values required to establish a connection to the database, such as host, database name, user name, and password:

```
import psycopg
import pandas as pd
import configparser

config = configparser.ConfigParser()
config.read('database.cfg')

params = dict(config['POSTGRESQL_PSYCOPG'])
```

2. You can establish a connection by passing the parameters to the `connect()` method. Once connected, you can create a **cursor** object that can be used to execute SQL queries:

```
conn = psycopg.connect(**params)
cursor = conn.cursor()
```

Here, `**params` unpacks the `params` dictionary as keyword arguments.

3. The cursor object provides several attributes and methods, including `execute`, `executemany`, `fetchall`, `fetchmany`, and `fetchone`. The following code uses the cursor object to pass a SQL query and then checks the number of records that have been produced by that query using the `rowcount` attribute:

```
cursor.execute("""
    SELECT date, close, volume
    FROM msft
    ORDER BY date;
""")

cursor.rowcount

>> 1259
```

4. The returned result set after executing the query will not include a header (no column names). Alternatively, you can grab the column names from the cursor object using the `description` attribute, as shown in the following code:

```
cursor.description

>>
[<Column 'date', type: varchar(50) (oid: 1043)>,
 <Column 'close', type: float4 (oid: 700)>,
 <Column 'volume', type: int4 (oid: 23)>]
```

This gives us the column names that can be used as `DataFrame` headers.

5. You can use a list comprehension to extract the column names from `cursor.description` to pass as column headers when creating the `DataFrame`:

```
columns = [col[0] for col in cursor.description]
columns

>>
['date', 'close', 'volume']
```


- To fetch the results that were produced by the executed query, you will use the `fetchall` method. You will create a `DataFrame` based on the fetched result set. Make sure that you pass the column names that you just captured:

```
data = cursor.fetchall()
df = pd.DataFrame(data, columns=columns)
df.info()

>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1259 entries, 0 to 1258
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  -
 0   date     1259 non-null   object
 1   close    1259 non-null   float64
 2   volume   1259 non-null   int64
dtypes: float64(1), int64(1), object(1)
memory usage: 29.6+ KB
```

Notice the date column is returned as an object type, not a datetime type.

- Parse the date column using `pd.to_datetime()` and set it as the index for the `DataFrame`:

```
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')
print(df.tail(3))

>>
           close    volume
date
2024-08-30  417.14  24308300
2024-09-03  409.44  20285900
2024-09-04  408.84   9167942
```

In the preceding code, the cursor returned a list of tuples **without a header**. You can confirm this by running the following code:

```
data = cursor.fetchall()
data[0:5]

>>
[('2019-09-04', 131.45726, 17995900),
 ('2019-09-05', 133.7687, 26101800),
 ('2019-09-06', 132.86136, 20824500),
 ('2019-09-09', 131.35222, 25773900),
 ('2019-09-10', 129.97684, 28903400)]
```

You can instruct the cursor to return a `dict_row` type, which will include the column name information (the header). This is more convenient when converting into a `DataFrame`. This can be done by passing the `dict_row` class to the `row_factory` parameter:

```
from psychopg.rows import dict_row
conn = psychopg.connect(**params, row_factory=dict_row)

cursor = conn.cursor()

cursor.execute("SELECT * FROM msft;")
data = cursor.fetchall() # Auto-includes headers
data[0:2]

>>
[{'date': '2019-09-04',
  'open': 131.14206,
  'high': 131.51457,
  'low': 130.35883,
  'close': 131.45726,
  'volume': 17995900},
 {'date': '2019-09-05',
  'open': 132.87086,
  'high': 134.08391,
  'low': 132.53656,
  'close': 133.7687,
  'volume': 26101800}]
```

Notice the column names available. You can now create a `DataFrame` as in the following code:

```
df = pd.DataFrame(data)
```

8. Close the cursor and the connection to the database:

```
cursor.close()
conn.close()
```

Note that `psychopg` connections and cursors can be used in Python's `with` statement for exception handling when committing a transaction. The cursor object provides three different fetching functions – that is, `fetchall`, `fetchmany`, and `fetchone`. The `fetchone` method returns a single tuple. The following example shows this concept:

```
with psychopg.connect(**params) as conn:
    with conn.cursor() as cursor:
        cursor.execute('SELECT * FROM msft')
        data = cursor.fetchone()
```

```
print(data)

>>
('2019-09-04', 131.14206, 131.51457, 130.35883, 131.45726, 17995900)
```

Using pandas and SQLAlchemy

SQLAlchemy is a very popular open source library for working with relational databases in Python. SQLAlchemy can be referred to as an **Object-Relational Mapper (ORM)**, which provides an abstraction layer (think of it as an interface) so that you can use object-oriented programming to interact with a relational database.

You will be using SQLAlchemy because it integrates very well with pandas, and several of the pandas SQL reader and writer functions depend on SQLAlchemy as the abstraction layer. SQLAlchemy does the behind-the-scenes translation for any pandas SQL read or write requests. This translation ensures that the SQL statement from pandas is represented in the correct syntax/format for the underlying database type (MySQL, Oracle, SQL Server, or PostgreSQL, to name a few).

Some of the pandas SQL reader functions that rely on SQLAlchemy include `pandas.read_sql`, `pandas.read_sql_query`, and `pandas.read_sql_table`. Let's perform the following steps:

1. To utilize the `database.cfg` file to store your database connection information, you will need to update it by adding a new entry for PostgreSQL to support SQLAlchemy. Unlike using `psycopg.connect()`, which expects `dbname` and `user` parameters, SQLAlchemy expects `database` and `username` parameters instead. Create a new entry, such as the following example:

```
[POSTGRESQL]
host=127.0.0.1
database=postgres
username=your_username
password=your_password
port=5432
```

2. Start by importing the necessary libraries. Note that, behind the scenes, SQLAlchemy will use `psycopg` (or any other database driver that is installed and supported by SQLAlchemy):

```
import pandas as pd
from sqlalchemy import create_engine, URL
```

3. You will use the `URL` class to generate the URL connection string:

```
params = dict(config['POSTGRESQL'])
url = URL.create('postgresql+psycopg', **params)
url
>>
postgresql+psycopg://postgres:***@127.0.0.1:5432/postgres
```

4. Create the engine object and pass it to the `con` parameter:

```
engine = create_engine(url)
query = "SELECT * FROM msft"
df = pd.read_sql(query,
                  con=engine,
                  index_col='date',
                  parse_dates=['date'])
```

Just as we used the `WITH` statement for context management with `psycopg`, you can apply the same pattern with SQLAlchemy connections when using pandas. This ensures connections are properly closed even if errors occur. When using the context manager pattern with SQLAlchemy and pandas, it's important to pass the connection object (not the engine) to the pandas function:

```
# Good practice for connection management
with engine.connect() as conn:
    df = pd.read_sql(query,
                     conn,
                     index_col='date',
                     parse_dates=['date'])
```

Unlike the previous `psycopg` approach, `pandas.read_sql` did a better job in getting the data types correct. Notice that our index is of the `DatetimeIndex` type:

```
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1259 entries, 2019-09-04 to 2024-09-04
Data columns (total 5 columns):
#   Column  Non-Null Count  Dtype
---  -
0    open    1259 non-null    float64
1    high    1259 non-null    float64
2    low     1259 non-null    float64
3    close   1259 non-null    float64
4    volume  1259 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 59.0 KB
```

5. You could also accomplish the same results using `pandas.read_sql_query`:

```
df = pd.read_sql_query(query,
                       engine,
                       index_col='date',
                       parse_dates=['date'])
```

6. The `pandas.read_sql_table` SQL reader function is another one provided by pandas that takes in a table name instead of a SQL query. Think of this as a `SELECT * FROM tablename` query:

```
df = pd.read_sql_table('msft',  
                       engine,  
                       index_col='date',  
                       parse_dates=['date'])
```



The `read_sql` reader function is more versatile as it is a wrapper to `read_sql_query` and `read_sql_table`. The `read_sql` function can take either a SQL query or a table name.

The **SQLAlchemy** with pandas approach offers several advantages:

- Requires less boilerplate code than direct `psycopg`
- Handles date parsing automatically through the `parse_dates` parameter, simplifying the conversion of string dates to datetime objects
- Provides built-in SQL injection protection through parameterized queries
- Offers database-agnostic code that can work with different database backends with minimal changes

How it works...

You explored two methods to connect to a PostgreSQL database in this recipe: using the `psycopg` driver directly or utilizing pandas and **SQLAlchemy**.

When working directly with **psycopg** to connect to PostgreSQL, you first need to create a connection object followed by a cursor object. This concept of a connection object and cursor is consistent across different database drivers in Python. Generally, you will follow a traditional database access pattern:

- **Connection:** Establish a connection to the database
- **Cursor:** Create a cursor for executing queries
- **Execution:** Run SQL statements through the cursor
- **Fetching:** Retrieve results using one of the several methods available
- **Cleanup:** Close resources when done

Once a cursor object is created, you have access to several methods, including the following:

- `execute()`: Executes a single SQL query (CRUD) or command to the database
- `executemany()`: Executes the same database operation with a sequence of input data – for example, this can be useful with `INSERT INTO` for bulk insert.
- `fetchall()`: Returns all remaining records from the current query result set. This is a good option when the result set is small and can fit in memory.
- `fetchone()`: Returns the next record (one record) from the current query result set. This is a good option when processing rows individually.

- `fetchmany(n)`: Returns `n` number of records from the current query result set. This is a good option when working with large datasets to control memory usage.
- `close()`: Closes the current cursor and frees associated resources.

On the other hand, creating an engine object is the first step when working with **SQLAlchemy**, as it provides instructions on the database that is being considered. This is known as a **dialect**.

When you created the engine with `create_engine`, you passed a URL as the connection string.

```
create_engine("dialect+driver://username:password@host:port/database")
```

- **dialect**: The name of the SQLAlchemy dialect (database type) such as `postgresql`, `mysql`, `sqlite`, `oracle`, or `mssql`.
- **driver**: The name of the installed driver (DBAPI) to connect to the specified dialect, such as `psycopg` or `pg8000` for a PostgreSQL database.
- **username**: The login username for database authentication.
- **password**: The password for the username specified.
- **host**: The server where the database is hosted.
- **port**: The specific port for the database.
- **database**: The name of the specific database you want to connect to.

Earlier, you used `psycopg` as the database driver for PostgreSQL. The `psycopg` driver is referred to as a **database API (DBAPI)**, and SQLAlchemy supports many DBAPI wrappers based on Python's DBAPI specifications to connect to and interface with various types of relational databases. SQLAlchemy already comes with built-in dialects to work with different flavors of RDBMS, such as the following:

- SQL Server
- SQLite
- PostgreSQL
- MySQL
- Oracle
- Snowflake



When connecting to a database using SQLAlchemy, you need to specify the **dialect** and the **driver** (DBAPI) to be used. For **PostgreSQL** with the latest **psycopg** driver (version 3), the connection string looks like this: `create_engine("postgresql+psycopg://username:password@localhost:5432/dbname")`

If you were using the older `psycopg2` driver, it would look like this:

```
create_engine("postgresql+psycopg2://username:password@localhost:5432/dbname")
```

This is what it would look like if you were using a **MySQL** database with the **PyMySQL** driver:

```
create_engine("mysql+pymysql://username:password@localhost:3306/dbname")
```

In the previous code examples in the *How to do it...* section, you did not need to specify the `psycopg` driver since it is the default DBAPI that SQLAlchemy uses. This example would work just fine, assuming that `psycopg` is installed:

```
create_engine("postgresql://username:password@localhost:5432/dbname")
```

There are other PostgreSQL drivers (DBAPI) that are supported by SQLAlchemy, including the following:

```
psycopg  
pg8000  
asyncpg
```

For a more comprehensive list of supported dialects and drivers, you can visit the official documentation page at <https://docs.sqlalchemy.org/en/20/dialects/index.html>.

The advantage of using SQLAlchemy is that it is well-integrated with pandas. If you read the official pandas documentation for `read_sql`, `read_sql_query`, `read_sql_table`, and `to_sql`, you will notice that the `con` argument is expecting a SQLAlchemy connection object (engine).

Another advantage is that you can easily change the backend engine (database) – for example, from PostgreSQL to MySQL – without needing to change the rest of the code.

In the SQLAlchemy section, you used the `parse_dates` parameter. The `parse_dates` parameter can be used in two ways: you can either pass a list of column names to be parsed as dates or a dictionary where keys are column names and values are format strings. The dictionary approach is particularly useful when working with non-standard date formats or integer-based timestamps.

```
df = pd.read_sql(query,  
                 engine,  
                 index_col='date',  
                 parse_dates={'date': '%Y-%m-%d'})  
  
print(df.tail(3))
```

In the preceding example, you passed a dictionary in the format of {key: value}, where key is the column name and value is a string representation of the date format.

There's more...

In this section, we will explore some additional concepts to help you better grasp the versatility of SQLAlchemy.

Specifically, we will discuss the following:

- Generating the connection URL in SQLAlchemy
- Extending to the Amazon Redshift database
- Chunking in pandas for handling large datasets

Generating the connection URL in SQLAlchemy

In this recipe, you were introduced to the `create_engine` function from the SQLAlchemy library, which takes a URL string to establish a connection to a database. So far, you have been creating the URL string manually, but there is a more convenient way to generate the URL for you. This can be accomplished with the `create` method from the `URL` class in SQLAlchemy.

The following code demonstrates this:

```
from sqlalchemy import URL, create_engine

url = URL.create(
    drivename='postgresql+psycopg',
    host= '127.0.0.1',
    username='postgres',
    password='password', # In production, use env vars or secret manager
    database='postgres',
    port= 5432
)
>>
postgresql+psycopg://postgres:***@127.0.0.1:5432/postgres
```

Notice that `drivename` consists of the *dialect* and the *driver* in the format `dialect+driver`.

You can now pass the `url` to `create_engine` as you have done before.

```
engine = create_engine(url)
df = pd.read_sql('select * from msft;', engine)
```

Extending to the Amazon Redshift database

We discussed the versatility of SQLAlchemy, which allows you to change the engine (database back-end) and keep the remaining code as is – for example, using PostgreSQL or MySQL, or any other dialect supported by SQLAlchemy. We will also explore connecting to a cloud data warehouse such as Amazon Redshift.

It is worth mentioning that **Amazon Redshift**, a cloud data warehouse, is based on PostgreSQL at its core. You will install the Amazon Redshift driver for SQLAlchemy (it uses the `psycopg2` DBAPI, which is different from `psycopg3`, which we used earlier).

You can also install it using `pip`:

```
pip install sqlalchemy-redshift psycopg2-binary
```



Installing `sqlalchemy-redshift` will likely downgrade your version of the installed SQLAlchemy from 2.0.40 (or a newer version) to an older version (e.g., 1.4.x or earlier). To upgrade to the latest SQLAlchemy version, you can run the following command:

```
pip install sqlalchemy --force
```


Because we do not want to expose AWS credentials in the code, you will update the `database.cfg` file discussed in the *Technical requirements* section to include your AWS Redshift information:

```
[AWS]
host=your_end_point.your_region.redshift.amazonaws.com
port=5439
database=dev
username=your_username
password=your_password
```

You will use `configparser` to load your values:

```
from configparser import ConfigParser
config = ConfigParser()
config.read('database.cfg')
params = dict(config['AWS'])
```

You will use the `URL.create` method to generate your URL:

```
url = URL.create('redshift+psycopg2',
                 **params) # Unpacks host, port, etc.
aws_engine = create_engine(url)
```

Now, you can switch the engine from our previous code, which originally pointed to our local instance of PostgreSQL, to run the same query on Amazon Redshift. This assumes you have a `msft` table in Amazon Redshift.

```
df = pd.read_sql(query,
                 aws_engine,
                 index_col='date',
                 parse_dates=['date'])
```

To learn more about `sqlalchemy-redshift`, you can refer to the project's repository here: <https://github.com/sqlalchemy-redshift/sqlalchemy-redshift>.



The Amazon Redshift example can be extended to other databases, such as Google BigQuery, Teradata, or Microsoft SQL Server, as long as there is a supported SQLAlchemy dialect for that database. For a complete list, visit the official page here: <https://docs.sqlalchemy.org/en/20/dialects/index.html>.

Chunking with pandas

When you executed the query against the `msft` table, it returned 1,259 records. Now, imagine working with a much larger database that returns millions of records, if not more. This is where the chunking parameter helps in memory management.

The `chunksize` parameter allows you to break down a large dataset into smaller and more manageable chunks of data that can fit into your local memory. When executing the `read_sql` function, just pass the number of rows to be retrieved (per chunk) to the `chunksize` parameter, which then returns a generator object. You can then loop through the generator object, or use `next()` to capture one chunk at a time, and perform whatever calculations or processing is needed. Let's look at an example of how chunking works. You will request 500 records (rows) at a time:

```
df_gen = pd.read_sql(query,
                     engine,
                     index_col='date',
                     parse_dates=True,
                     chunksize=500)
```

The preceding code will generate three (3) chunks. You can iterate through the `df_gen` generator object as shown:

```
for idx, chunk in enumerate(df_gen):
    print(f"Chunk {idx}: Shape {chunk.shape}")

    # Process each chunk individually
    # For example, calculate statistics on each chunk
    print(f"Mean close price for chunk {idx}: {chunk['close'].mean():.2f}")

>>
Chunk 0: Shape (500, 5)
Mean close price for chunk 0: 198.07
Chunk 1: Shape (500, 5)
Mean close price for chunk 1: 280.95
Chunk 2: Shape (259, 5)
Mean close price for chunk 2: 391.79
```

Using the `chunksize` parameter helps reduce memory usage since the code loads a smaller number of rows per chunk at a time.

This approach dramatically reduces memory usage since only a portion of the data is loaded at a time. It's particularly valuable when working with time series data that may span years or decades and contain millions of rows.

See also

For additional information regarding these topics, take a look at the following links:

- For **SQLAlchemy**, you can visit <https://www.sqlalchemy.org/>
- For the `pandas.read_sql` function, visit https://pandas.pydata.org/docs/reference/api/pandas.read_sql_table.html

- For the `pandas.read_sql_query` function, visit https://pandas.pydata.org/docs/reference/api/pandas.read_sql_query.html
- For the `pandas.read_sql_table` function, visit https://pandas.pydata.org/docs/reference/api/pandas.read_sql_table.html

Reading data from Snowflake

A very common place to extract data for analytics is a company's *data warehouse*. Data warehouses host a massive amount of data that, in most cases, contains integrated data to support various reporting and analytics needs, in addition to historical data from various source systems.

The evolution of the cloud brought us modern cloud data warehouses such as Amazon Redshift, Google BigQuery, Azure SQL Data Warehouse, and Snowflake. Unlike traditional on-premise data warehouses, modern cloud data warehouses offer elastic scalability, separating storage and compute, with built-in features for handling semi-structured data.

In this recipe, you will work with **Snowflake**, a powerful **Software as a Service (SaaS)** cloud-based data warehousing platform that can be hosted on different cloud platforms, such as **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, and Microsoft Azure. You will learn how to connect to Snowflake using Python to extract time series data and load it into a pandas `DataFrame`.

Getting ready

This recipe assumes you have access to Snowflake. If you do not have access to Snowflake, you can sign up for a 30-day free trial here: <https://signup.snowflake.com/>. You will explore three (3) different Python approaches to extracting time series data from Snowflake:

- Snowflake Connector: Using the native Snowflake Connector for Python with pandas support for direct low-level access
- SQLAlchemy: ORM-based integration with the Snowflake dialect
- Snowpark: The `DataFrame` API for advanced analytics

The recommended approach for the `snowflake-connector-python` library is to install it using **pip**, allowing you to install *extras* such as `pandas`, as shown:

```
pip install snowflake-sqlalchemy # SQLAlchemy integration
pip install snowflake-snowpark-python # Snowpark DataFrame API
pip install "snowflake-connector-python[pandas]" #includes pandas support
```



Note that Python 3.13 is not yet supported, and you should check Snowflake's documentation for the most up-to-date compatibility information. When installing Snowpark, it's recommended to create a Python virtual environment: <https://docs.snowflake.com/en/developer-guide/snowpark/python/setup>.

Ensure the `database.cfg` configuration file that you created in the *Technical requirements* section contains your **Snowflake** connection information:

```
[SNOWFLAKE]
user=your_username
password=your_password
account=your_snowflakeaccount
warehouse=your_warehouse
database=SNOWFLAKE_SAMPLE_DATA
schema=TPCH_SF1
role=your_role
```

In this recipe, you will be working with the `SNOWFLAKE_SAMPLE_DATA` database and the `TPCH_SF1` schema, which are provided by Snowflake as sample datasets available to all users.

To find your account details in Snowsight (Snowflake web interface), click the menu in the bottom-left corner of the Snowsight sidebar and select **Account**, then choose **View account details** (see *Figure 2.3*).

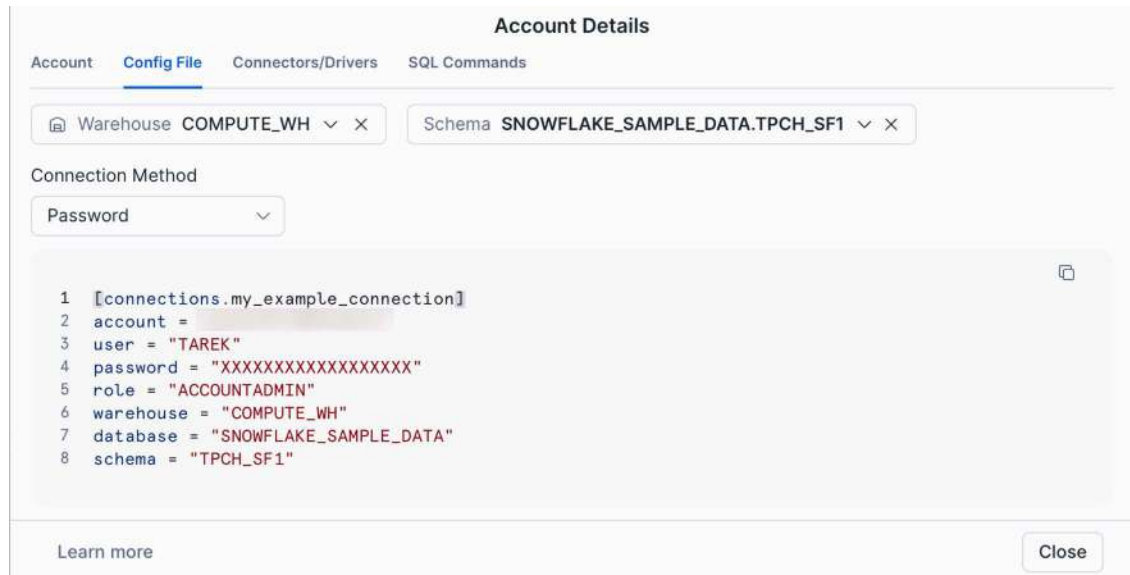


Figure 2.3: The Config File tab from the Account Details screen in Snowsight

How to do it...

We will explore three different Python approaches and libraries to connect to Snowflake. In the first method, you will use the Snowflake Python connector to establish a connection and create a cursor to query and fetch the data. In the second method, you will use Snowflake **SQLAlchemy**. In the third method, you will explore the **Snowpark** Python API. Let's get started:

Using Snowflake Python Connector

1. We will start by importing the necessary libraries:

```
import pandas as pd
from snowflake import connector
from configparser import ConfigParser
```

2. Using ConfigParser, you will extract the content under the [SNOWFLAKE] section to avoid exposing or hardcoding your credentials. You can read the entire content of the [SNOWFLAKE] section and convert it into a dictionary object, as shown here:

```
config = ConfigParser()
config.read('database.cfg')
params = dict(config['SNOWFLAKE'])
```

3. You will need to pass the parameters to `connector.connect()` to establish a connection with Snowflake. We can easily *unpack* the dictionary's content since the dictionary keys match the parameter names. Once the connection has been established, we can create our *cursor*:

```
con = connector.connect(**params)
cursor = con.cursor()
```

4. The cursor object has many methods, such as `execute`, `fetchall`, `fetchmany`, `fetchone`, `fetch_pandas_all`, and `fetch_pandas_batches`.

You will start with the `execute` method to pass a SQL query to the database, then use any of the available `fetch` methods to retrieve the data. In the following example, you will query the `ORDERS` table and then leverage the `fetch_pandas_all` method to retrieve the entire result set as a pandas DataFrame:

```
query = "SELECT * FROM ORDERS;"
cursor.execute(query)
df = cursor.fetch_pandas_all()
```

The previous code could have been written as follows:

```
df = cursor.execute(query).fetch_pandas_all()
```

5. Inspect the DataFrame using `df.info()`:

```
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1500000 entries, 0 to 1499999
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
#   Column                Non-Null Count  Dtype
```

```

0   O_ORDERKEY      1500000 non-null  int32
1   O_CUSTKEY       1500000 non-null  int32
2   O_ORDERSTATUS   1500000 non-null  object
3   O_TOTALPRICE    1500000 non-null  float64
4   O_ORDERDATE     1500000 non-null  object
5   O_ORDERPRIORITY 1500000 non-null  object
6   O_CLERK         1500000 non-null  object
7   O_SHIPPRIORITY  1500000 non-null  int8
8   O_COMMENT       1500000 non-null  object
dtypes: float64(1), int32(2), int8(1), object(5)
memory usage: 81.5+ MB

```

- From the preceding output, you can see that the DataFrame's index is just a sequence of numbers and that the O_ORDERDATE column is not a Date field. You can parse the O_ORDERDATE column to a datetime type by using the `pandas.to_datetime()` function and then setting the column as the DataFrame's index with the `DataFrame.set_index()` method:

```

df_ts = (df.set_index(pd.to_datetime(df['O_ORDERDATE'])))
        .drop(columns='O_ORDERDATE')

```

Let's display the first four columns and first five rows of the `df_ts` DataFrame:

```

print(df_ts.iloc[0:3, 1:5])
>>
           O_CUSTKEY O_ORDERSTATUS  O_TOTALPRICE  O_ORDERPRIORITY
O_ORDERDATE
1994-02-21         13726             F         99406.41           3-MEDIUM
1997-04-14         129376             0         256838.41  4-NOT SPECIFIED
1997-11-24         141613             0         150849.49  4-NOT SPECIFIED

```

- Inspect the index of the DataFrame. Print the first two indexes:

```

df_ts.index[0:2]
>>
DatetimeIndex(['1994-02-21', '1997-04-14'], dtype='datetime64[ns]',
              name='O_ORDERDATE', freq=None)

```

- Finally, you can close the cursor and the current connection:

```

cursor.close()
con.close()

```

You now have a time series DataFrame with `DatetimeIndex`.

- Alternatively, the code could be further simplified by using a **context manager** to automatically allocate and release resources:

```
# Connection with automatic cleanup
with connector.connect(**params) as con:
    with con.cursor() as cursor:
        # Query execution
        query = "SELECT * FROM ORDERS;"
        df = cursor.execute(query).fetch_pandas_all()
    # Convert to proper time series
df_ts = (df.set_index(pd.to_datetime(df['O_ORDERDATE'])))
        .drop(columns='O_ORDERDATE'))
```

Using SQLAlchemy

In the previous recipe, *Reading data from a relational database*, you explored the pandas `read_sql`, `read_sql_query`, and `read_sql_table` functions. This was accomplished by utilizing SQLAlchemy and installing one of the supported dialects. Here, we will use the Snowflake dialect after installing the `snowflake-sqlalchemy` driver.



When using pandas 2.2 or later with SQLAlchemy and Snowflake, compatibility breaks completely if you use SQLAlchemy 1.4. Make sure to upgrade to SQLAlchemy 2.0+ to avoid errors and ensure smooth integration.

You can upgrade SQLAlchemy using `pip install -U sqlalchemy`.

SQLAlchemy is better integrated with pandas, as you will experience in this section.

- Start by importing the necessary libraries and reading the Snowflake connection parameters from the `[SNOWFLAKE]` section in the `database.cfg` file.

```
from sqlalchemy import create_engine
from snowflake.sqlalchemy import URL
import configparser

config = ConfigParser()
config.read('database.cfg')
params = dict(config['SNOWFLAKE'])
```

- You will use the `URL` class to generate the URL connection string. We will create our engine object and then open a connection with `engine.connect()`:

```
url = URL(**params)
engine = create_engine(url)
con = engine.connect()
```

- Now, you can use either the `read_sql` or `read_sql_query` to execute our query against the `ORDERS` table in the `SNOWFLAKE_SAMPLE_DATA` database:

```
query = "SELECT * FROM ORDERS;"
df = pd.read_sql(query,
                 con,
                 index_col='o_orderdate',
                 parse_dates=['o_orderdate'])

df.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1500000 entries, 1992-04-22 to 1994-03-19
Data columns (total 8 columns):
#   Column                Non-Null Count  Dtype
---  -
0   o_orderkey             1500000 non-null  int64
1   o_custkey              1500000 non-null  int64
2   o_orderstatus          1500000 non-null  object
3   o_totalprice           1500000 non-null  float64
4   o_orderpriority        1500000 non-null  object
5   o_clerk                1500000 non-null  object
6   o_shippriority         1500000 non-null  int64
7   o_comment              1500000 non-null  object
dtypes: float64(1), int64(3), object(4)
memory usage: 103.0+ MB
```

Notice how we were able to parse the `o_orderdate` column and set it as an index all in one step, when compared to the Snowflake Python connector method performed earlier.

- Finally, close the connection to the database:

```
con.close()
engine.dispose()
```

- The code could be further simplified by using a **context manager** to automatically allocate and release resources. The following example uses `with engine.connect():`:

```
query = "SELECT * FROM ORDERS;"
url = URL(**params)

engine = create_engine(url)
with engine.connect() as con:
    df = pd.read_sql(query,
                     con,
                     index_col='o_orderdate',
                     parse_dates=['o_orderdate'])
```


This should produce the same results without needing to close the connection and dispose of the engine.

Using the Snowpark API

The Snowpark API supports Java, Python, and Scala. You have already installed `snowflake-snowpark-python`, as described in the *Getting ready* section of this recipe.

1. Start by importing the necessary libraries and reading the Snowflake connection parameters from the `[SNOWFLAKE]` section in the `database.cfg` file:

```
from snowflake.snowpark import Session
from configparser import ConfigParser

config = ConfigParser()
config.read('database.cfg')
params = dict(config['SNOWFLAKE'])
```

2. Create a session by establishing a connection with the Snowflake database:

```
session = Session.builder.configs(params).create()
```

3. The session has several `DataFrameReader` methods, such as `read`, `table`, and `sql`. Any of these methods would return a Snowpark `DataFrame` object. The returned Snowpark `DataFrame` object has access to the `to_pandas` method to convert into a more familiar pandas `DataFrame`. You will explore the `read`, `table`, and `sql` methods to return the same result set.
4. Start with the `read` method. More specifically, you will be using `read.table` and passing it a table name. This will return the content of the table and convert it into a pandas `DataFrame` with the `to_pandas` method. Think of this as equivalent to `SELECT * FROM TABLE`.

```
orders = session.read.table("ORDERS").to_pandas()
```

Similarly, the `table` method takes a table name and the returned object (Snowpark `DataFrame`) as access to the `to_pandas` method:

```
orders = session.table("ORDERS").to_pandas()
```

Lastly, the `sql` method takes a SQL query:

```
query = 'SELECT * FROM ORDERS'
orders = session.sql(query).to_pandas()
orders.info()
```

All three approaches should produce the same pandas `DataFrame`.

How it works...

The Snowflake Python connector, Snowflake SQLAlchemy driver, and Snowpark Python require the same input variables to establish a connection to the Snowflake database. Recall that in the previous activity, you used the same configuration for all three methods in the `database.cfg` file under the `[SNOWFLAKE]` section.

These include the following:

Parameter	Required	Description
account	Yes	The account identifier, including region and cloud platform
user	Yes	The username for Snowflake authentication.
password	Yes	The password associated with the user account.
database	Optional	The name of the database to connect to. Defaults to the user's default database if not specified.
schema	Optional	The name of the schema within the database. Defaults to the user's default schema if not specified.
role	Optional	The role to assume for the session. Defaults to the user's default role if not specified.
warehouse	Optional	The name of the virtual warehouse to use for query execution.

Table 2.1: Input variables for the Snowflake Python connector

Table 2.2 provides a comparison of the three Snowflake connection methods you explored, highlighting their key features and appropriate use cases. Each approach offers distinct advantages depending on your specific requirements, existing codebase, and the complexity of your data processing needs.

Feature	Connector	SQLAlchemy	Snowpark
Ease of Use	Medium (manual query handling)	High (integrates seamlessly with pandas)	Medium-High (API requires learning curve)
pandas Integration	Direct (fetch_pandas_all, write_pandas)	Excellent (read_sql, to_sql)	Good (to_pandas method, excellent with Snowpark pandas API)
Performance	Fastest (low-level direct connection)	Fast (optimized ORM queries)	Medium (depends on server-side execution and lazy evaluation)
Advanced Features	Basic CRUD	ORM, DDL Support	Lazy evaluation, UDFs, distributed processing
Best For	Simple Queries	Existing SQLAlchemy-based code bases	Complex transformations, large-scale ETL/ML pipelines
Scalability	Limited (client-side operations)	Medium	High (server-side execution and distributed processing)

Table 2.2: Comparing Python connection methods for Snowflake

Snowflake Python Connector

When using the Python connector, you first establish a connection to the database with `con = connector.connect(**params)`. Once the connection is accepted, you create a cursor object with `cursor = con.cursor()`.

The cursor provides methods for performing execute and fetch operations such as `describe()`, `execute()`, `execute_async()`, `executemany()`, `fetchone()`, `fetchall()`, `fetchmany()`, `fetch_pandas_all()`, and `fetch_pandas_batches()`, and each cursor has several attributes, including `description`, `rowcount`, and `rownumber`, to name a few. Note that there are familiar methods and attributes discussed in the previous recipe, *Reading data from a relational database*, when using the Python connector `psycopg`.

- `execute()`: Executes a single SQL query (CRUD) or command to the database
- `executemany()`: Executes the same database operation with a sequence of input data – for example, this can be useful with `INSERT INTO` for bulk insert
- `fetchall()`: Returns all remaining records from the current query result set
- `fetchone()`: Returns the next record (one record) from the current query result set
- `fetchmany(n)`: Returns `n` number of records from the current query result set
- `fetch_pandas_all()`: Returns all remaining records from the current query result set and loads them into a pandas `DataFrame`
- `fetch_pandas_batches()`: Returns a subset of the remaining records from the current query result set and loads them into a pandas `DataFrame`
- `close()`: Closes the current cursor and frees associated resources
- `describe()`: Returns metadata about the result set without executing the query. Alternatively, you can use `execute()` followed by the `description` attribute to obtain the same metadata information.

For a complete list of attributes and methods, you can refer to the official documentation at <https://docs.snowflake.com/en/user-guide/python-connector-api.html#object-cursor>.

SQLAlchemy API

With SQLAlchemy, you are able to leverage the `pandas.read_sql`, `pandas.read_sql_query`, and `pandas.read_sql_query` reader functions and utilize many of the available parameters to transform and process the data at read time, such as `index_col` and `parse_dates`. On the other hand, when using the Snowflake Python connector, the `fetch_pandas_all()` function does not take in any parameters, and you will need to parse and adjust the `DataFrame` afterward.

The Snowflake SQLAlchemy library provides a convenience method, `URL`, to help construct the connection string to connect to the Snowflake database. Typically, a SQLAlchemy connection string for Snowflake would look like this:

```
# URL format example
'snowflake://<user>:<password>@<account>/<database>/<schema>
?warehouse=<warehouse>&role=<role>'
```

Using the URL class, you can construct this connection string in a more programmatic way:

```
engine = create_engine(URL(
    account = '<your_account>',
    user = '<your_username>',
    password = '<your_password>',
    database = '<your_database>',
    schema = '<your_schema>',
    warehouse = '<your_warehouse>',
    role='<your_role>',
))
```

Alternatively, we can store our Snowflake parameters in the `database.cfg` configuration file and load them as a Python dictionary. This way, you avoid exposing your credentials within the code.

```
params = dict(config['SNOWFLAKE'])
url = URL(**params)
engine = create_engine(url)
```



If you compare the process in this recipe, using **SQLAlchemy**, for Snowflake, and that of the previous recipe, *Reading data from a relational database*, you will observe similarities in the process and code. This is one of the advantages of using SQLAlchemy: it creates a standard process across a variety of databases as long as SQLAlchemy supports them. *SQLAlchemy is well integrated with pandas and makes it easy to switch dialects (backend databases) without many changes to your code.*

The Snowpark API

The previous methods (connector and SQLAlchemy) primarily serve as connectors to your Snowflake database, where you fetch the data to process locally in your Python environment. This approach works well for small to medium datasets, but becomes inefficient for large-scale time series analysis.

Snowpark offers more than just a mechanism to connect to your database. It allows you to process data directly within the Snowflake environment in the cloud without the need to transfer potentially massive datasets to your local environment. Snowpark is well suited for more complex tasks, such as building complex data pipelines, or there's Snowpark ML for working with machine learning models, all in the Snowflake cloud.

In our recipe, similar to the other methods, we first establish our connection with Snowflake using the session class:

```
params = dict(config['SNOWFLAKE'])
session = Session.builder.configs(params).create()
```

There are intentional similarities in the API and concepts of **Snowpark** and **PySpark** (Apache Spark). Both use **DataFrames** abstraction with similar transformation methods (**select**, **filter**, **join**), both implement **lazy evaluation** where operations are only executed when an action is triggered, and both optimize query execution plans behind the scenes. This makes Snowpark more familiar to Spark users simplifying migration between the two platforms. Despite these similarities, there are fundamental architectural differences. Snowpark leverages Snowflake's managed infrastructure and query optimization capabilities, while PySpark relies on distributed Spark clusters.

In the recipe, you used the `to_pandas` method, which does two things: it triggers execution of all pending operations (**executes the query**) and **fetches the results** into a local pandas DataFrame (outside of Snowflake). To move the data in the opposite direction, from a local pandas DataFrame into Snowflake, you can use the `create_dataframe` method as shown:

```
snowpark_df = session.create_dataframe(orders)
```

To execute the preceding code successfully, you need to have *write permission* since Snowflake creates a **temporary** table in your current schema to store the pandas DataFrame (inside Snowflake) and then returns a Snowpark DataFrame that points to that temporary table.

If you want to persist the pandas DataFrame into a named table, you can use the `write_pandas` method as shown:

```
snowpark_df = session.write_pandas(orders, table_name='temp_table')
```

In the preceding code, you passed the pandas DataFrame and a table name. The method writes the data to Snowflake and returns a Snowpark DataFrame object that references the newly created or updated table.



The `write_pandas` method requires `CREATE TABLE`, `CREATE STAGE`, and `CREATE FILE FORMAT` privileges in your target schema. If you encounter permission errors, either request these privileges or switch to a schema where you have write access using `session.sql("USE SCHEMA database.schema").collect()` before calling `write_pandas`.

This returned DataFrame allows you to immediately work with the data in Snowflake without creating a new connection or query.



Unless you set `auto_create_table=True`, you must first create a table in Snowflake that matches the structure of your pandas DataFrame. If the DataFrame schema doesn't match the existing table, an exception will be raised. For quick tests or prototyping, using `auto_create_table=True` can be convenient, but for production workflows, you'll typically want to define your tables explicitly with appropriate data types and constraints.

For time series analysis, this bi-directional movement capability is particularly useful. You can use Snowflake to perform heavy data preparation and aggregation on large datasets, bring only the relevant summary data to pandas for specialized time series modeling, and then push results back to Snowflake for enterprise-wide access.

There's more...

You may have noticed that the columns in the returned DataFrame, when using the Snowflake Python connector and Snowpark, all came back in uppercase, while they were lowercase when using Snowflake SQLAlchemy.

The reason for this is that Snowflake, by default, stores unquoted object names in uppercase when these objects are created. In the previous code, for example, our `Order Date` column was returned as `O_ORDERDATE`.

To explicitly indicate the name is case-sensitive, you will need to use quotes when creating the object in Snowflake (for example, `'o_orderdate'` or `'OrderDate'`). In contrast, using Snowflake SQLAlchemy converts the names into lowercase by default.

See also

- For more information on the **Snowflake Connector for Python**, you can visit the official documentation at <https://docs.snowflake.com/en/user-guide/python-connector.html>
- For more information regarding **Snowflake SQLAlchemy**, you can visit the official documentation at <https://docs.snowflake.com/en/user-guide/sqlalchemy.html>
- For more information regarding **Snowpark API**, you can visit the official documentation at <https://docs.snowflake.com/developer-guide/snowpark/reference/python/latest/snowpark/index>

Reading data from MongoDB

MongoDB, a NoSQL database, stores data in documents and uses Binary JSON (BSON), which extends JSON with additional data types and is more efficient for storage and retrieval. Unlike relational databases, where data is stored in tables that consist of rows and columns, document-oriented databases store data in collections and documents.

A document represents the lowest granular level of data being stored, as rows do in relational databases. A collection, like a table in relational databases, stores documents. Unlike relational databases, a collection can store documents of different schemas and structures, making MongoDB's flexible schema ideal for time series data that might evolve with changing metrics or collection patterns.

Since version 5.0, MongoDB offers dedicated **time series collections** that efficiently store and organize time series data. These specialized collections automatically organize measurements by time and metadata (such as sensor ID or data source), providing optimized storage, improved query performance, and reduced disk usage compared to standard collections. For time series analysis, MongoDB stores key components: a timestamp, metadata that identifies the data source, and the actual measurements tracked over time.

Getting ready

In this recipe, it is assumed that you have a running instance of MongoDB. To get ready for this recipe, you will need to install the PyMongo Python library to connect to MongoDB.

To install MongoDB using pip, run the following command:

```
python -m pip install pymongo
```

If you do not have access to a MongoDB instance, the fastest way to get up and running is via Docker (https://hub.docker.com/_/mongo). Use the following command to start a MongoDB container:

```
docker run -d \
--name mongo-ch3 \
-p 27017:27017 \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=your_password \
mongo:8.0.6
```

Alternatively, you can try MongoDB Atlas for free here: <https://www.mongodb.com/products/platform/atlas-database>. MongoDB Atlas is a fully managed cloud database that can be deployed on your favorite cloud providers, such as AWS, Azure, and GCP. To learn how to deploy a free cluster, refer to the instructions here: <https://www.mongodb.com/docs/atlas/tutorial/deploy-free-tier-cluster/>.

Note about using MongoDB Atlas



If you are connecting to MongoDB Atlas (Cloud) Free Tier or their M2/M5 shared tier cluster, then you will be using the `mongodb+srv` protocol. In this case, install PyMongo with SRV support: `python -m pip install "pymongo[srv]"`. You will need to whitelist your IP address in Atlas in order to establish a connection.

Install **MongoDB Compass** for a GUI interface to your MongoDB instance. You can download Compass here: <https://www.mongodb.com/products/tools/compass>

I am using MongoDB Compass to create the database, collection, and load the data. In *Chapter 4*, you will learn how to create your database and collection, and load data using Python.

Using **Compass**, click the **Add new connection** button to connect to your MongoDB instance. I am connecting to my local instance (Docker container). Under **Advanced Connection Options** select, then fill in the Username and Password, then click **Save & Connect**.

Once connected, click the **Create Database** button in the top-left corner. For **Database Name**, enter `stock_data`, and for **Collection Name**, enter `microsoft`. Check the **Time-Series** option and specify date as the `timeField`.

Once the database and collection are created, click **ADD DATA**, select the **Import JSON or CSV file** option, and navigate to the MSFT stock CSV dataset provided in the `datasets/Ch2/MSFT.csv` folder.

On the final page, verify the data types and then click **Import**.

How to do it...

In this recipe, you will connect to the MongoDB instance you have set up. For an on-premises installation (local install or Docker container), your connection string will be something such as `mongodb://<username>:<password>@<host>:<port>/<DatabaseName>`. If you are using MongoDB Atlas, your connection string will be similar to `mongodb+srv://<db_username>:<db_password>@cluster0.7d1ej.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0`, which you can obtain by clicking the connect option on the Clusters page.

Perform the following steps:

1. Start by importing the necessary libraries:

```
import pandas as pd
from pymongo import MongoClient, uri_parser
```

2. Establish a connection to MongoDB. For a self-hosted instance, such as a local install, this would look something like this:

```
# connecting to a self-hosted instance
url = "mongodb://127.0.0.1:27017"
client = MongoClient(url)
>>
MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False, connect=True)
```

This is equivalent to the following:

```
client = MongoClient(host=['127.0.0.1:27017'],
                    password=None,
                    username=None,
                    document_class=dict,
                    tz_aware=False,
                    connect=True)
```

If your self-hosted MongoDB instance requires a **username** and **password**, you must supply those. In the *Getting ready* section, in the command for running the Docker container, we configured the MongoDB instance with a **username** (`MONGO_INITDB_ROOT_USERNAME=admin`) and a **password** (`MONGO_INITDB_ROOT_PASSWORD=your_password`):

```
client = MongoClient(host=['127.0.0.1:27017'],
                    password='your_password',
                    username='admin',
                    document_class=dict,
                    tz_aware=False,
                    connect=True)
```


Alternatively, you can separate the host and port as shown:

```
client = MongoClient(host='127.0.0.1',
                    port=27017,
                    password='your_password',
                    username='admin',
                    document_class=dict,
                    tz_aware=False,
                    connect=True)
```

The `uri_parser` function is a useful utility function that allows you to validate a MongoDB URL, as shown:

```
uri = "mongodb://admin:your_password@127.0.0.1:27017"
uri_parser.parse_uri(uri)
>>
{'odelist': [('127.0.0.1', 27017)],
 'username': 'admin',
 'password': 'your_password',
 'database': None,
 'collection': None,
 'options': {},
 'fqdn': None}
```

If you are connecting to **MongoDB Atlas**, your connection string will look like this:

```
# connecting to Atlas cloud Cluster
cluster = 'cluster0'
username = 'user'
password = 'your_password'
database = 'stock_data'
url = \
f"mongodb+srv://{username}:{password}@{cluster}.3rncb.mongodb.net/
{database}"
client = MongoClient(url)
client
>>
MongoClient(host=['cluster0-shard-00-00.3rncb.mongodb.net:27017',
 'cluster0-shard-00-01.3rncb.mongodb.net:27017', 'cluster0-shard-00-
02.3rncb.mongodb.net:27017'], document_class=dict, tz_aware=False,
connect=True, authsource='somesource', replicaset='Cluster0-shard-0',
ssl=True)
```

Never hardcode credentials in your scripts. Always use environment variables or configuration files, as demonstrated in previous recipes. For example, you can use a `database.cfg` file to store your connection information and hide your credentials.

If your username or password contains special characters, including the space character (`:` `/` `?` `#` `[` `]` `@` `!` `$` `&` `'` `(` `)` `*` `,` `;` `=` `%`), then you will need to encode them. You can perform percent-encoding (percent-escape) using the `quote_plus()` function from the `urllib` Python library.

Here is an example:



```
from urllib.parse import quote_plus
username = quote_plus('user!*@')
password = quote_plus('pass/w@rd')
```

For more information, you can read the documentation here: <https://www.mongodb.com/docs/atlas/troubleshoot-connection/#std-label-special-pass-characters>

3. Once connected, list all available databases. In this example, the database is named `stock_data` and the collection is `microsoft`:

```
client.list_database_names()
>>
['admin', 'config', 'local', 'stock_data']
```

4. List the collections available in the `stock_data` database using `list_collection_names`:

```
db = client['stock_data']
db.list_collection_names()
>>
['microsoft', 'system.buckets.microsoft', 'system.views']
```

The output shows three collections:

- `Microsoft`: Contains the actual time series stock data
- `system.buckets.microsoft`: An internal collection used by MongoDB for time series data (automatically created when using time series collections in MongoDB 5.0+)
- `system.views`: Stores view definitions if any are present

For this recipe, we'll work with the `microsoft` collection.

5. Specify the collection to query. In this case, we are interested in the `microsoft` collection:

```
collection = db['microsoft']
```

- Query the database and load the results into a pandas DataFrame using the `find` method:

```
results = collection.find({})
msft_df = (pd.DataFrame(results)
           .set_index('date')
           .drop(columns='_id'))

msft_df.head()
>>
```

	close	low	volume	high	open
date					
2019-09-04	131.457260	130.358829	17995900	131.514567	131.142059
2019-09-05	133.768707	132.536556	26101800	134.083908	132.870864
2019-09-06	132.861359	132.001715	20824500	133.892908	133.749641
2019-09-09	131.352219	130.339762	25773900	133.482199	133.329371
2019-09-10	129.976837	128.477244	28903400	130.750506	130.664546

How it works...

In this recipe, we extracted stock price data (open/high/low/close/volume) from MongoDB into a pandas DataFrame. Let's break down each step of the process to understand how MongoDB stores and retrieves time series data.

Establishing the MongoDB connection

The first step is to connect to the database by creating a client object using `MongoClient`. This provides access to various methods, such as `list_database_names()` and `list_databases()`, and additional attributes, such as `address` and `HOST`.

`MongoClient()` accepts a connection string that follows MongoDB's URI format:

```
client = MongoClient("mongodb://localhost:27017")
```

This assumes no username or password is configured. Alternatively, you can explicitly provide *host* (string) and *port* (numeric) as positional arguments:

```
client = MongoClient('localhost', 27017)
```

The host string can either be the hostname or the IP address:

```
client = MongoClient('127.0.0.1', 27017)
```

To connect to localhost, which uses the default port (27017), you can establish a connection without providing any arguments (assuming no username or password is configured):

```
# using default values for host and port
client = MongoClient()
```

Further, you can explicitly supply named parameters:

```
client = MongoClient(host='127.0.0.1',
                     port=27017,
                     password='your_password',
                     username='your_username',
                     document_class=dict,
                     tz_aware=False,
                     connect=True)
```

Let's explore these parameters:

- **host**: This can be a **hostname**, **IP address**, or a MongoDB **URI**. It can also take a Python list of hostnames.
- **password**: Your assigned password. Read the note regarding special characters in the *How to do it* section.
- **Username**: Your assigned username. Read the note regarding special characters in the *How to do it* section.
- **document_class**: Specifies the class to use for the documents returned from your query. The default value is `dict`.
- **tz_aware**: Specifies whether datetime instances are time zone aware. The default is `False`, meaning they are naive (not time zone aware).
- **connect**: Whether to immediately connect to the MongoDB instance. The default value is `True`.

For additional parameters, you can reference the official documentation page here:

https://pymongo.readthedocs.io/en/stable/api/pymongo/mongo_client.html

Navigating databases and collections

After establishing a connection, the next step is to navigate to the specific database and collection containing our time series data. Unlike SQL databases, where you'd select a table, MongoDB requires you to specify both the database and collection you want to work with. The overall flow in terms of navigation before you can query and retrieve the documents is to specify the **database**, select the **collection** you are interested in, and then submit the **query**.

In the preceding example, our database was called `stock_data`, which contained a collection called `microsoft`. You can have multiple collections in a database and multiple documents in a collection. To think of this in terms of relational databases, recall that a collection is like a table and that documents represent rows in that table.

In PyMongo, you can specify a database using different syntax, as shown in the following code. Keep in mind that all these statements will produce a `pymongo.database.Database` object:

```
# Specifying the database
db = client['stock_data']
db = client.stock_data
db = client.get_database('stock_data')
```

The `get_database()` method can take additional arguments for the `codec_options`, `read_preference`, `write_concern`, and `read_concern` parameters, where the latter two are focused more on operations across nodes and how to determine whether the operation was successful or not.

Similarly, once you have the PyMongo database object, you can specify a collection using different syntax, as shown in the following example:

```
# Specifying the collection
collection = db.microsoft
collection = db['microsoft']
collection = db.get_collection('microsoft')
```

The `get_collection()` method provides additional parameters, similar to `get_database()`. The three syntax variations in the preceding example return a `pymongo.database.Collection` object.

Querying documents

Now that we've accessed the appropriate collection, we need to retrieve our time series documents. MongoDB offers several methods to fetch data, with different options depending on whether you need single or multiple documents. The `find()` method is your primary tool for retrieving documents, acting like a `SELECT` statement in SQL.

The `pymongo.database.Collection` provides access to additional built-in methods and attributes such as `find`, `find_one`, `find_one_and_delete`, `find_one_and_replace`, `find_one_and_update`, `update`, `update_one`, `update_many`, `delete_one`, and `delete_many`, to name a few.

Let's explore the different *retrieval* collection methods:

- `find()`: Retrieves multiple documents from a collection based on the submitted query.
- `find_one()`: Retrieves a single document from a collection based on the submitted query. If multiple documents match, then the first instance is returned.
- `find_one_and_delete()`: Finds a single document, like `find_one`, but it deletes it from the collection and returns it (the deleted document).
- `find_one_and_replace()`: Finds a single document and replaces it with a new document, returning either the original or the replaced document.
- `find_one_and_update()`: Finds a single document and updates it, returning either the original or the updated document. This is different than `find_one_and_replace`, since it updates the existing document instead of replacing the entire document.

Once you are at the collection level, you can start querying the data. In the recipe, you used `find()`, which you can think of as doing something similar to a `SELECT` statement in SQL.

In the *How to do it...* section, in step 5, you queried the entire collection to retrieve all the documents using this line of code:

```
collection.find({})
```

The empty dictionary, {}, in `find()` represents our filtering criteria. When you pass an empty filter criterion with {}, you are retrieving everything. This resembles `SELECT *` in a SQL database. Alternatively, one could also use `collection.find()` to retrieve all documents.

Filtering data with the MongoDB Query Language (MQL)

To retrieve specific time periods or data points from your time series collection, you'll need to create targeted queries using MongoDB's Query Language. MQL provides powerful operators that let you filter documents based on date ranges, numeric thresholds, and other conditions.

A query or a filter takes a key-value pair to return a select number of documents where the keys match the values specified. The following is an example of a query to find stocks with closing prices greater than 130:

```
query = {"close": {"$gt": 130}}
results = collection.find(query)
```

The results object is actually a **cursor** and does not contain the result set yet. You can loop through the cursor or convert it into a DataFrame. Generally, when `collection.find()` is executed, it returns a **cursor** (more specifically, a `pymongo.cursor.Cursor` object). This cursor object is just a pointer to the result set of the query, which allows you to iterate over the results. You can then use a `for` loop or `next()` method (think of a Python iterator). However, in this recipe, instead of looping through our cursor object, we conveniently converted the entire result set into a pandas DataFrame.

Here is an example of retrieving the result set into a pandas DataFrame:

```
df = pd.DataFrame(results)
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1256 entries, 0 to 1255
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   date        1256 non-null   datetime64[ns]
1   close       1256 non-null   float64
2   _id         1256 non-null   object
3   low         1256 non-null   float64
4   volume      1256 non-null   int64
5   high        1256 non-null   float64
6   open        1256 non-null   float64
dtypes: datetime64[ns](1), float64(4), int64(1), object(1)
memory usage: 68.8+ KB
```

Notice from the output that the `_id` column is added, which was not part of the original `MSFT.csv` file. This was automatically added by MongoDB as a **unique identifier** for each document in a collection.

In the preceding code, the query acts as a filter to only retrieve data where `close` values are greater than 130. PyMongo allows you to pass a dictionary (key-value pair) that specifies which fields are to be retrieved. Here is an example:

```
query = {"close": {"$gt": 130}}
projection = {
    "_id": 0,
    "date": 1,
    "close": 1,
    "volume": 1
}

results = collection.find(query, projection)

df = pd.DataFrame(results).set_index(keys='date')
```

In the preceding code, we specified that `_id` should not be returned, and only the `date`, `close`, and `volume` fields are to be returned.

Lastly, in our previous example, notice the `$gt` used in the query. This represents greater than, and more specifically, it translates to “*greater than 130*.” In MQL, operators start with the dollar sign `$`. Here is a sample list of commonly used operators in MQL:

- `$eq`: Matches values that are equal to a specified value.
 - Example: The query `{"close": {"$eq": 130}}` finds documents where the `close` field is exactly 130.
- `$gt`: Matches values that are greater than a specified value.
 - Example: The query `{"close": {"$gt": 130}}` finds documents where the `close` field is greater than 130.
- `$gte`: Matches values that are greater than or equal to a specified value.
 - Example: The query `{"close": {"$gte": 130}}` finds documents where the `close` field is greater than or equal to 130.
- `$lt`: Matches values that are less than a specified value.
 - Example: The query `{"close": {"$lt": 130}}` finds documents where the `close` field is less than 130.
- `$lte`: Matches values that are less than or equal to a specified value.
 - Example: The query `{"close": {"$lte": 130}}` finds documents where the `close` field is less than or equal to 130.

- **\$and:** Joins query clauses with a logical AND operator. All conditions must be true.
 - Example: The query `{"$and": [{"close": {"$gt": 130}}, {"volume": {"$lt": 20000000}}]}` finds documents where the close field is greater than 130 AND the volume is less than 20,000,000.
- **\$or:** Joins query clauses with a logical OR operator. At least one condition must be true.
 - Example: The query `{"$or": [{"close": {"$gt": 135}}, {"volume": {"$gt": 30000000}}]}` finds documents where the close field is greater than 135 OR the volume is greater than 30,000,000.
- **\$in:** Matches values specified in an array (list).
 - Example: The query `{"date": {"$in": [datetime.datetime(2019, 9, 4), datetime.datetime(2019, 9, 5), datetime.datetime(2019, 9, 6)]}}` finds documents where the date field matches any of the specified dates: September 4, 2019; September 5, 2019; September 6, 2019.

For a more comprehensive list of operators in MQL, you can visit the official documentation here:

<https://www.mongodb.com/docs/manual/reference/mql/query-predicates/>

Working with Time Series data

When working with financial time series such as stock prices, you can leverage MongoDB's query capabilities to perform common time series operations such as filtering by date ranges, identifying values above thresholds, or calculating metrics across specific periods.

Filtering by date ranges is straightforward with MQL operators – for example, to analyze a specific trading week from your MSFT data:

```
import datetime

query = {
    "date": {
        "$gte": datetime.datetime(2024, 8, 5),
        "$lte": datetime.datetime(2024, 8, 9)
    }
}

results = collection.find(query)
msft_df = (pd.DataFrame(results)
           .set_index('date')
           .drop(columns='_id'))
```


If you need daily averages or weekly trends, then MongoDB's aggregation pipeline provides tools for such time series analysis. Using aggregation pipelines, you can transform, group, and analyze your time series data to extract valuable insights. Here's an example of how to calculate the average closing price for each day of the week across a month of stock data:

```
# Example: Calculate average closing price by day of week
pipeline = [
    {"$match": {
        "date": {
            "$gte": datetime.datetime(2024, 8, 1),
            "$lte": datetime.datetime(2024, 8, 31)
        }
    }},
    {"$project": {
        # 1 for Sunday, 2 for Monday, etc.
        "dayOfWeek": {"$dayOfWeek": "$date"},
        "close": 1
    }},
    {"$group": {
        "_id": "$dayOfWeek",
        "avg_close": {"$avg": "$close"}
    }},
    {"$sort": {"_id": 1}}
]

day_analysis = list(collection.aggregate(pipeline))
print("Average closing price by day of week:")
for day in day_analysis:
    day_name = ["Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"][day["_id"]-1]
    print(f"{day_name}: ${day['avg_close']:.2f}")
>>
Average closing price by day of week:
Monday: $408.88
Tuesday: $412.70
Wednesday: $412.14
Thursday: $413.60
Friday: $413.09
```

The **aggregation pipeline** processes the time series data in stages:

1. First, it filters the documents to include only those from August 2024.

2. Next, it projects (selects) only the closing price and calculates which day of the week each data point falls on.
3. Then, it groups the data by day of week and calculates the average closing price from each group.
4. Finally, it sorts the results by day of week for easier interpretation.

There's more...

There are different ways to retrieve data from MongoDB using PyMongo. In the previous section, we used `db.collection.find()`, which always returns a cursor. As we discussed earlier, `find()` returns all the matching documents that are available in the specified collection. If you want to return the first occurrence of matching documents, then `db.collection.find_one()` would be the best choice and would return a dictionary object, not a cursor. Keep in mind that this only returns one document, as shown in the following example:

```
db.microsoft.find_one()
>>>
{'date': datetime.datetime(2019, 9, 4, 0, 0),
 'close': 131.45726013183594,
 '_id': ObjectId('66e30c09a07d56b6db2f446e'),
 'low': 130.35882921332006,
 'volume': 17995900,
 'high': 131.5145667829114,
 'open': 131.14205897649285}
```

When it comes to working with cursors, there are several ways you can traverse through the data:

Converting into a pandas DataFrame using `pd.DataFrame(cursor)`, as shown in the following code:

```
cursor = db.microsoft.find()
df = pd.DataFrame(cursor)
```

Converting into a Python list or tuple:

```
data = list(db.microsoft.find())
```

You can also convert the Cursor object into a Python list and then convert that into a pandas DataFrame, like this:

```
data = list(db.microsoft.find())
df = pd.DataFrame(data)
```

You can use `next()` to move the pointer to the next item in the result set:

```
cursor = db.microsoft.find()
cursor.next()
```

You can loop through the object – for example, with a for loop:

```
cursor = db.microsoft.find()
for doc in cursor:
    print(doc)
```

The previous code will loop through the entire result set. If, for example, you want to loop through the first five records, you can use the following:

```
cursor = db.microsoft.find()
for doc in cursor[0:5]:
    print(doc)
```

This specifies an index. Here, we are printing the first value:

```
cursor = db.microsoft.find()
cursor[0]
```

Note that if you provided a slice, such as `cursor[0:1]`, which is a range, then it will return a cursor object (not a document) that's limited to the specified range of documents.

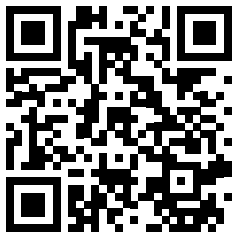
See also

- For more information on the **PyMongo** API, please refer to the official documentation, which you can find here: <https://pymongo.readthedocs.io/en/stable/index.html>

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



3

Persisting Time Series Data to Files

In this chapter, you will use the pandas library to persist your time series DataFrames to different file formats, such as **CSV**, **Excel**, **Parquet**, and **pickle** files. When performing analysis or data transformations on DataFrames, you essentially leverage pandas' in-memory analytics capabilities, offering great performance. However, being in memory means the data can easily be lost since it has not yet been persisted in disk storage.

When working with DataFrames, you will need to persist your data for future retrieval, creating backups, or sharing your data with others. The pandas library is bundled with a rich set of writer functions to persist your in-memory DataFrame (or series) to disk in various file formats. These writer functions allow you to store data on a local drive or a remote server location, such as a cloud storage filesystem, including **Google Drive**, **AWS S3**, **Azure Blob Storage**, and **Dropbox**.

In this chapter, you will explore writing to different file formats locally (on-premises) and cloud storage locations, such as Amazon Web Services (AWS), Google Cloud, and Azure.

Here are the recipes that will be covered in this chapter:

- Serializing time series data with pickle
- Writing to CSV and other delimited files
- Writing data to an Excel file
- Storing data in cloud storage (AWS, GCP, and Azure)



Don't miss the GitHub-exclusive recipe—*Chapter 3-1: Writing large datasets*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Technical requirements

In this chapter and beyond, we will extensively use pandas 2.3.3 (released September 29, 2025).

Throughout our journey, you will be installing several Python libraries to work in conjunction with pandas. These are highlighted in the *Getting ready* section for each recipe. You can also download Jupyter notebooks from the GitHub repository (<https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>) to follow along. You can download the datasets used in this chapter here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch3>.

Serializing time series data with pickle

When working with time series data in Python, you often need to save your processed datasets between analysis steps or preserve the exact state of your `DatetimeIndex` and other time-specific data structures. One technique is to serialize your data into a byte stream to store it in a file. In Python, the **pickle** module is a popular approach to object serialization and deserialization (the reverse of serialization), also known as *pickling* and *unpickling*.

Getting ready

The `pickle` module comes with Python, so no additional installation is needed.

In this recipe, we will explore two different methods for serializing data, commonly referred to as **pickling**.

You will be using the COVID-19 dataset provided by the *COVID-19 Data Repository by the Center for Systems Science and Engineering (CSSE)* at *Johns Hopkins University*, which you can download from the official GitHub repository here: https://github.com/CSSEGISandData/COVID-19/tree/master/csse_covid_19_data/csse_covid_19_time_series. You will be using the `time_series_covid19_confirmed_global.csv` file. Note that John Hopkins University is no longer updating the dataset as of March 10, 2023.

How to do it...

You will write to a **pickle** file using pandas' `DataFrame.to_pickle()` function and then explore an alternative option using the `pickle` library directly.

Writing to a pickle file using pandas

You will start by reading the COVID-19 time series data into a `DataFrame`, making some transformations, and then persisting the results to a pickle file for future analysis. This should resemble a typical scenario for persisting data that is still a work in progress (in terms of analysis):

1. To start, let's load the CSV data into a pandas `DataFrame`:

```
import pandas as pd
from pathlib import Path
file = Path(
    '../..../datasets/Ch3/time_series_covid19_confirmed_global.csv')

df = pd.read_csv(file)
df.head()
```

You can observe from the output that this is a wide DataFrame with 1,147 columns, where each column represents the data's collection date, starting from 1/22/20 to 3/9/23.

- Let's assume that part of the analysis is to focus on the United States and only data collected in the summer of 2021 (June, July, August, and September). You will transform the DataFrame by applying the necessary filters, and then unpivot the data so that the dates are in rows as opposed to columns (converting from a **wide** format DataFrame that has dates as columns to a **long** format DataFrame that has dates as rows):

```
# filter data where Country is United States
df_usa = df[df['Country/Region'] == 'US']
# filter columns from June to end of September
df_usa_summer = df_usa.loc[:, '6/1/21':'9/30/21']

# unpivot using pd.melt()
df_usa_summer_unpivoted = \
    pd.melt(df_usa_summer,
            value_vars=df_usa_summer.columns,
            value_name='cases',
            var_name='date').set_index('date')

df_usa_summer_unpivoted.index = \
    pd.to_datetime(df_usa_summer_unpivoted.index, format="%m/%d/%y")
```

You have now transformed the DataFrame from a wide format, where each date is a separate column, to a long format using `pd.melt()`. This unpivoting process is essential for time series analysis, as it organizes the data with dates as rows, making it more suitable for various analytical tasks.

- Inspect the `df_usa_summer_unpivoted` DataFrame and print the first five records:

```
df_usa_summer_unpivoted.info()
>>

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 122 entries, 2021-06-01 to 2021-09-30
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype
---  -
0    cases   122 non-null        int64
dtypes: int64(1)
memory usage: 1.9 KB
```

```
df_usa_summer_unpivoted.head()
>>
```

	cases
date	
2021-06-01	33407540
2021-06-02	33424131
2021-06-03	33442100
2021-06-04	33459613
2021-06-05	33474770

You filtered the dataset and transformed it from a wide DataFrame to a long time series DataFrame.

- Let's say you are now satisfied with the dataset and ready to pickle (serialize) the dataset. You will write the DataFrame to a `covid_usa_summer_2021.pkl` file using the `DataFrame.to_pickle()` function:

```
output = Path('../datasets/Ch3/covid_usa_summer_2021.pkl')
df_usa_summer_unpivoted.to_pickle(output)
```

Pickling preserves the schema of the DataFrame. When you ingest the pickled data again (de-serialization), you will return the DataFrame in its original construct – for example, with a `DatetimeIndex` type.

- Read the pickle file using the `pandas.read_pickle()` reader function and inspect the DataFrame:

```
unpickled_df = pd.read_pickle(output)
unpickled_df.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 122 entries, 2021-06-01 to 2021-09-30
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype
---  -
0    cases   122 non-null     int64
dtypes: int64(1)
memory usage: 1.9 KB
```

From the preceding example, you were able to deserialize the data using `pandas.read_pickle()`, into a DataFrame, with all the previously committed transformations and datatypes preserved.

Writing a pickle file using the pickle library

Python comes shipped with the pickle library, which you can import and use to serialize (pickle) objects using `dump` (to write) and `load` (to read). In the following steps, you will use `pickle.dump()` and `pickle.load()` to serialize and then deserialize the `df_usa_summer_unpivoted` DataFrame.

1. Import the pickle library:

```
import pickle
```

2. You then persist the `df_usa_summer_unpivoted` DataFrame using the `dump()` method:

```
file_path = Path('../..../datasets/Ch3/covid_usa_summer_2021_v2.pkl')

with open(file_path, "wb") as file:
    pickle.dump(df_usa_summer_unpivoted, file)
```

Notice the mode used is "wb" because we are writing in binary mode (written as raw bytes).

3. You can read the file and inspect the DataFrame using the `load()` method. Notice in the following code that the ingested object is a pandas DataFrame, even though you used `pickle.load()` instead of `Pandas.read_pickle()`. This is because pickling preserved the schema and data structures:

```
with open(file_path, "rb") as file:
    df = pickle.load(file)
type(df)
>>
pandas.core.frame.DataFrame
```

Notice the mode used is "rb" because we are reading in binary mode (read as raw bytes).



While pickle is convenient, it's important to note that you should only unpickle data from trusted sources, as it can execute arbitrary code during unpickling.

Using the pickle library directly is handy when you need to serialize objects other than pandas DataFrames or Series, or if you want to serialize multiple distinct objects into the same file.

How it works...

In Python, pickling is the process of serializing any Python object. More concretely, it uses a binary serialization protocol to convert objects into binary information, which is not a human-readable format. The protocol allows us to reconstruct (deserialize) the pickled file, in binary format, into its original content without losing valuable information. As in the preceding examples, we confirmed that a time series DataFrame, when reconstructed (de-serialization), returned to its exact form (schema).

The pandas `DataFrame.to_pickle()` function has two additional parameters that are important to know. The first is the compression parameter, which is also available in other writer functions such as `to_csv()`, `to_json()`, and `to_parquet()`, to name a few.

In the case of the `DataFrame.to_pickle()` function, the default compression value is set to `infer`, which lets pandas determine which compression mode to use based on the file extension provided. In the previous example, we used `DataFrame.to_pickle(output)`, where `output` was defined with a `.pkl` file extension, such as `covid_usa_summer_2021.pkl` in our example. If you instead provided a path ending in `.zip`, such as `covid_usa_summer_2021.zip`, pandas would automatically apply **ZIP** compression, and the output would be a compressed binary serialized file stored in ZIP format. You can try the following example:

```
zip_output = Path('../../datasets/Ch3/covid_usa_summer_2021.zip')

# Write the Dataframe
df_usa_summer_unpivoted.to_pickle(zip_output)

# Read the Dataframe
pd.read_pickle(zip_output)
```

Supported compression formats include `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, and `'zstd'`.

The second parameter is `protocol`. By default, the `DataFrame.to_pickle()` writer function uses the highest protocol version available in your Python environment (version dependent). As of Python 3.8+, protocol version 5 is the highest. According to the Pickle documentation, there are six different protocols to choose from when pickling, starting from protocol version 0 to the latest protocol version, 5.

Outside of pandas, you can check what the highest protocol configuration is by using the following command:

```
pickle.HIGHEST_PROTOCOL
>> 5
```

Similarly, by default, `pickle.dump()` uses the `HIGHEST_PROTOCOL` value if no other value was provided. The construct looks like the following code:

```
with open(file_path, "wb") as file:
    pickle.dump(df_usa_summer_unpivoted,
                file,
                pickle.HIGHEST_PROTOCOL)

with open(file_path, "wb") as file:
    pickle.dump(df_usa_summer_unpivoted,
                file,
                5)
```

The preceding two code snippets are equivalent.

Generally, using the `HIGHEST_PROTOCOL` is recommended as newer protocols are often more efficient and can handle a wider range of Python objects.

There's more...

One of the advantages of pickling as a binary serialization method is that we can pretty much pickle most Python objects, whether a Python dictionary, a machine learning model, a Python function, or a more complex data structure, such as a pandas DataFrame. However, there are some limitations on certain objects, such as lambda and nested functions.

Let's examine how you can pickle a function and its output. You will create a `covid_by_country` function that takes three arguments: *the CSV file to read*, *the number of days back*, and *the country*. The function will return a time series DataFrame. You will then pickle the function, the function's output, and its plot as well:

```
def covid_by_country(file, days, country):
    ts = pd.read_csv(file)
    ts = ts[ts['Country/Region'] == country]
    final = ts.iloc[:, -days:].sum()
    final.index = pd.to_datetime(final.index, format="%m/%d/%y")
    return final

file = Path('../datasets/Ch3/time_series_covid19_confirmed_global.csv')

us_past_120_days = covid_by_country(file, 120, 'US')

plot_example = us_past_120_days.plot(
    title=f'COVID confirmed case for US',
    xlabel='Date',
    ylabel='Number of Confirmed Cases'
);
```

Before pickling your objects, you can further enhance the content by adding additional information to remind you what the content is all about. In the following code, you will serialize the function and the returned DataFrame with additional information (known as metadata) encapsulated using a Python dictionary:

```
from datetime import datetime

metadata = {
    'date': datetime.now(),
    'data': '''
        COVID-19 Data Repository by the
        Center for Systems Science and Engineering (CSSE)
    '''
}
```

```

        at Johns Hopkins University'
        '''
        'author': 'Tarek Atwan',
        'version': 1.0,
        'function': covid_by_country,
        'example_df' : us_past_120_days,
        'example_plot': plot_example
    }

    file_path = Path('../..../datasets/Ch3/covid_data.pkl')

    with open(file_path, 'wb') as file:
        pickle.dump(metadata, file)

```

To gain a better intuition of how this works, you can load the content and deserialize using `pickle.load()`:

```

with open(file_path, 'rb') as file:
    content = pickle.load(file)
content.keys()

>>
dict_keys(['date', 'data', 'author', 'version', 'function', 'example_df',
'example_plot'])

```

You can retrieve and use the function, as shown in the following code:

```

file_path = Path(
    '../..../datasets/Ch3/time_series_covid19_confirmed_global.csv')

loaded_func = content['function']
loaded_func(file_path, 120, 'China').tail()

>>
2023-03-05    4903524
2023-03-06    4903524
2023-03-07    4903524
2023-03-08    4903524
2023-03-09    4903524
dtype: int64

```

You can also retrieve the previous DataFrame stored for the US:

```
loaded_df = content['example_df']
loaded_df.tail()

>>
2023-03-05    103646975
2023-03-06    103655539
2023-03-07    103690910
2023-03-08    103755771
2023-03-09    103802702
dtype: int64
```

You can also load the view of the figure you just stored.

```
loaded_plot = content['example_plot']
loaded_plot.get_figure()
```



While pickling plot objects works, sometimes saving the plot directly to an image file (using `plot_example.get_figure().savefig('my_plot.png')`) and storing the filename can be more robust if you share pickled files across different environments or library versions.

The preceding examples demonstrate how pickling can be useful to store objects and additional metadata information. This can be helpful when storing a work-in-progress or performing multiple experiments and wanting to keep track of them and their outcomes. A similar approach can be taken when working with machine learning experiments, as you can store your model and any related information around the experiment and its outputs.

See also

- For more information about `Pandas.DataFrame.to_pickle`, please visit this page: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_pickle.html.
- For more information about the Python Pickle module, please visit this page: <https://docs.python.org/3/library/pickle.html>.

Writing to CSV and other delimited files

In this recipe, you will export a DataFrame as a CSV file and leverage the different parameters in the `DataFrame.to_csv()` writer function.

Getting ready

The file is provided in the GitHub repository for this book, which you can find here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>. The file you will be working with is named `movieboxoffice.csv`, which you will read first to create your DataFrame.

To prepare for this recipe, you will read the file into a DataFrame with the following code:

```
import pandas as pd
from pathlib import Path
filepath = Path('../..../datasets/Ch3/movieboxoffice.csv')

movies = pd.read_csv(filepath,
                      header=0,
                      parse_dates=[0],
                      index_col=0,
                      usecols=['Date', 'Daily'],
                      date_format="%d-%b-%y")

movies.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 128 entries, 2021-04-26 to 2021-08-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  -
 0   Daily   128 non-null      object
dtypes: object(1)
memory usage: 2.0+ KB
```

You now have a time series DataFrame with an index of type DatetimeIndex.

How to do it...

Writing a DataFrame to a CSV file is pretty straightforward with pandas. The DataFrame object has access to many writer methods, such as `.to_csv`, which is what you will be using in the following steps:

1. You will use the pandas DataFrame writer method to persist the DataFrame as a CSV file. The method has several parameters, but at a minimum, all you need is to pass a file path and a filename:

```
output = Path('../..../datasets/Ch3/df_movies.csv')
movies.to_csv(output)
```

The CSV file created is comma-delimited by default (hence the name **CSV – Comma Separated Values**).

2. To change the delimiter, use the `sep` parameter and pass in a different argument. In the following code, you are creating a pipe (|) delimited file:

```
output = Path('../..../datasets/Ch3/piped_df_movies.csv')
movies.to_csv(output, sep='|')
```

3. Read the pipe-delimited file and inspect the resulting DataFrame object:

```
movies_df = pd.read_csv(output, sep='|')
movies_df.info()

>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 128 entries, 0 to 127
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    Date    128 non-null      object
1   Daily    128 non-null      object
dtypes: object(2)
memory usage: 2.1+ KB
```

Notice from the preceding output that some information was lost when reading the CSV file. For example, the Date column in the original movies DataFrame was not a column but an index of type DatetimeIndex. The current movies_df DataFrame does not have an index of type DatetimeIndex (the index is now a RangeIndex type, which is just a range for the row numbers). This means you will need to configure the read_csv() function and pass the necessary arguments to parse the file appropriately. This highlights a key difference from the pickle format we saw in the previous recipe (*Serializing time series data with pickle*) – CSV files don't preserve pandas-specific metadata like index types and column datatypes. Generally, CSV file formats do not preserve index type or column datatype information.

For instance, when reading piped_df_movies.csv back in, you would need to specify sep='|', index_col='Date', and parse_dates=['Date'] within pd.read_csv() to attempt restoring the original DatetimeIndex and data types, similar to how we loaded the initial data:

```
pd.read_csv(output, sep='|', index_col=['Date'], parse_dates=['Date'])
```

How it works...

The default behavior for DataFrame.to_csv() is to write a **comma-delimited** CSV file based on the default sep parameter, which is set to ",". You can overwrite this by passing a different delimiter, such as tab ("\t"), pipe ("|"), or semicolon (";").

The following code shows examples of different **delimiters** and their representations:

```
demo = pd.read_csv(filepath, usecols=['Date', 'DOW', 'Daily', 'To Date'])
demo.head(3).to_csv('tab_demo.csv', sep='\t', index=False)
demo.head(3).to_csv('comma_demo.csv', sep=',', index=False)
demo.head(3).to_csv('semicolon_demo.csv', sep=';', index=False)
demo.head(3).to_csv('pipe_demo.csv', sep='|', index=False)
```

If you inspect the files, you will observe the following:

```
# tab "\t"
Date DOW      Daily To Date
26-Apr-21    Friday $125,789.89   $125,789.89
27-Apr-21    Saturday   $99,374.01   $225,163.90
28-Apr-21    Sunday   $82,203.16   $307,367.06

# comma ","
Date,DOW,Daily,To Date
26-Apr-21,Friday," $125,789.89 "," $125,789.89 "
27-Apr-21,Saturday," $99,374.01 "," $225,163.90 "
28-Apr-21,Sunday," $82,203.16 "," $307,367.06 "

# semicolon ";"
Date;DOW;Daily;To Date
26-Apr-21;Friday; $125,789.89 ; $125,789.89
27-Apr-21;Saturday; $99,374.01 ; $225,163.90
28-Apr-21;Sunday; $82,203.16 ; $307,367.06

# pipe "|"
Date|DOW|Daily|To Date
26-Apr-21|Friday| $125,789.89 | $125,789.89
27-Apr-21|Saturday| $99,374.01 | $225,163.90
28-Apr-21|Sunday| $82,203.16 | $307,367.06
```

There's more...

Notice in the preceding example that the comma-separated string values are not encapsulated within double quotes (""). What will happen if our string object contains commas (,) and we write it to a comma-separated CSV file? Let's see how pandas handles this scenario.

In the following code, we will create a person DataFrame:

```
import pandas as pd
person = pd.DataFrame({
    'name': ['Bond, James', 'Smith, James', 'Bacon, Kevin'],
    'location': ['Los Angeles, CA', 'Phoenix, AZ', 'New York, NY'],
    'net_worth': [10000, 9000, 8000]
})

print(person)

>>
```

	name	location	net_worth
0	Bond, James	Los Angeles, CA	10000
1	Smith, James	Phoenix, AZ	9000
2	Bacon, Kevin	New York, NY	8000

Now, export the DataFrame to a CSV file. You will specify `index=False` to ignore the index (row numbers) when exporting:

```
person.to_csv('person_a.csv', index=False)
```

If you inspect a `person_a.csv` file, you will see the following representation (notice the double quotes added by pandas):

```
name,location,net_worth
"Bond, James","Los Angeles, CA",10000
"Smith, James","Phoenix, AZ",9000
"Bacon, Kevin","New York, NY",8000
```

The `to_csv()` function has a quoting parameter with a default value set to `csv.QUOTE_MINIMAL`. This comes from the Python `csv` module, which is part of the Python installation. The `QUOTE_MINIMAL` argument only quotes fields that contain special characters, such as a comma (`,`).

The `csv` module provides four constants we can pass as arguments to the quoting parameter within the `to_csv()` function. These include the following:

- `csv.QUOTE_ALL`: Quotes all the fields, whether numeric or non-numeric
- `csv.QUOTE_MINIMAL`: The default option in the `to_csv()` function, which quotes values that contain special characters
- `csv.QUOTE_NONNUMERIC`: Quotes all non-numeric fields
- `csv.QUOTE_NONE`: To not quote any field

To better understand how these values can impact the output CSV, you will test passing different quoting arguments in the following example. This is done using the `person` DataFrame:

```
import csv
person.to_csv('person_b.csv',
              index=False,
              quoting=csv.QUOTE_ALL)
person.to_csv('person_c.csv',
              index=False,
              quoting=csv.QUOTE_MINIMAL)
person.to_csv('person_d.csv',
              index=False,
              quoting= csv.QUOTE_NONNUMERIC)
person.to_csv('person_e.csv',
              index=False,
              quoting= csv.QUOTE_NONE, escapechar='\\t')
```


Note that in the preceding example, when using `csv.QUOTE_NONE`, you must provide an additional argument for the `escapechar` parameter; otherwise, it will throw an error. This is because when quotes are disabled, pandas needs another way to handle (escape) delimiter characters that appear in your data (e.g., commas). The `escapechar` tells pandas which character (e.g., `'\t'`, which means tab) to use immediately before these special characters (e.g., commas) in the output file, signaling they are part of the data and not a separator.

Now, if you open and inspect these files, you should see the following representations:

```
person_b.csv
"name","location","net_worth"
"Bond, James","Los Angeles, CA","10000"
"Smith, James","Phoenix, AZ","9000"
"Bacon, Kevin","New York, NY","8000"

person_c.csv
name,location,net_worth
"Bond, James","Los Angeles, CA",10000
"Smith, James","Phoenix, AZ",9000
"Bacon, Kevin","New York, NY",8000

person_d.csv
"name","location","net_worth"
"Bond, James","Los Angeles, CA",10000
"Smith, James","Phoenix, AZ",9000
"Bacon, Kevin","New York, NY",8000

person_e.csv
name,location,net_worth
Bond, James,Los Angeles, CA,10000
Smith, James,Phoenix, AZ,9000
Bacon, Kevin,New York, NY,8000
```

In the `person_e.csv` output, the `escapechar='\t'` adds a tab before commas to prevent parsing errors.

See also

- For more information on the `Pandas.DataFrame.to_csv()` function, please refer to this page: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html.
- For more information on the CSV module, please refer to this page: <https://docs.python.org/3/library/csv.html>.

Writing data to an Excel file

In this recipe, you will export a `DataFrame` as an Excel file format and leverage the different parameters available in the `DataFrame.to_excel()` writer function.

Getting ready

In the *Reading data from an Excel file* recipe in *Chapter 1*, you had to install `openpyxl` as the engine for reading Excel files with `pandas.read_excel()`. For this recipe, you will use the same `openpyxl` as the engine for writing Excel files with `DataFrame.to_excel()`.

To install `openpyxl` using `conda`, run the following:

```
conda install openpyxl
```

You can also use `pip`:

```
pip install openpyxl
```

The file is provided in the GitHub repository for this book, which you can find here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>. The file is named `movieboxoffice.csv`.

To prepare for this recipe, you will read the file into a `DataFrame` with the following code:

```
import pandas as pd
from pathlib import Path
filepath = Path('../..../datasets/Ch3/movieboxoffice.csv')

movies = pd.read_csv(filepath,
                     header=0,
                     parse_dates=[0],
                     index_col=0,
                     usecols=['Date', 'Daily'],
                     date_format="%d-%b-%y")
```

How to do it...

To write the `DataFrame` to an Excel file, you need to provide the writer function with `filename` and `sheet_name` parameters. The file name contains the file path and name. Make sure the file extension is `.xlsx` since you are using `openpyxl`.

The `DataFrame.to_excel()` method will determine which engine to use based on the file extension – for example, `.xlsx` or `.xls`. You can also explicitly specify which engine to use with the `engine` parameter, as shown in the following code:

1. Determine the location of the file output and create the file path, desired sheet name, and engine for the `DataFrame.to_excel()` writer function:

```
output = Path('../..../datasets/Ch3/daily_boxoffice.xlsx')

movies.to_excel(output,
                sheet_name='movies_data',
                engine='openpyxl', # default engine for xlsx files
                index=True)
```

The preceding code will create a new Excel file in the specified location. You can open and inspect the file, as shown in the following figure:

	A	B
1	Date	Daily
2	2021-04-26 00:00:00	\$125,789.89
3	2021-04-27 00:00:00	\$99,374.01
4	2021-04-28 00:00:00	\$82,203.16
5	2021-04-29 00:00:00	\$33,530.26
6	2021-04-30 00:00:00	\$30,105.24
7	2021-05-01 00:00:00	\$22,955.06
8	2021-05-02 00:00:00	\$19,579.02
9	2021-05-03 00:00:00	\$37,018.43
10	2021-05-04 00:00:00	\$55,900.40
11	2021-05-05 00:00:00	\$40,984.73
12	2021-05-06 00:00:00	\$9,724.08
13	2021-05-07 00:00:00	\$11,365.60
14	2021-05-08 00:00:00	\$7,646.02
15	2021-05-09 00:00:00	\$6,808.62
16	2021-05-10 00:00:00	\$14,673.11
17	2021-05-11 00:00:00	\$24,949.01
18	2021-05-12 00:00:00	\$17,715.13
19	2021-05-13 00:00:00	\$4,253.08
20	2021-05-14 00:00:00	\$5,192.90
21	2021-05-15 00:00:00	\$3,418.58
22	2021-05-16 00:00:00	\$3,079.61

movies_data +

Figure 3.1: Example output from the `daily_boxoffice.xlsx` file

Note that the sheet name is `movies_data`. In the Excel file, you will notice that the date is not in the format you would expect. Let's say the expectation was for the Date column to be in a specific format, such as `MM-DD-YYYY`.



Reading the same file using `read_excel` will read the Date column properly, as expected.

2. To achieve this, you will use another pandas-provided class, the `pandas.ExcelWriter` class. While this class `datetime_format` and `date_format` properties, these two work properly with the `xlsxwriter` engine but not with `openpyxl` due to an open bug (as of this writing). Since **openpyxl** has advantages over **xlsxwriter**, such as appending to existing Excel files, we'll use **openpyxl**'s direct cell formatting capabilities via the `number_format` property to fix our date formatting issue. The following code shows how this can be achieved:

```
date_col = 'Date'
with pd.ExcelWriter(
    output,
    engine='openpyxl',
    mode='a',
    if_sheet_exists='replace'
) as writer:
    movies.to_excel(writer, sheet_name='movies_fixed_dates', index=True)

    worksheet = writer.sheets['movies_fixed_dates']

    for col in worksheet.iter_cols():
        header = col[0] # capture headers
        if header.value == date_col:
            for row in range(
                2, # skip first row
                worksheet.max_row+1
            ):
                worksheet.cell(
                    row,
                    header.column
                ).number_format='MM-DD-YYYY'
```

The following is a representation of what the new output would look like. This was accomplished by passing `MM-DD-YYYY` to the `datetime_format` property of the writer object:

	A	B	C
1	Date	Daily	
2	04-26-2021	\$125,789.89	
3	04-27-2021	\$99,374.01	
4	04-28-2021	\$82,203.16	
5	04-29-2021	\$33,530.26	
6	04-30-2021	\$30,105.24	
7	05-01-2021	\$22,955.06	
8	05-02-2021	\$19,579.02	
9	05-03-2021	\$37,018.43	
10	05-04-2021	\$55,900.40	
11	05-05-2021	\$40,984.73	
12	05-06-2021	\$9,724.08	
13	05-07-2021	\$11,365.60	
14	05-08-2021	\$7,646.02	
15	05-09-2021	\$6,808.62	
16	05-10-2021	\$14,673.11	
17	05-11-2021	\$24,949.01	
18	05-12-2021	\$17,715.13	
19	05-13-2021	\$4,253.08	
20	05-14-2021	\$5,192.90	
21	05-15-2021	\$3,418.58	
22	05-16-2021	\$3,079.61	

◀ ▶
movies_data
movies_fixed_dates

Figure 3.2: Using `pd.ExcelWriter` and `number_format` to update the Date column format to `MM-DD-YYYY`

How it works...

The `DataFrame.to_excel()` method, by default, creates a new Excel file if it doesn't exist or overwrites the file if it exists. To append to an existing Excel file or write to multiple sheets, you will need to use the `Pandas.ExcelWriter` class. The `ExcelWriter()` class has a `mode` parameter that can accept either "w" for write or "a" for append. As of this writing, `xlsxwriter` does not support the append mode, while `openpyxl` supports both modes.

Additionally, when using append mode (`mode="a"`), you will need to specify how to handle existing sheets through the `if_sheet_exists` parameter, which accepts one of three values:

- **error**: Raises a **ValueError** exception if the sheet exists
- **replace**: Overwrites the existing worksheet.
- **new**: Creates a new worksheet with a new name. If you re-execute the preceding code and update `if_sheet_exists='new'`, then a new sheet will be created and named `movies_fixed_dates1`.
- **overlay**: Allows you to write new data to an existing sheet without erasing the existing content.

When writing data to Excel, you'll often encounter missing values (NaN) in your DataFrame. By default, pandas represents these missing values as empty cells in Excel. However, you might want to display something more descriptive, such as "N/A", "Missing", or "Unknown" instead. The `na_rep` parameter allows you to specify how missing values should be represented in the Excel output – for example:

```
movies.to_excel(output,
                 sheet_name='movies_with_na',
                 engine='openpyxl',
                 index=True,
                 na_rep='Missing')
```

This will replace all NaN values in the pandas DataFrame with the string "Missing" in the Excel output.

There's more...

If you need to create multiple worksheets in the same Excel file, then `ExcelWriter` can be used to achieve this. For example, assume the goal is to split each month's data into its own sheet and name the sheet accordingly. In the following code, you will add a `Month` column and use that to split that DataFrame by month, using `groupby` to write each group into a new sheet.

First, let's create our helper function `sheet_date_format`, to format our `Date` column in each sheet to the **MM-DD-YYYY** format:

```
def sheet_date_format(sheet_name, writer, date_col):
    worksheet = writer.sheets[sheet_name]

    for col in worksheet.iter_cols():
        header = col[0]
        if header.value == date_col:
            for row in range(2, worksheet.max_row+1):
                worksheet.cell(
                    row,
                    header.column
                ).number_format='MM-DD-YYYY'
```

The next piece of code will add a Month column to the movies DataFrame, and then write each month to individual sheets and name each sheet with the corresponding month name:

```
movies['Month'] = movies.index.month_name()

output = Path('../datasets/Ch3/boxoffice_by_month.xlsx')
with pd.ExcelWriter(output, engine='openpyxl') as writer:
    for month, data in movies.groupby('Month'):
        data.to_excel(writer, sheet_name=month)
        sheet_date_format(month, writer, date_col='Date')
```

The preceding code will create a new Excel file named `boxoffice_by_month.xlsx` with five sheets for each month, as shown in the following figure:

	A	B	C	D	E	F
1	Date	Daily	Month			
2	04-26-2021	\$125,789.89	April			
3	04-27-2021	\$99,374.01	April			
4	04-28-2021	\$82,203.16	April			
5	04-29-2021	\$33,530.26	April			
6	04-30-2021	\$30,105.24	April			
7						
8						
9						
	April	August	July	June	May	+

Figure 3.3: Each month in the movies DataFrame was written to its own sheet in Excel

See also

The pandas `to_excel()` method and `ExcelWriter` class make writing DataFrames to an Excel file very convenient. If you want a more granular control outside of pandas DataFrames, you should consider exploring the `openpyxl` library you installed as the reader/writer engine – for example, the `openpyxl` library has a `dataframe` module (`openpyxl.utils.dataframe`) for working with pandas DataFrames. An example is the `dataframe_to_rows()` function.

- To learn more about `Pandas.DataFrame.to_excel()`, please refer to https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html
- To learn more about `Pandas.ExcelWriter()`, please refer to <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.ExcelWriter.html#pandas.ExcelWriter>
- To learn more about `openpyxl`, please refer to <https://openpyxl.readthedocs.io/en/stable/index.html>
- To learn more about `openpyxl.utils.dataframe`, please refer to <https://openpyxl.readthedocs.io/en/stable/pandas.html#working-with-pandas-dataframes>

Storing data in cloud storage (AWS, GCP, and Azure)

In this recipe, you will use pandas to write to cloud storage such as Amazon S3, Google Cloud Storage, and Azure Blob Storage. Several of the pandas writer functions support writing to cloud storage through the `storage_options` parameter.

Getting ready

In the *Reading data from a URL* recipe in *Chapter 1*, you were instructed to install `boto3` and `s3fs` to read from AWS S3 buckets. In this recipe, you will be leveraging the same libraries in addition to the needed libraries for Google Cloud Storage (`gcsfs`) and Azure Blob Storage (`adlfs`).

To install using `pip`, you can use this:

```
pip install boto3 s3fs
pip install google-cloud-storage gcsfs
pip install adlfs azure-storage-blob azure-identity
```

To install using `conda`, you can use this:

```
conda install -c conda-forge boto3 s3fs -y
conda install -c conda-forge google-cloud-storage gcsfs -y
conda install -c conda-forge adlfs azure-storage-blob azure-identity -y
```

You will work with the *boxoffice_by_month.xlsx* file we created in the previous recipe, *Writing data to an Excel file*. The file is provided in the GitHub repository for this book, which you can find here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook>.

To prepare for this recipe, you will read the file into a `DataFrame` with the following code:

```
import pandas as pd
from pathlib import Path
source = "../../datasets/Ch3/boxoffice_by_month.xlsx"
movies = pd.concat(pd.read_excel(source,
                                sheet_name=None,
                                index_col='Date',
                                parse_dates=True)).droplevel(0)

print(movies.head())
>>
```

	Daily	Month
Date		
2021-04-26	\$125,789.89	April
2021-04-27	\$99,374.01	April
2021-04-28	\$82,203.16	April
2021-04-29	\$33,530.26	April
2021-04-30	\$30,105.24	April

Notice the movie DataFrame has two columns (Daily and Month) and a DatetimeIndex (Date).

Next, you will store your AWS, Google Cloud, and Azure credentials in a config `cloud.cfg` file outside your Python script. Then, use `configparser` to read and store the values in Python variables. You do not want your credentials exposed or hardcoded in your code.

Here is an example of a configuration file, which you can create and store as `cloud.cfg`:

```
[AWS]
aws_access_key=<your_access_key>
aws_secret_key=<your_secret_key>

[GCP]
key_file_path=<GCPKeyFileexample.json>

[AZURE]
storage_account_key=<your_storageaccount_key>
```

We can then load an `aws.cfg` file using `config.read()`:

```
import configparser
config = configparser.ConfigParser()
config.read('cloud.cfg')

AWS_ACCESS_KEY = config['AWS']['aws_access_key']
AWS_SECRET_KEY = config['AWS']['aws_secret_key']
AZURE_ACCOUNT_KEY = config['AZURE']['storage_account_key']
GCP_KEY_FILE = config['GCP']['key_file_path']
```

How to do it...

Several of the pandas writer functions support writing directly to a remote or cloud storage filesystem using, for example, AWS's `s3://`, Google's `gs://`, and Azure's `abfs://` and `az://` protocols. These writer functions provide the `storage_options` parameter to support working with remote file storage systems. This is in part because pandas utilized `fsspec` to handle system non-HTTP (s) URLs, such as those specific to each cloud storage. For each cloud, you will need to use the specific filesystem implementation – for example, `s3fs` for AWS S3, `gcsfs` for Google Cloud, and `adlfs` for Azure.

The `storage_options` parameter takes a Python dictionary to provide additional information, such as credentials, tokens, or any information the cloud provider requires as a key-value pair.

Writing to Amazon S3 using pandas

In this section, you will write the `movies` DataFrame to the `tscookbook-private` S3 bucket as CSV and Excel files using pandas:

Several pandas writer functions, such as `to_csv`, `to_parquet`, and `to_excel`, allow you to pass AWS S3-specific credentials (key and secret) as specified in `s3fs` through the `storage_options` parameter. The following code shows how you can utilize `to_csv` and `to_excel` to write your `movies` DataFrame to the `tscookbook` S3 bucket as `movies_s3.csv` and `movies_s3.xlsx`:

```
# Writing to Amazon S3

movies.to_csv(
    's3://tscookbook-private/movies_s3.csv',
    storage_options={
        'key': AWS_ACCESS_KEY,
        'secret': AWS_SECRET_KEY
    })

movies.to_excel(
    's3://tscookbook-private/movies_s3.xlsx',
    storage_options={
        'key': AWS_ACCESS_KEY,
        'secret': AWS_SECRET_KEY
    })
```

The `storage_options` parameter allows us to pass in the AWS credentials required for authentication. Under the hood, pandas utilizes `fsspec` and `s3fs` to handle the connection to S3, enabling seamless reading and writing of files.

Writing to Google Cloud Storage using pandas

In this section, you will write the `movies` DataFrame to the `tscookbook` bucket on Google Cloud Storage as CSV and Excel files using pandas.

When working with **Google Cloud**, you will use a service account private key stored as a JSON file. This is generated and downloaded from Google Cloud. In `storage_options`, you will pass the file path through the `token` parameter. The following code shows how you can utilize `to_csv` and `to_excel` to write your `movies` DataFrame to the `tscookbook` bucket as `movies_gs.csv` and `movies_gs.xlsx`:

```
# Writing to Google Cloud Storage

movies.to_csv('gs://tscookbook/movies_gs.csv',
              storage_options={'token': GCP_KEY_FILE})

movies.to_excel('gs://tscookbook/movies_gs.xlsx',
                storage_options={'token': GCP_KEY_FILE})
```

In this example, `GCP_KEY_FILE` contains the path to the downloaded file (e.g., `valiant-cycle-431419-h2-c5bc48019533.json`).

Writing to Azure Blob Storage using pandas

In this section, you will write the `movies` `DataFrame` to Azure Blob Storage in a container named `objects` as a CSV file using `pandas`:

When working with **Azure Blob Storage**, you use either the `abfs://` or `az://` protocols. Generally, `abfs://` is the recommended scheme for Azure Data Lake Storage Gen2, while `az://` is more general for Blob Storage.

In `storage_options` you will pass `account_key`, which is your API key for the storage account on Azure. The following code shows how you can utilize `to_csv` to write your `movies` `DataFrame` to a container named `objects`. The following three code snippets are equivalent and illustrate the different URI schemes and `storage_options` you can use with Azure Blob Storage:

```
# Writing to Azure Blob Storage

movies.to_csv(
    "abfs://objects@tscookbook.dfs.core.windows.net/movies_abfs.csv",
    storage_options={
        'account_key': AZURE_ACCOUNT_KEY
    })

movies.to_csv(
    "az://objects@tscookbook.dfs.core.windows.net/movies_az.csv",
    storage_options={
        'account_key': AZURE_ACCOUNT_KEY
    })

movies.to_csv(
    "az://objects/movies_az2.csv",
    storage_options={
        'account_name': "tscookbook",
        'account_key': AZURE_ACCOUNT_KEY
    })
```

How it works...

In the preceding code section, we used the `DataFrame.to_csv()` and `DataFrame.to_excel()` methods to write to Amazon S3, Azure Blob Storage, and Google Cloud Storage. The `storage_options` parameter allows passing a key-value pair containing the information required for the storage connection; for example, AWS S3 requires passing `key` and `secret`, GCP requires `token`, and Azure requires `account_key`.

Several of the pandas writer functions support writing directly to a remote or cloud storage filesystem through the `storage_options` parameter. These include the following:

- `Pandas.DataFrame.to_excel()`
- `Pandas.DataFrame.to_json()`
- `Pandas.DataFrame.to_parquet()`
- `Pandas.DataFrame.to_pickle()`
- `Pandas.DataFrame.to_markdown()`
- `Pandas.DataFrame.to_stata()`
- `Pandas.DataFrame.to_xml()`

This is possible because pandas leverages **fsspec** (filesystem specification), a unified interface for working with different file systems, including cloud storage.

There's more...

For more granular control, you can use the specific Python SDK for AWS (`boto3`), Google Cloud (`google-cloud-storage`), or Azure (`azure-storage-blob`) to write your data.

First, let's store our movie DataFrame in CSV format for uploading the data to the various cloud storage services.

```
data = movies.to_csv(encoding='utf-8', index=True)
```

Note that we set `index=True` (which is the default) because our Date column is the DataFrame index and we need to ensure it gets written out as a column in the CSV file.

Writing to Amazon S3 using the boto3 library

You will explore both the **Resource API** and **Client API**. The Resource API is a higher-level abstraction that simplifies the code and interactions with AWS services. At the same time, the Client API provides a low-level abstraction, allowing for more granular control over AWS services.

When using the resource API with `boto3.resource("s3")`, you will first need to create an Object resource using the `Object` method by supplying the S3 bucket name and an Object key (file name). Once defined, you will have access to several methods, including `copy`, `delete`, `put`, `download_file`, `load`, `get`, and `upload`, to name a few. The `put` method will add an object to the S3 bucket defined.

When using the client API with `boto3.client("s3")`, you have access to many methods at the Bucket and Object levels, including `create_bucket`, `delete_bucket`, `download_file`, `put_object`, `delete_object`, `get_bucket_lifecycle`, `get_bucket_location`, `list_buckets`, and many more. The `put_object` method will add an object to the S3 bucket defined.

```
import boto3
bucket = "tscookbook-private"

# Using the Resource API
s3_resource = boto3.resource(
```

```

    "s3",
    aws_access_key_id = AWS_ACCESS_KEY,
    aws_secret_access_key = AWS_SECRET_KEY)
s3_resource.Object(bucket, 'movies_boto3_resourceapi.csv').put(Body=data)

# Using the Client API
s3_client = boto3.client(
    "s3",
    aws_access_key_id = AWS_ACCESS_KEY,
    aws_secret_access_key = AWS_SECRET_KEY)
s3_client.put_object(Body=data, Bucket=bucket,
                    Key='movies_boto3_clientapi.csv')

```

Writing to Google Cloud Storage using the google-cloud-storage library

You will first need to create a client object, an instance of the `Client` class from the storage module. You will authenticate using the service account JSON key file. This is specified using the `from_service_account_json` method. You will use the `bucket` and `blob` methods to create a reference to the blob object you want to place in the `tscookbook` bucket in Google Storage. Finally, you can upload the data, using the `upload_from_string` method, into the blob object specified.

```

from google.cloud import storage
# Authenticate using the service account key
storage_client = storage.Client.from_service_account_json(GCP_KEY_FILE)

bucket_name = 'tscookbook'
file_path = 'movies_gsapi.csv'

blob = storage_client.bucket(bucket_name).blob(file_path)

blob.upload_from_string(data)

```

Writing to Azure Blob Storage using the azure-storage-blob library

You will start by creating a `BlobServiceClient` object and authenticating using your Azure storage account API key. You then create the blob object for the specified container using `get_blob_client` and upload that data into the specified object using the `upload_blob` method.

```

from azure.storage.blob import BlobServiceClient
blob_service_client = BlobServiceClient(
    account_url="https://tscookbook.blob.core.windows.net",
    credential=AZURE_ACCOUNT_KEY
)

```

```
blob_client = blob_service_client.get_blob_client(  
    container='objects',  
    blob='movies_blobapi.csv'  
)  
blob_client.upload_blob(data)
```

See also

To learn more about managing cloud storage with Python, explore the official documentation for these popular libraries:

- Amazon S3 (Boto3): <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html>
- Azure Blob Storage: <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python>
- Google Cloud Storage: <https://cloud.google.com/python/docs/reference/storage/latest>

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



4

Persisting Time Series Data to Databases

After completing a data analysis task, where data is extracted, processed, transformed, and possibly modeled, you'll typically need to store the results somewhere. While you can store the data in a flat file such as a CSV, handling large amounts of corporate time series data (including sensitive or critical business data) requires a more robust and secure storage solution. Databases offer several advantages:

- **Security** with encryption at rest, protecting your data from unauthorized access, even if physical storage is compromised
- **Concurrency**, allowing multiple users to query the database without impacting performance
- **ACID** compliance, guaranteeing that transactions are processed reliably with atomicity, consistency, isolation, and durability
- **Optimized** read and write performance for large datasets through indexing, caching, and query optimization
- **Scalability** to handle growing data volumes by adding more resources horizontally or vertically
- **Distributed** computing capabilities that spread processing and storage across multiple machines

In a corporate context, once data is stored in a database, it can be shared across various departments. For example, finance, marketing, sales, and product development teams can access the data for their specific needs. Additionally, the data can be democratized and utilized for numerous use cases by different organizational roles, such as business analysts, data scientists, data engineers, marketing analysts, and business intelligence developers.

In this chapter, you will learn how to write your time series data to a database system for persistence. You will explore different types of databases (relational and non-relational) and use **Python** to push your data.

Specifically, you will use the `pandas` library, as much of your analysis will involve `pandas` `DataFrames`. You will learn how to use `pandas` to persist your time series `DataFrame` to a database storage system. Many databases offer Python APIs and connectors, and many of them support `pandas` `DataFrames` (for reading and writing) due to their popularity and widespread adoption. This chapter aims to give you first-hand experience working with different methods to connect to these database systems to persist your time series `DataFrame`. You will be working with select representatives of the common database types explored in *Chapter 2*, each offering different strengths for time series data management:

- **PostgreSQL** (relational database): Excellent for structured data with strong consistency and guarantees
- **MongoDB** (document database): Flexible schema for evolving time series data structures
- **InfluxDB** (time series database): Optimized specifically for timestamp data and time-based queries
- **Snowflake** and **Amazon Redshift** (cloud data warehouse): Scalable analytics on large volumes of historical time series data

The following is a list of the recipes that we will cover in this chapter:

- Writing time series data to a relational database
- Writing time series data to MongoDB
- Writing time series data to Snowflake



Don't miss the GitHub-exclusive recipe—*Chapter 4-1: Writing time series data to InfluxDB*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Writing to a database and permissions

When you install your database instance or use your cloud service, writing your data is straightforward since you have owner/admin privileges.



However, in any corporate environment, you must collaborate with database owners, maintainers, and possibly IT, database admins, or cloud admins. Typically, they will grant you permissions to write your data in a sandbox or a development environment. Once you are done, another team (such as DataOps or DevOps) may inspect the code and evaluate performance before migrating it to a quality assurance (QA)/user acceptance testing (UAT) environment. At this stage, the business may test and validate the data for approval. Finally, the data may be promoted to the production environment for general use. This process ensures that only validated, secure, and performant data pipelines make it to production, protecting both data quality and system stability.

Technical requirements

In this chapter, we will be using `pandas` 2.3.3 (released September 29, 2025) extensively.

Throughout our journey, you will install several Python libraries to work with pandas. These are highlighted in the *Getting ready* section for each recipe. You can also download the Jupyter notebooks from this book's GitHub repository (<https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>) to follow along.

You should refer to the *Technical requirements* section in *Chapter 2*. This includes creating a configuration file, such as `database.cfg`, to store your credentials and connection information to the various databases.

You will be using the same dataset throughout the recipes in this chapter. The dataset is based on Amazon's stock data from January 2019 to December 2023, pulled using the `yfinance` library and written as a pandas `DataFrame`.

You can also install using `pip` with the following:

```
pip install yfinance
```

Before diving deeper, let's prepare a simple helper function to fetch and clean stock data using the `yfinance` library. This function retrieves price data for any ticker between a start and optional end date, then returns a formatted `DataFrame` with the most relevant columns.

```
import yfinance as yf
import pandas as pd

def get_stock_data(ticker, start, end=None):
    """Fetch stock data and return cleaned DataFrame"""
    stock = yf.Ticker(ticker)
    data = stock.history(start=start, end=end)
    hist.index = hist.index.tz_localize(None).normalize()
    return data[['Open', 'High', 'Low', 'Close', 'Volume']]
```

For example, to get Microsoft's stock data from January 1, 2024, to the latest available date:

```
msft = get_stock_data('MSFT', '2024-01-01')
```

The returned `DataFrame` includes daily Open, High, Low, Close, and Volume columns with dates in a standardized format and contains all data starting from January 1, 2024, up to the most recent available trading day.

Writing time series data to a relational database

Relational databases remain the cornerstone of enterprise data storage. In this recipe, you will learn to persist your `DataFrame` to a relational database such as **PostgreSQL**. The approach is the same for any relational database system supported by the `SQLAlchemy` Python library. You will experience how `SQLAlchemy` makes switching the backend database (called *dialect*) simple without altering the code. The abstraction layer provided by the `SQLAlchemy` library makes it feasible to switch to any supported database, such as from PostgreSQL to Amazon Redshift, using the same code.

The sample list of supported relational databases (dialects) in SQLAlchemy includes the following:

- Microsoft SQL Server
- MySQL/MariaDB
- PostgreSQL
- Oracle
- SQLite

Additionally, external dialects are available to install and use with **SQLAlchemy** to support other databases (dialects), such as Snowflake, Microsoft SQL Server, and Google BigQuery. Please visit the official page of SQLAlchemy for a list of available dialects: <https://docs.sqlalchemy.org/en/20/dialects/>.

Getting ready



Refer to the *Reading data from a relational database* recipe in *Chapter 2*, on how to set up your PostgreSQL instance and the different ways to connect to PostgreSQL.

In this recipe, you will use the `yfinance` Python library to pull stock data and **SQLAlchemy** with appropriate database drivers to write the data.

To install the libraries using `pip`, run the following:

```
>> pip install sqlalchemy
>> pip install psycpg
```

This should install `psycpg3`.

If you ran into an issue when importing `psycpg`, run the following command:

```
pip install "psycpg[binary]"
```

The Jupyter notebook for this chapter is provided in the GitHub repository of this book, which you can find here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>.

How to do it...

In this recipe, you will be pulling Amazon's stock data from January 2019 to December 2023 using the `yfinance` library into a pandas `DataFrame`, and then writing the `DataFrame` to a table in a PostgreSQL database:

1. Start by calling the `get_stock_data` function created in the *Technical requirements* section:

```
amzn_hist = get_stock_data('AMZN', '2019-01-01', '2023-12-31')
```

2. You will need to create a SQLAlchemy engine object. The **engine** informs SQLAlchemy and pandas which *dialect* (backend database) we plan to interact with, and connection details for the running database instance. Utilize the `URL.create()` method to create a properly formatted URL object by providing the necessary parameters (drivername, username, password, host, port, and database). These parameters are better stored in a configuration file. You can utilize the same `database.cfg` file created in *Chapter 2*, or you can create a new one with the following content:

```
[POSTGRESQL]
host= 127.0.0.1
database= postgres
username= your_username
password= your_password
port=5432
```

Load the parameter values for the database connection using `config.read()`:

```
from sqlalchemy import create_engine, URL
from configparser import ConfigParser

config = ConfigParser()
config.read('database.cfg')
config.sections()
params = dict(config['POSTGRESQL'])

url = URL.create('postgresql+psycopg', **params)
print(url)

>>
postgresql+psycopg://postgres:***@127.0.0.1:5432/postgres
```

3. You can now pass the `url` object to `create_engine`:

```
engine = create_engine(url)
print(engine)

>>
Engine(postgresql+psycopg://postgres:***@127.0.0.1:5432/postgres)
```

4. Write the `amz_hist` DataFrame to a new `amzn` table in your PostgreSQL database instance. This is achieved using the `DataFrame.to_sql()` writer function, which leverages SQLAlchemy's capabilities to convert the DataFrame into the appropriate table schema and translate the data into the appropriate SQL statements, such as `CREATE TABLE` and `INSERT INTO`, specific to the dialect (backend database). If the table does not exist, a new table is created before loading the data; otherwise, if the table exists, then you will need to provide instructions on how it should be handled. This is done through the `if_exists` parameter, which accepts either `'fail'`, `'replace'`, or `'append'`:

```
amzn_hist.to_sql('amzn',
                engine,
                index=True,
                index_label='Date',
                if_exists='replace')
```

An alternative method to accomplish the same tasks as the preceding code is by utilizing the `with` clause, which automatically manages the connection for you. This is generally a preferred approach for better resource management:

```
with engine.connect() as connection:
    amzn_hist.to_sql('amzn',
                    connection,
                    index=True,
                    index_label='Date',
                    if_exists='replace')
```

Once the preceding code is executed, a new `amzn` table is created under the public schema in the default postgres database (default).

You can validate this by running the following against the database:

```
from sqlalchemy import text

query = """
SELECT EXISTS (
    SELECT FROM information_schema.tables
    WHERE table_schema = 'public'
    AND table_name = 'amzn'
);"""

with engine.connect() as conn:
    result = conn.execute(text(query))
    print(result.fetchone())

>>
(True,)
```

Notice the use of the `text()` function, which wraps the SQL string so that SQLAlchemy can execute it directly. This is helpful when you want to run custom SQL queries. The `text()` function creates a `TextClause` object that represents the SQL statement.

5. Check how the data types from the pandas DataFrame got translated into the appropriate data types in PostgreSQL. This is an important verification step because SQLAlchemy automatically maps pandas data types to their equivalent database types:

```
query = """
SELECT column_name, data_type
FROM information_schema.columns
```

```

        WHERE table_name = 'amzn';
    """
    with engine.connect() as conn:
        result = conn.execute(text(query))
        for row in result:
            print(row)

>>
('Date', 'timestamp without time zone')
('Open', 'double precision')
('High', 'double precision')
('Low', 'double precision')
('Close', 'double precision')
('Volume', 'bigint')

```

Notice how pandas' `DatetimeIndex` became a PostgreSQL timestamp, floating-point numbers became double precision, and integer values were stored as `bigint`. This automatic type mapping is one of the benefits of using SQLAlchemy with pandas.

6. Confirm that the data was written to the database by querying the `amzn` table and counting the number of records:

```

query = "select count(*) from amzn;"

with engine.connect() as conn:
    result = conn.execute(text(query))
    result.fetchone()

>>
(1258,)

```

7. Next, request additional Amazon stock prices using `get_stock_data`—this time, the 2024 data (for example, January 1, 2024, to December 31, 2024)—and append it to the existing `amzn` table. Here, you will take advantage of the `if_exists` parameter in the `to_sql()` writer function:

```

amzn_hist_2024 = get_stock_data('AMZN', '2024-01-01', '2024-12-31')
print(amzn_hist_2024.shape)

>>
(251, 5)

```

Make sure to pass `append` to the `if_exists` parameter, as shown in the following code:

```

with engine.connect() as connection:
    amzn_hist_2024.to_sql('amzn',
                          connection,
                          index=True,
                          index_label='Date',
                          if_exists='append')

```

- Count the total number of records to ensure that you have appended all 251 to the original 1,258 records. You will run the same query that was executed earlier, as shown in the following code:

```
query = "select count(*) from amzn;"

with engine.connect() as conn:
    result = conn.execute(text(query))
    print(result.fetchone())

>>
(1509,)
```

Indeed, you can observe that all 1,509 records were written to the `amzn` table.

How it works...

The `DataFrame.to_sql()` writer function leverages SQLAlchemy to handle many complex tasks under the hood, including creating the table schema, converting data types, inserting records, and committing changes to the database, without having to write a single SQL statement.

When working with pandas and SQLAlchemy to read and write to a relational database, we follow a similar workflow (as discussed in *Chapter 2* in the *Reading data from a relational database* recipe). The same engine object and connection patterns apply, so once you're comfortable with one, the other comes naturally.

We always start by creating an **engine** with `create_engine` and specifying the **dialect** (backend database). When you call the `to_sql()` method, pandas maps your `DataFrame`'s data types to the appropriate matching SQL data types for your chosen database (e.g., PostgreSQL). This is one advantage of using an **object relational mapper (ORM)** such as SQLAlchemy; it gives you an abstraction layer, so you do not have to worry about *converting* the `DataFrame` schema into a specific database schema:

- pandas `DatetimeIndex` → PostgreSQL `timestamp without time zone`
- pandas `float64` → PostgreSQL `double precision`
- pandas `int64` → PostgreSQL `bigint`

In our example, you used the `if_exists` parameter in the `to_sql()` function with two different arguments:

- Initially, you set the value to `if_exists='replace'`, which would overwrite the table if it existed. Under the hood, this is like running `DROP TABLE` followed by `CREATE TABLE`. This is useful for resetting data, but it can be dangerous if you have existing data you want to keep. Because of this concern, the default value is `if_exists='fail'`, which would throw an error if the table existed in the database.
- In the second part of the recipe, the plan was to insert additional records (rows) into the existing table without modifying its structure, so we switched to `if_exists='append'`.

When you pull stock data using `yfinance`, the `Date` field is automatically set as the `DataFrame`'s index (`DatetimeIndex`). In other words, there was no `Date` column because it was set as an index. By default, `to_sql()` writes the `DataFrame`'s index as a regular table column in your SQL database, and is controlled by the `index` parameter (which is `True` by default). If your `DataFrame`'s index wasn't meaningful, you could set `index=False` to exclude it from the database table. We also used `index_label='Date'` to explicitly name that column in the SQL table.

Another handy parameter that can be extremely useful is `chunksize`. The default value is `None`, which writes all the rows in the `DataFrame` at once. But if your dataset is extremely large, you can use the `chunksize` parameter to write to the database in batches; for example, `chunksize=500` would write to the database in batches of 500 rows at a time:

```
with engine.connect() as connection:
    amzn_hist.to_sql('amzn',
                    connection,
                    chunksize=500,
                    if_exists='append')
```

In summary, `pandas` and `SQLAlchemy` make it straightforward to persist time series data to a relational database, handling the heavy lifting so you can focus on your analysis—not the plumbing.

There's more...

When using the `pandas` `read_sql`, `read_sql_table`, `read_sql_query`, and `to_sql` I/O functions, they expect a `SQLAlchemy` connection object (`SQLAlchemy` engine). To use `SQLAlchemy` to connect to a database of choice, you need to install the appropriate Python DBAPI (driver) for that specific database (for example, Amazon Redshift, Google BigQuery, MySQL, MariaDB, PostgreSQL, or Oracle). This gives you the advantage of writing your script once and having it work with other dialects (backend databases) supported by `SQLAlchemy`. To demonstrate this, let's extend the last example.

Amazon Redshift, a popular cloud data warehouse database, is based on PostgreSQL at its core, and it offers several enhancements, including columnar storage for fast analytical queries. You will explore the simplicity of `SQLAlchemy`, as well as other options for writing a `pandas` `DataFrame` to Amazon Redshift.

Writing to Amazon Redshift with SQLAlchemy

You will use the same code, but write to an Amazon Redshift database this time. The only requirement is installing a Python DBAPI (driver) for Amazon Redshift. Note that `sqlalchemy-redshift` requires `psycopg2`.

To install using `pip`, run the following command:

```
pip install sqlalchemy-redshift psycopg2-binary
```




Installing **sqlalchemy-redshift** will likely downgrade your version of the installed SQLAlchemy from 2.0.40 (or a newer version) to an older version (e.g., 1.4.x or earlier). If you want to upgrade to the latest SQLAlchemy version, you can run the following command:

```
pip install sqlalchemy --force
```

If you do not have an Amazon Redshift cluster, you can follow these steps: <https://docs.aws.amazon.com/redshift/latest/mgmt/create-cluster.html>. The following is an example configuration screen to set up an Amazon Redshift cluster. I kept the remaining configurations with their defaults.

Create cluster [Info](#)

Cluster configuration

Cluster identifier

This is the unique key that identifies a cluster.

The identifier must be from 1-63 characters. Valid characters are a-z (lowercase only) and - (hyphen).

Choose the size of the cluster

- ☒ I'll choose
- ☐ Help me choose

Node type [Info](#)

Choose a node type that meets your CPU, RAM, storage capacity, and drive type requirements.

AZ configuration [Info](#)

Choose if you want to deploy the Redshift cluster in another Availability Zone.

- ☒ **Single-AZ**
Compute resources are deployed in a single Availability Zone. The cluster is default to use the **Current track**

- ☐ **Multi-AZ - new**
Compute resources are deployed in two Availability Zones. The cluster is default to use the **Trailing track**

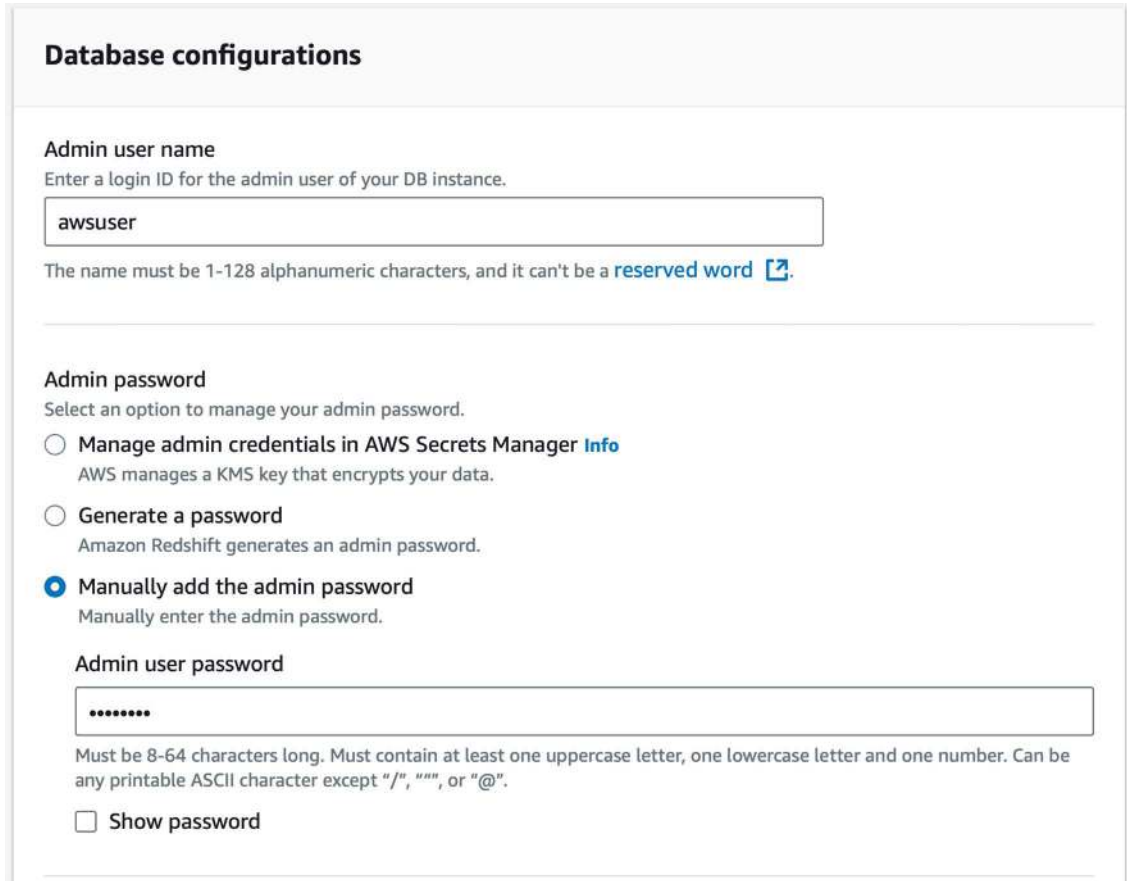
Number of nodes

Enter the number of nodes that you need.

Range (1-16)

Figure 4.1: Amazon Redshift cluster configuration section

Note: the default Admin username is **awsuser**. To simplify the process, click **Manually** add the admin password and type in a password of your choice. You will use this same password for connecting to the Amazon Redshift cluster.



The screenshot shows the 'Database configurations' section of the Amazon Redshift console. It is divided into two main parts: 'Admin user name' and 'Admin password'.

Admin user name
Enter a login ID for the admin user of your DB instance.

The name must be 1-128 alphanumeric characters, and it can't be a [reserved word](#) .

Admin password
Select an option to manage your admin password.

☐ **Manage admin credentials in AWS Secrets Manager** [Info](#)
AWS manages a KMS key that encrypts your data.

☐ **Generate a password**
Amazon Redshift generates an admin password.

☒ **Manually add the admin password**
Manually enter the admin password.

Admin user password

Must be 8-64 characters long. Must contain at least one uppercase letter, one lowercase letter and one number. Can be any printable ASCII character except `"/"`, `"'"`, or `"@"`.

☐ **Show password**

Figure 4.2: Amazon Redshift database configuration section

Once the configuration is complete, click the **Create cluster** button. Wait a few minutes until the cluster status changes to Available. The cluster endpoint includes the host URL, port (5439), and default database (dev). You must also enable external access and whitelist your IP address: in the Properties tab, scroll to the Network and security settings section, click Edit, turn on public access, and then click Save changes, noting that it may take a few minutes for the update to apply.

If needed, you can edit the security group(s) attached to your Redshift cluster to allow inbound traffic on port 5439 (the default Redshift port) from your IP address or desired IP range.

You will use the same code used in the PostgreSQL section; the only difference is the SQLAlchemy engine, which uses **Amazon Redshift DBAPI**. Start by loading the connection parameters from your configuration file. In this example, the configurations are stored in a `database.cfg` file:

```
[AWS]
host=yourHostURL
port=5439
database=dev
username=yourusername
password=yourpassword
```

Use `ConfigParser` and `URL` to extract the parameters and construct the URL:

```
from configparser import ConfigParser
config = ConfigParser()
config.read('database.cfg')
config.sections()
params = dict(config['AWS'])

from sqlalchemy import URL, create_engine

url = URL.create('redshift+psycopg2', **params)
print(url)
>>
redshift+psycopg2://awsuser:***@redshift-cluster-1.***.us-east-1.redshift.
amazonaws.com:5439/dev
```

You can now create the engine using the following:

```
aws_engine = create_engine(url)
```

Using the `yfinance` library, create a new `amzn_hist` `DataFrame` based on the past **five years** of stock data:

```
amzn = yf.Ticker("AMZN")
amzn_hist = amzn.history(period="5y")
amzn_hist = amzn_hist[['Open', 'High', 'Low', 'Close', 'Volume']]
```

Before writing our `DataFrame`, we need to reset the index. This will give us back our `Date` column. We do this because Amazon Redshift does not support traditional indexes since it is a columnar database (instead, you can define a **sort key**):

```
amzn_hist = amzn_hist.reset_index()
with aws_engine.connect() as conn:
    # Manually drop table using Redshift-compatible syntax
    conn.execute(text("DROP TABLE IF EXISTS public.amzn CASCADE"))
    conn.commit()
```

```
# Now create fresh table
amzn_hist.to_sql('amzn',
                 con=conn,
                 schema='public',
                 index=False,
                 if_exists='replace')
```

Using `if_exists='replace'` will check whether the table exists before creating a new one. If it exists, you may run into issues due to compatibility between `sqlalchemy-redshift` and `SQLAlchemy` (latest version). A good workaround is to explicitly execute the SQL `DROP TABLE IF EXISTS public.amzn CASCADE`, as shown in the code example. Alternatively, if you plan to append to the existing table, then you do not need that line and should instead use `if_exists='append'`.

Notice `index=False` in the preceding code. This is done because `to_sql` will write the index object in a `DataFrame` since the default is `index=True`. When you reset the `DataFrame` index, it moves the `DatetimeIndex` to a **Date** column and replaces the index with a `RangeIndex` (ranging from 0 to 1,257). Using `index=False` ensures that we do not attempt to write the `RangeIndex` in Amazon Redshift.

Finally, you can validate the total number of records written:

```
from sqlalchemy import text

query = "select count(*) from amzn;"
with aws_engine.connect() as conn:
    result = conn.execute(text(query))
    result.fetchone()
```

Writing to Amazon Redshift using `redshift_connector`

In this example, you will utilize a different library, `redshift_connector`, which is the official Amazon Redshift connector for Python and easily integrates with `pandas`. Before we start, you will need to install the library.

You can also install it with `pip` using the following:

```
pip install redshift_connector
```

Note: `redshift_connector` expects a `user` parameter, unlike `SQLAlchemy`, which accepts a `username` parameter. For this, you can create a new section in your configuration file. An example is shown here:

```
[AWS2]
host=yourendpoint
port=5439
database=dev
user=username
password=password
```

The following code reads the parameters from a `database.cfg` file, and passes these parameters to `redshift_connector.connect()` to create a connection object:

```
import redshift_connector

from configparser import ConfigParser
config = ConfigParser()
config.read('database.cfg')
config.sections()
params2 = dict(config['AWS2'])

conn = redshift_connector.connect(**params2)
```

You will create a cursor object that gives access to the `write_dataframe` method:

```
cursor = conn.cursor()
cursor.write_dataframe(amzn_hist, 'amzn')
```

Finally, you will commit the transaction:

```
conn.commit()
conn.close()
```

Note that `write_dataframe` in the `redshift_connector` library only **appends** to an existing table. It does not support modes such as *overwrite* or *upsert*. The previous code assumes that the `amzn` table is created from the previous section (*Writing to Amazon Redshift with SQLAlchemy*). Therefore, make sure the table exists in Redshift before using this method. You can create the table schema using `SQLAlchemy` or manually from your Redshift console. If you need to *overwrite* or *upsert* data, consider using the `awswrangler` library (introduced in the next section), which provides more flexibility.

Writing to Amazon Redshift with AWS SDK for pandas

The `awswrangler` library or **AWS SDK for pandas** simplifies integrating with various AWS services such as Athena, Glue, Redshift, Neptune, DynamoDB, EMR, S3, and more.

You can also install it using `pip`:

```
pip install 'awswrangler[redshift]'
```

You can utilize the `conn` object created in the previous section, *Writing to Amazon Redshift using redshift_connector*:

```
import awswrangler as wr

conn = redshift_connector.connect(**params2)

wr.redshift.to_sql(
    df=amzn_hist,
```

```
table='amzn',
schema='public',
con=conn,
mode='overwrite'
)
```

The mode parameter supports three options: `overwrite` replaces the existing table, `append` adds new records to the existing table, and `upsert` updates existing records and inserts new ones.

Each method for writing time series data to Amazon Redshift has its advantages. **SQLAlchemy** offers flexibility across many database systems, **redshift_connector** provides the official integration with optimal performance, and **awswrangler** simplifies working with the broader AWS ecosystem. The choice depends on your specific requirements.

See also

Here are some additional resources:

- To learn more about the `DataFrame.to_sql()` function, you can visit https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html
- To learn more about **SQLAlchemy** features, you can start by reading their features page: <https://www.sqlalchemy.org/features.html>
- To learn about **awswrangler**, you can visit their GitHub repo here: <https://github.com/aws/aws-sdk-pandas>

Writing time series data to MongoDB

MongoDB is a document database system that stores data in **BSON** format. When you query data from MongoDB, the data will be represented in JSON format. BSON is similar to JSON; it is the binary encoding of JSON. Unlike JSON, though, it is not in a human-readable format. JSON is great for transmitting data and is system-agnostic. BSON is designed to store data and is associated with MongoDB.

In this recipe, you will explore writing a pandas `DataFrame` to MongoDB.

Getting ready



Refer to the *Reading data from MongoDB in Chapter 2*, on how to set up your MongoDB instance and the different ways to connect to MongoDB.

In the *Reading data from MongoDB in Chapter 2*, we installed `pymongo`. For this recipe, you will be using that same library again.

To install using `pip`, run the following:

```
$ python -m pip install pymongo
```

How to do it...

To store data in MongoDB, you will create a **database** and a **collection**. A database contains one or more collections, which are like tables in relational databases. Once a collection is created, you will write your data as documents. A collection contains documents, which are like rows in relational databases, but in JSON-like format. Follow these steps:

1. Start by importing the necessary libraries:

```
import pandas as pd
from pymongo import MongoClient
```

2. Create a MongoClient instance to establish a connection to the database:

```
client = MongoClient('mongodb://localhost:27017')
```

3. Create a new database named `stock_data` and a collection for our time series data. MongoDB offers two approaches for collections. Let's look at both.

First, this is the traditional way to create a regular collection:

```
db = client['stock_data']
collection = db.create_collection("daily_data")
```

This will create a new database called `stock_data` and a collection named `daily_data`. If `stock_data` already exists, it will add the `daily_data` collection to the existing database. If you're following along from *Chapter 2*, you should already have a `stock_data` database and a collection named `microsoft`. Note: using `create_collection()` assumes that the collection does not already exist. If it did, it would throw an error.

However, since we are working with time series data, MongoDB offers a more efficient way to store and query our data by creating a **time series collection**. Starting with MongoDB version 5.0, time series collections are optimized for timestamped data:

```
db = client['stock_data']

ts = db.create_collection(
    name="daily_data",
    timeseries={
        "timeField": "Date",
        "metaField": "symbol",
        "granularity": "hours"
    }
)
```

With this **time series collection**, you get improved storage efficiency and query performance specifically designed for time-based data like our stock prices.

We'll use the `ts` variable to interact with this time series collection.

4. You will use the `get_stock_data()` function created in the *Technical requirements* section to pull historical Amazon stock data from January 1, 2019, to December 31, 2024:

```
amzn_hist = get_stock_data('AMZN', '2019-01-01', '2024-12-31')
```

5. In pandas, we work with data in a tabular format, where each column represents a variable, and each row represents a data point. MongoDB, however, stores data as documents in a JSON-like format (BSON), where each document is an independent record that can include a timestamp, metadata, and other key-value pairs.

Before inserting data into MongoDB, you need to transform the DataFrame into a list of dictionaries, where each dictionary (or document) represents a stock data point. Each dictionary will include a timestamp (Date), stock information (e.g., High, Low), and ticker identifier (e.g., `"symbol": "AMZN"`)

You will explore two methods: the first option utilizes the `to_dict()` method, and the second option iterates over the DataFrame.

Let's explore the first option:

```
amzn_hist['symbol'] = "AMZN"
amzn_hist = amzn_hist.reset_index()
amzn_hist['Date'] = pd.to_datetime(amzn_hist['Date'])
amzn_records = amzn_hist.to_dict(orient='records')
amzn_records[0:2]

>>
[{'Date': Timestamp('2019-01-02 00:00:00'),
  'Open': 73.26000213623047,
  'High': 77.66799926757812,
  'Low': 73.04650115966797,
  'Close': 76.95649719238281,
  'Volume': 159662000,
  'symbol': 'AMZN'},
 {'Date': Timestamp('2019-01-03 00:00:00'),
  'Open': 76.00050354003906,
  'High': 76.9000015258789,
  'Low': 74.85549926757812,
  'Close': 75.01399993896484,
  'Volume': 139512000,
  'symbol': 'AMZN'}]
```

The `reset_index()` is used to reset the index to bring the Date column back as a regular column—MongoDB needs this timestamp field to be explicitly present in each document.



The default value for the `orient` parameter in the `to_dict()` method is `dict`, which produces a **dictionary** that follows the `{column -> {index -> value}}` pattern.

On the other hand, using `records` as the value produces a **list** that follows the `[{column -> value}, ..., {column -> value}]` pattern.

Now, let's explore the second option, which provides more flexibility for adding specific fields or applying transformations to individual records, for example, a different ticker value:

```
amzn_hist = amzn_hist.reset_index()
amzn_records = []
for idx, row in amzn_hist.iterrows():
    doc = {
        "Date": pd.to_datetime(row['Date']),
        "symbol": "AMZN",
        "High": row['High'],
        "Low": row['Low'],
        "Close": row['Close'],
        "Open": row['Open'],
        "Volume": row['Volume']
    }
    amzn_records.append(doc)
amzn_records[0:2]
```

You now have a Python list of 1,509 dictionaries, each representing a single time-based record:

```
len(amzn_records)
>>
1509
```

6. Now, you are ready to write to the `amazon` time series collection using the `insert_many()` method:

```
result = ts.insert_many(amzn_records)
```

7. You can validate that the database and collection are created with the following code:

```
client.list_database_names()
>>
['admin', 'config', 'local', 'stock_data']

db.list_collection_names()
>>
['daily_data',
```

```
'system.buckets.daily_data',  
'microsoft',  
'system.buckets.microsoft',  
'system.views']
```

The output shows five collections:

- **daily_data**: Contains actual time series stock data for the AMZN ticker you just created in this recipe.
 - **system.buckets.daily_data**: An internal collection used by MongoDB for time series data (automatically created when using time series collections in MongoDB 5.0+).
 - **system.views**: Stores view definitions if any are present.
 - **microsoft**: Contains actual time series stock data for the MSFT ticker created in *Chapter 2*.
 - **system.buckets.microsoft**: An internal collection used by MongoDB for time series data (automatically created when using time series collections in MongoDB 5.0+). This was created in *Chapter 2*.
8. Next, pull Microsoft stock data (MSFT) and add it to the same **daily_data** time series collection. Later, you will explore how the **symbol** field can be used to distinguish between the different stock symbols (e.g., AMZN vs. MSFT) when querying the data:

```
msft_hist = get_stock_data('MSFT', '2019-01-01', '2024-12-31')  
  
msft_hist['symbol'] = "MSFT"  
msft_hist = msft_hist.reset_index()  
msft_hist['Date'] = pd.to_datetime(msft_hist['Date'])  
msft_records = msft_hist.to_dict(orient='records')  
  
result = ts.insert_many(msft_records)
```

You can check the total number of documents written by querying the database, as shown in the following code:

```
ts.count_documents({})  
>>  
3018
```

By adding both AMZN and MSFT data to the same collection but with different ticker values in the **symbol** field, you've now set up a structure that allows efficient querying by stock symbol, which we'll explore in the next step.

9. Now, the collection contains data for two stock symbols. You can use the **symbol** field to filter for each ticker in your queries. You will start by querying the **daily_data** collection to retrieve only the Microsoft (**MSFT**) stock data. This is where the **symbol** field becomes useful, allowing you to filter by a specific ticker.

Let's first define a date range and then query for MSFT data only:

```
from datetime import datetime

# Define date range
start_date = datetime(2024, 1, 1)
end_date = datetime(2024, 1, 31)

# Query for MSFT stock data within the date range
results = ts.find({
    "symbol": "MSFT",
    "Date": {"$gte": start_date, "$lte": end_date}
})

# Convert the query results to a DataFrame
msft_df = (pd.DataFrame(results)
           .set_index('Date')
           .drop(columns=['_id', 'symbol']))

print(msft_df.head())
```

	Open	Close	Low	Volume	High
Date					
2024-01-02	370.342443	367.380585	363.319154	25258600	372.363258
2024-01-03	365.538123	367.113159	365.042827	23083500	369.748136
2024-01-04	367.182505	364.478180	363.715435	20901500	369.589635
2024-01-05	365.498501	364.289978	363.051739	20987000	368.559424
2024-01-08	365.825371	371.164673	365.538121	23134000	371.669884

We dropped the `_id` field as it's MongoDB's auto-generated identifier and is not needed for our analysis.

- Next, perform an aggregation to calculate the average close price per ticker (symbol):

```
msft_avg_close = ts.aggregate([
    {
        "$group": {
            "_id": "$symbol",
            "avgClose": {
                "$avg": "$Close"
            }
        }
    }
])
```

```
] )

for doc in msft_avg_close:
    print(doc)

>>
{'_id': 'AMZN', 'avgClose': 137.1042866286574}
{'_id': 'MSFT', 'avgClose': 260.9127264913934}
```

The MongoDB `$group` operator organizes documents by the `symbol` field (prefixed with `$` to reference the field's value), and the `$avg` operator calculates the mean of all `Close` prices for each group. The `_id` field holds the grouping key (the stock ticker), while `avgClose` contains the computed average closing price across all trading days.

How it works...

PyMongo provides two insert functions to write our records as documents into a collection. These functions are as follows:

- `insert_one()` inserts one document into a collection
- `insert_many()` inserts multiple documents into a collection

In the recipe, you used `insert_many()` and passed the data to be written as documents simultaneously. This efficiently writes all our stock data records at once. However, before insertion, it was essential to transform the `DataFrame` into a format that MongoDB understands. This was accomplished by converting the `DataFrame` to a list of dictionaries using the `to_dict(orient='records')` method, which produced the following structure: `[{column -> value}, ..., {column -> value}]`.

When documents are inserted into MongoDB, each one is assigned a unique identifier stored in the `_id` field. MongoDB automatically generates one during the insert operation if the document does not already have an `_id`. You can capture the generated `_id` because the insert function returns a result object—either `InsertOneResult` for single inserts or `InsertManyResult` for bulk inserts. The following code demonstrates how this works using `insert_one` and the `InsertOneResult` class:

```
one_record = amzn_records[0]
one_record
>>
{'Date': Timestamp('2019-01-02 00:00:00'),
 'symbol': 'AMZN',
 'High': 77.66799926757812,
 'Low': 73.04650115966797,
 'Close': 76.95649719238281,
 'Open': 73.26000213623047,
 'Volume': 159662000,
 '_id': ObjectId('68075ef6beeb1cec4caea674')}
```

```
result_id = ts.insert_one(one_record)
result_id
>>
InsertOneResult(ObjectId('68075ef6beeb1cec4caea674'), acknowledged=True)
```

The returned object is an instance of `InsertOneResult`; to see the actual ID value, you can use the `inserted_id` property:

```
result_id.inserted_id
>>
ObjectId('68075ef6beeb1cec4caea674')
```

Our time series collection was created with specific **parameters** that optimize how MongoDB stores and retrieves the time-based data:

- "timeField" tells MongoDB which field contains the timestamp
- "metaField" identifies which field contains **metadata** for grouping ("symbol" in our documents)
- "granularity" indicates the expected frequency of our data points ("hours" in our example)

The **granularity** parameter controls how MongoDB groups your time series data into buckets for storage optimization. This is particularly useful when working with high-frequency data. For daily stock data spanning multiple years, "hours" is appropriate as it allows each bucket to contain up to 30 days of data. If you have ticker data in minutes or seconds, you might choose "minutes" (for buckets up to 24 hours) or "seconds" (for buckets up to 1 hour), respectively.

MongoDB uses these parameters to organize your time series data into optimized internal storage structures called “buckets,” which group documents with similar timestamps and the same metadata values. This is why you saw the `system.buckets.daily_data` collection created automatically alongside your `daily_data` collection.

There’s more...

When you query the database for all available collections, you will notice several appear:

```
db = client['stock_data']
db.list_collection_names()
>>
['daily_data',
 'system.buckets.daily_data',
 'microsoft',
 'system.buckets.microsoft',
 'system.views']
```

You created the `daily_data` collection, which also created the internal `system.buckets.daily_data` collection. This is due to MongoDB’s **bucket pattern**—a data modeling technique that optimizes how time series data is stored and queried.

By default, when you insert your DataFrame as a list of dictionaries, you are essentially inserting each record (row) as a separate MongoDB document. This creates a one-to-one mapping between records and documents. However, MongoDB's bucket **pattern** groups related data points into a single document, reducing the total number of documents, improving performance, and simplifying queries.

For example, if you have hourly data, you can group it into a bucket, such as a 24-hour period, and store all the data for that range in one document. Similarly, if we have sensor data from multiple devices, you can use the bucket pattern to group the data (for example, by device ID and by a time range) and insert it as a single document. This approach is especially powerful for high-volume time series data, as it reduces storage overhead and speeds up queries.

When you create a time series collection, MongoDB automatically applies the bucket pattern to store the data in an efficient format. Let's break it down:

- `daily_data`: This is the main time series collection you created. It acts as a logical view that allows you to interact with your time series data using standard MongoDB operations.
- `system.buckets.daily_data`: This is an internal collection where MongoDB stores the actual time series data using the bucket pattern. MongoDB automatically implements this strategy for time series collections to improve storage and query performance. Here's how it works:
 - Documents are grouped into "buckets" based on timestamps and metadata fields (e.g., the stock symbol)
 - Each bucket contains data points close together in time and shares the same metadata values
 - This bucketing strategy significantly reduces the number of documents stored, improving query efficiency and reducing disk usage
- `system.views`: This is a system collection that MongoDB uses to store information about all views in the database, including the view for your time series collection.

To better understand how the bucket pattern is applied, let's explore the technique by creating a new collection (a regular collection, not a time series collection) and bucketing the daily stock data by year and month:

```
db = client['stock_data']  
bucket = db.create_collection(name='stock_bucket')
```



Note: if `stock_bucket` exists, then the `create_collection()` function will throw an error (`CollectionInvalid: collection stock_bucket already exists`). An alternative to this is using the following code:

```
# If stock_bucket already exists, use  
bucket = db["stock_bucket"]
```

This will implicitly create a `stock_bucket` collection if it doesn't exist, unlike `create_collection`, which explicitly creates a collection immediately, whether you insert documents into it or not. The implicit approach does not actually create the collection until you insert the first document.

Next, let's create a new DataFrame and add two additional columns—**month** and **year**:

```
amzn_hist = get_stock_data('AMZN', start='2019-01-01', end='2025-03-31')
amzn_hist = amzn_hist[['Open',
                       'High',
                       'Low',
                       'Close',
                       'Volume']].reset_index()

amzn_hist['month'] = amzn_hist['Date'].dt.month
amzn_hist['year'] = amzn_hist['Date'].dt.year
```

In the preceding code, you added the month and year columns to the DataFrame and initiated a new collection named `stock_bucket`.

We can check how many unique month/year combinations we'll create:

```
unique_combinations = amzn_hist.groupby(['year', 'month'])
print(f"Buckets to create: {len(unique_combinations)}")
>>
Buckets to create: 75
```

In the next code segment, you will loop through the data and write your groups (by year and month) as a single document:

```
# Group by year and month, then process each group
for (year, month), group_data in amzn_hist.groupby(['year', 'month']):
    record = {
        'month': int(month),
        'year': int(year),
        'symbol': 'AMZN',
        'price': [float(price) for price in group_data['Close'].values]
    }
    bucket.insert_one(record)
```

Notice how we're storing all closing prices for each month as an array within a single document. This demonstrates the core of the bucket pattern.

In the previous code, you loop through the unique year and month combinations. For each combination, you create a record dictionary containing the month, year, symbol, and a list of Close prices. This record is then inserted into the `stock_bucket` collection, effectively bucketing the data by month and year.

To illustrate the difference in the number of documents, run the following code:

```
print('without bucketing: ',
      db.daily_data.count_documents({}))
print('with bucketing: ',
```

```
db.stock_bucket.count_documents({})  
  
>>  
without bucketing: 3019  
with bucketing: 75
```

Notice that we now have 75 documents containing the same data that previously required 3,019 documents—a 97.5% reduction. Each bucketed document contains all daily prices for its month, making historical analysis more efficient.

To query the database for January 2025 and see how the document is represented, use the following code example:

```
results = pd.DataFrame(bucket.find({'year':2025, 'month': 1}))  
results['price'].to_dict()[0]
```

You can also execute the same query using MongoDB Compass.

Notice how one document was returned but contains 20 data points (array).

See also

- For more information on storing time series data and bucketing in MongoDB, refer to this MongoDB blog post: <https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2-schema-design-best-practices>
- To learn more about the MongoDB bucket pattern, check the official documentation: <https://www.mongodb.com/docs/manual/data-modeling/design-patterns/group-data/bucket-pattern/>
- For detailed information about MongoDB's **time series collections**, visit the official manual: <https://www.mongodb.com/docs/manual/core/timeseries-collections/>

Writing time series data to Snowflake

Snowflake has become a very popular cloud database option for building big data analytics, due to its scalability, performance, and being SQL-oriented (a columnar-stored relational database).

Snowflake's connector for Python simplifies the interaction with the database, whether it's for reading or writing data, or, more specifically, the built-in support for **pandas** DataFrames.

Getting ready



Refer to the *Reading data from a Snowflake* recipe in Chapter 2, as a refresher on the different ways to connect to Snowflake.

The recommended approach for the `snowflake-connector-python` library is to install it using `pip`, allowing you to install *extras* such as `pandas`, as shown:

```
pip install snowflake-sqlalchemy # SQLAlchemy integration
pip install snowflake-snowpark-python # Snowpark DataFrame API
pip install "snowflake-connector-python[pandas]" #includes pandas support
```



Note that Python 3.13 is not yet supported, and you should check Snowflake's documentation for the most up-to-date compatibility information. When installing Snowpark, it's recommended to create a Python virtual environment (<https://docs.snowflake.com/en/developer-guide/snowpark/python/setup>).

Create a configuration (for example, `database.cfg`) to store your Snowflake connection information, as shown:

```
[SNOWFLAKE]
ACCOUNT=<YOURACCOUNT>
USER=<YOURUSERNAME>
PASSWORD= <YOURPASSWORD>
WAREHOUSE=COMPUTE_WH
DATABASE=TSBOOK
SCHEMA=CHAPTER5
ROLE=<YOURROLE>
```

Using `ConfigParser`, extract the content under the `[SNOWFLAKE]` section to avoid exposing or hard-coding your credentials. Read parameters under the `[SNOWFLAKE]` section and convert to a Python dictionary, as shown:

```
config = ConfigParser()
config.read('database.cfg')
config.sections()
params = dict(config['SNOWFLAKE'])
```

You will use the `get_stock_data` function created in the *Technical requirements* section to pull **Amazon** stock data from January 1, 2019, to March 31, 2025:

```
amzn_hist = get_stock_data('AMZN', '2019-01-01', '2025-03-31')
```

The `amzn_hist` DataFrame does not have a **Date** column; instead, it has `DatetimeIndex`. You will need to convert the index into a column since the API does not support writing index objects:

```
amzn_hist = amzn_hist.reset_index()
amzn_hist.shape
>>
(1569, 6)
```

You will be referencing the `amzn_hist` DataFrame and the `params` object throughout this recipe.

How to do it...

We will explore three methods and libraries to connect to the Snowflake database. You will start by using the **Snowflake Python connector** (the `snowflake-connector-python` library), then explore the **Snowflake SQLAlchemy** (the `snowflake-sqlalchemy` library), and finally, you will explore the **Snowpark Python API** (the `snowflake-snowpark-python` library). Let's get started.

Using the Snowflake Python connector

The recipe in this section will utilize the `snowflake-connector-python` library for connecting and writing data to a Snowflake database:

1. Import the required libraries for this recipe:

```
import pandas as pd
from snowflake import connector
from snowflake.connector.pandas_tools import pd_writer, write_pandas
```

The `pandas_tools` module provides several functions for working with pandas DataFrames; this includes two writer methods (`write_pandas` and `pd_writer`).

The `write_pandas` method is for writing pandas DataFrames to a Snowflake database. Behind the scenes, the function will store the data to **Parquet** files, upload the files to a temporary stage, and finally, insert the data from the files to the specified table via the `COPY INTO` command.

On the other hand, the `pd_writer` method is an insertion method for inserting data into a Snowflake database with the `DataFrame.to_sql()` method and passing a SQLAlchemy engine. You will explore `pd_writer` in the *Using SQLAlchemy* section following this recipe.

2. Establish a connection to your Snowflake database instance and create a cursor object:

```
con = connector.connect(**params)
cursor = con.cursor()
```

The cursor object will be used to execute a SQL query to verify that the dataset has been properly written to the Snowflake database.

3. Write the `amzn_hist` DataFrame to Snowflake using the `write_pandas` method. The method takes the connection object (`con`), the DataFrame, the destination table name, and other optional arguments, such as `auto_create_table` and `table_type`, to name a few:

```
success, nchunks, nrows, copy_into = write_pandas(
    con,
    amzn_hist,
    auto_create_table=True,
    table_name='AMAZON',
    table_type='temporary'
)
```

When using `write_pandas`, it returns a tuple. In the previous code, we unpacked the tuple into `success`, `nchunks`, `nrows`, and `copy_into`. Let's inspect the values inside these objects:

```
print('success: ', success)
print('number of chunks: ', nchunks)
print('number of rows: ', nrows)
print('COPY INTO output', copy_into)
>>
success: True
number of chunks: 1
number of rows: 1569
COPY INTO output [('ntporcytgw/file0.txt', 'LOADED', 1569, 1569, 1, 0,
None, None, None, None)]
```

The `success` object is a Boolean (True or False) to indicate whether the function successfully writes the data to the specified table. The `nchunks` object represents the number of chunks during the write process, which, in this case, is the entire data written as one chunk. The `nrows` object represents the number of rows inserted by the function. Lastly, the `copy_into` object contains the output of the `COPY INTO` command.

Notice the use of `auto_create_table=True`; if this is not set to True and the `AMAZON` table did not already exist in Snowflake, `write_pandas` will throw an error. When it is set to True, we are explicitly asking `write_pandas` to create the table. Additionally, if the table exists, you can specify whether you want to overwrite the existing table with the `overwrite=True` parameter.

The `table_type` supports permanent, temporary, and transient table types in Snowflake. The parameter can take one of these values: `'temp'`, `'temporary'`, or `'transient'`. If an empty string is passed, `table_type=''`, then it will create a permanent table (the default behavior).

4. You can further validate that all 1,569 records are written in the temporary table:

```
cursor.execute('SELECT count(*) FROM AMAZON;')
count = cursor.fetchone()[0]
print(count)
>>
1569
```

Indeed, you have all 1,569 records written into the `AMAZON` table in Snowflake.

Using SQLAlchemy

The recipe in this section will utilize `snowflake-sqlalchemy` and the `snowflake-connector-python` libraries for connecting and writing data to a Snowflake database:

1. Import the required libraries for this recipe:

```
import pandas as pd
from snowflake.connector.pandas_tools import pd_writer
```

```
from snowflake.sqlalchemy import URL
from sqlalchemy import create_engine
```

2. You will use the URL function from the Snowflake SQLAlchemy library to construct the connection string and create the SQLAlchemy engine to establish a connection to your Snowflake instance:

```
url = URL(**params)
engine = create_engine(url)
```

3. Write the DataFrame to the Snowflake database using the to_sql() writer function. You will need to pass an insertion method; in this case, you will pass pd_writer:

```
try:
    amzn_hist.to_sql(
        'amazon_alchemy',
        engine,
        index=False,
        if_exists='replace'
    )
except:
    print('failed to write')
```

The preceding code uses a standard SQL INSERT clause, one per row. The Snowflake connector API provides an insertion method, `pd_writer`, that you can pass to the method parameter in the `to_sql` method, as shown:

```
try:
    amzn_hist.to_sql(
        'amazon_alchemy',
        engine,
        index=False, # Prevents DataFrame index from being
                     # written as a column
        if_exists='replace',
        method=pd_writer
    )
except:
    print('failed to write')
```

Behind the scenes, the `pd_writer` function will use the `write_pandas` function to write the DataFrame to the Snowflake database instead.

4. To read and verify that the data was written, you can use `pandas.read_sql_table()` to query the `amazon_alchemy` table:

```
pd.read_sql_table('amazon_alchemy', con=engine).info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1569 entries, 2019-01-02 to 2025-03-28
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  -
 0   Open    1569 non-null    float64
 1   High    1569 non-null    float64
 2   Low     1569 non-null    float64
 3   Close   1569 non-null    float64
 4   Volume  1569 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 73.5 KB
```

The new DataFrame contains all 1,569 records and the exact number of columns expected.

Using the Snowpark API

The recipe in this section will utilize the Snowpark API for writing a pandas DataFrame:

1. Import the required libraries for this recipe:

```
from snowflake.snowpark import Session
import pandas as pd
```

2. Create a session by establishing a connection with the Snowflake database:

```
session = Session.builder.configs(params).create()
```

3. Before writing the DataFrame, you must convert the pandas DataFrame to a Snowpark DataFrame:

```
amzn_snowpark_df = session.create_dataframe(amzn_hist)
```

Snowpark DataFrames uses lazy evaluation, meaning operations are only executed when results are needed. This provides many advantages over pandas DataFrames, including optimized query execution, reduced memory usage for large time series, and the ability to leverage Snowflake's distributed processing capabilities.

4. To write the Snowpark DataFrame, you can use the `write` and `save_as_table` methods:

```
amzn_snowpark_df.write.mode("overwrite").save_as_table("amazon_snowpark")
```

5. To read and verify that the data was written, you can use `session.table` to query the table:

```
amzn_df = session.table("amazon_snowpark")
```

If you are more comfortable with pandas, then you can convert the Snowpark DataFrame to a pandas DataFrame using the `to_pandas` method, as shown:

```
df = amzn_df.to_pandas()
df.shape
>>
(1569, 6)
```

The DataFrame contains all 1,569 records and all 6 columns as expected.

How it works...

The Snowflake Python API provides two mechanisms for writing pandas DataFrames to Snowflake, which are provided to you in the `pandas_tools` module:

```
from snowflake.connector.pandas_tools import pd_writer, write_pandas
```

When working with larger time series datasets, the `write_pandas` and `pd_writer` methods offer significant performance advantages over row-by-row insertion, as they leverage Snowflake's bulk data loading capabilities.

When specifying the `table_type` parameter in `write_pandas()`, you have three options:

- **Permanent tables:** The default table type (when using an empty string, `table_type=''`). These tables persist until explicitly dropped, are available to all users with appropriate privileges, and include both Time Travel (default retention period) and Fail-safe recovery features.
- **Temporary tables:** Created using `table_type='temp'` or `table_type='temporary'`. These tables are session-specific and automatically dropped when your session ends. They're only visible within the current session that created them, support Time Travel for up to one day, but don't have Fail-safe protection.
- **Transient tables:** Created using `table_type='transient'`. These tables persist beyond the current session until explicitly dropped. They serve as a middle ground between permanent and temporary tables, supporting limited Time Travel (up to one day) but no Fail-safe period, which reduces storage costs.

In the recipe, you used `pd_writer` and passed it as an insertion method to the `DataFrame.to_sql()` writer function. When using `pd_writer` within `to_sql()`, you can change the insertion behavior through the `if_exists` parameter, which takes three arguments:

- **fail**, which raises `ValueError` if the table exists
- **replace**, which drops the table before inserting new values
- **append**, which inserts the data into the existing table

If the table doesn't exist, SQLAlchemy takes care of creating the table for you and maps the data types from pandas DataFrames to the appropriate data types in the Snowflake database. This is also true when reading the data from Snowflake using the SQLAlchemy engine through `pandas.read_sql()`.

Note that `pd_writer` uses the `write_pandas` function behind the scenes. They both work by dumping the `DataFrame` into Parquet files, uploading them to a temporary stage, and, finally, copying the data into the table via `COPY INTO`.

You used the `write.mode()` method when using the Snowpark API to write the `DataFrame`. The `mode()` method accepts different write mode options:

- `append`: Append data of the `DataFrame` to the existing table. If the table does not exist, it will be created.
- `overwrite`: Overwrite the existing table by dropping the old table.
- `truncate`: Overwrite the existing table by truncating the old table.
- `errorifexists`: Throw an exception if the table already exists.
- `ignore`: Ignore the operation if the table already exists.

Keep in mind that the default value is `errorifexists`.

There's more...

In the recipe, you used the `session.create_dataframe()` method in Snowpark. When creating a `DataFrame` from a pandas `DataFrame`, Snowpark will store the pandas `DataFrame` in a temporary table and return a `DataFrame` pointing to that temporary table. This temporary table will be automatically dropped at the end of your session.

An alternative method in Snowpark to write a pandas `DataFrame` directly without the need to convert it into a Snowpark `DataFrame` is by using `session.write_pandas()`:

```
snowpark_df = session.write_pandas(amzn_hist,
                                     table_name="amazon_temp",
                                     auto_create_table=True,
                                     table_type="temp")
```

The `session.write_pandas()` method provides a more flexible approach for persisting pandas `DataFrames` to Snowflake tables. This function writes the `DataFrame` directly to Snowflake and returns a Snowpark `DataFrame` object referencing that table. You can control the persistence level through the `table_type` parameters:

- Using an empty string (default) creates a permanent table
- Using `"temp"` or `"temporary"` creates a temporary table
- Using `"transient"` creates a transient table

The `auto_create_table=True` parameter instructs Snowflake to automatically create the table if it doesn't exist; otherwise, you would need to manually create the table first.

While both `create_dataframe` and `write_pandas` can handle pandas `DataFrames`, they serve different purposes. The `create_dataframe` method is ideal for temporary analysis as it stores the data in a temporary table that will be dropped when your session ends. The `write_pandas` method is designed for persisting data as it writes to a permanent table (unless specified otherwise with `table_type`). The `write_pandas` method offers flexibility to choose the table type based on your needs.



The `write_pandas` method from the Snowflake connector's `pandas_tools` module and the `write_pandas` method in Snowpark serve similar purposes but with some implementation differences. Both methods are designed to efficiently write pandas DataFrames to Snowflake tables, but they're accessed through different interfaces. The connector's `write_pandas` method returns a tuple containing (success, nchunks, nrows, copy_into), while Snowpark's `write_pandas` method returns a Snowpark DataFrame object referring to the table where the data was written.

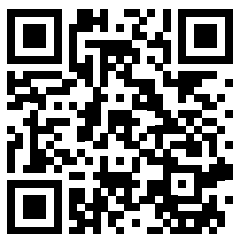
See also

- Visit the Snowflake documentation to learn more about the `write_pandas` and `pd_write` methods: https://docs.snowflake.com/en/user-guide/python-connector-api.html#write_pandas
- You can learn more about the pandas `DataFrame.to_sql()` function here: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html
- To learn more about the `write_pandas` method from the Snowpark API, refer to the official documentation here: https://docs.snowflake.com/en/developer-guide/snowpark/reference/python/1.30.0/snowpark/api/snowflake.snowpark.Session.write_pandas#snowflake.snowpark.Session.write_pandas

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



5

Working with Date and Time in Python

At the core of time-series data is **time**. **Time-series data** is a sequence of observations or data points captured in successive order and at regular time intervals. In the context of a pandas DataFrame, time-series data has an ordered index of type `DatetimeIndex`, as you have seen in earlier chapters, which offers efficient slicing, indexing, and time-based grouping of data.

Being familiar with manipulating the date and time in time-series data is an essential component of time-series analysis and modeling. This chapter provides recipes for common scenarios when working with date and time in time-series data.

Python has several built-in modules for working with date and time, such as `datetime`, `time`, `calendar`, and `zoneinfo`. Additionally, there are other popular libraries in Python that further extend its capability to work with and manipulate the date and time, such as `dateutil`, `pytz`, and `arrow`, to name a few.

In this chapter, you will be introduced to Python's `datetime` module. Then, the focus will shift to using **pandas** for more advanced and efficient date and time manipulation, as well as generating time-series DataFrames with a sequence of `DatetimeIndex` values.

The pandas library contains several date-specific and time-specific classes that inherit from the aforementioned Python modules. This eliminates the need to import additional date/time Python libraries. Key pandas classes covered include the following:

- **Timestamp**: Equivalent to Python's `datetime` class, and interchangeable in most cases.
- **Timedelta**: Equivalent to Python's `datetime.timedelta` objects
- **Period**: Represents a span of time, such as days, months, or years
- **DateOffset**: Allows you to shift data or time

The pandas library offers a more intuitive and powerful interface to handle most of your date and time manipulation needs without importing additional modules. It simplifies many challenging tasks associated with time-series data while maintaining an integrated, all-in-one solution.

Here is a list of the recipes covered in this chapter:

- Working with `DatetimeIndex`
- Providing a format argument to `DateTime`
- Working with Unix epoch timestamps
- Working with time deltas
- Converting `DateTime` with time zone information
- Working with date offsets
- Working with custom business days

In a real-world scenario, you may not use all of these techniques regularly, but they are invaluable when specific formatting or adjustments of dates are required, for instance:

- **`DatetimeIndex`** is commonly used when analyzing time-series data in pandas, such as stock prices or weather data, where the index represents time intervals
- **Epoch timestamps** are crucial when working with systems that log data in Unix time, such as server logs or IoT sensors
- **Time deltas** are essential for calculating differences between dates, such as determining the number of days between order and delivery dates
- **Time zones** are critical for global applications, ensuring correct scheduling or event tracking across different regions
- **Date offsets** and **custom business days** are often used in financial analysis, such as calculating business-specific deadlines or generating custom trading calendars

Technical requirements

In this chapter and going forward, we will extensively use pandas 2.3.3 (released on September 29, 2025).

Load these libraries in advance, since you will be using them throughout the chapter:

```
import pandas as pd
import numpy as np
import datetime as dt
```

You will use `dt`, `np`, and `pd` aliases going forward.

You can download the Jupyter notebooks from the GitHub repository at https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch5/Chapter_5.ipynb to follow along.

Working with `DatetimeIndex`

The pandas library offers numerous options and features to simplify tedious tasks when working with time-series data, dates, and time.

When working with time-series data in Python, it is common to load it into a pandas DataFrame with an index of type `DatetimeIndex`. As an index, the `DatetimeIndex` class extends pandas DataFrame capabilities to work more efficiently and intelligently with time-series data. This was demonstrated numerous times in *Chapter 1*, and *Chapter 2*. `DatetimeIndex` extends pandas' capabilities to support time-based indexing, filtering, resampling, and other operations commonly needed in time-series analysis.

By the end of this recipe, you will appreciate pandas' rich set of date functionality to handle almost any representation of date/time in your data. Additionally, you will learn how to use different functions in pandas to convert date-like objects to a `DatetimeIndex`.

How to do it...

In this recipe, you will explore Python's `datetime` module and learn about the `Timestamp` and `DatetimeIndex` classes and the relationship between them:

1. To understand the relationship between Python's `datetime.datetime` class and pandas' `Timestamp` and `DatetimeIndex` classes, you will create three different datetime objects representing the date 2024-01-01. You will then compare these objects to gain a better understanding:

```
dt1 = dt.datetime(2024,1,1)
dt2 = pd.Timestamp('2024-1-1')
dt3 = pd.to_datetime('2024-1-1')
```

Inspect their representations:

```
print(dt1)
print(dt2)
print(dt3)
>>
2024-01-01 00:00:00
2024-01-01 00:00:00
2024-01-01 00:00:00
```

Inspect their data types:

```
print(type(dt1))
print(type(dt2))
print(type(dt3))
>>
<class 'datetime.datetime'>
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

Examine how the different datetime representations compare:

```
dt1 == dt2 == dt3
>> True
```

The comparison returned True, indicating all three objects represent the same point in time, despite being of different types.

Check inheritance relationship using `isinstance(<date object>, <datetime class>)`, which helps to check whether a date object (e.g., `dt2`) is an instance of the datetime class (e.g., `pd.Timestamp`):

```
isinstance(dt2, dt.datetime)
>> True
isinstance(dt2, pd.Timestamp)
>> True
isinstance(dt1, pd.Timestamp)
>> False
```

You can see from the preceding code that pandas' Timestamp object is equivalent to Python's `datetime.datetime` object.

```
issubclass(pd.Timestamp, dt.datetime)
>> True
```

Note that `dt2` is an instance of the `Timestamp` class, and the `Timestamp` class is a subclass of Python's `datetime.datetime` class, but the reverse is not true.

2. When you used the `pandas.to_datetime()` function, it returned a `Timestamp` object. Now, use the `to_datetime()` function on a list and examine the output:

```
sample_dates = ['2024-01-01', '2024-01-02', '2024-01-03']
pd_dates = pd.to_datetime(sample_dates)
print(pd_dates)
>>
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03'],
              dtype='datetime64[ns]', freq=None)

print(type(pd_dates))
>>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Interestingly, the output is now of type `DatetimeIndex`, created using the same `to_datetime()` function that you used earlier. Previously, when using the same function on an individual object, the result was of type `Timestamp`, but when applied to a list, it produced a sequence of type `DatetimeIndex`. You will perform one more task to make things clearer.

Access an individual element, for example, the first item (slice) from the `pd_dates` variable:

```
print(pd_dates[0])
print(type(pd_dates[0]))
>>
2024-01-01 00:00:00
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

Here, you can see that a `DatetimeIndex` is essentially a sequence of `Timestamp` objects.

3. You can convert various datetime representations, such as strings, integers, lists, pandas Series, or other datetime objects, into a `DatetimeIndex` using the `to_datetime()` function.

Let's create a list of various date formats and call it `dates`:

```
dates = [
    '2024-01-01', # date str format %Y-%m-%d
    '2/1/2024', # date str format %m/%d/%Y
    '03-01-2024', # date str format %m-%d-%Y
    'April 1, 2024', # date str format %B %d, %Y
    '20240501', # date str format %Y%m%d
    np.datetime64('2024-07-01'), # numpy datetime64
    dt.datetime(2024, 8, 1), # python datetime
    pd.Timestamp(2024,9,1) # pandas Timestamp
]
```

Convert the list to a uniform format using `map()` and `pd.to_datetime()`:

```
parsed_dates_map = pd.Series(list(map(pd.to_datetime, dates)))
print(parsed_dates_map)
>>
0    2024-01-01
1    2024-02-01
2    2024-03-01
3    2024-04-01
4    2024-05-01
5    2024-07-01
6    2024-08-01
7    2024-09-01
dtype: datetime64[ns]
```

The `map()` function applies `to_datetime()` to each element in the list, ensuring all dates are parsed into a consistent pandas `Timestamp` format:

```
DatetimeIndex(
    ['2021-01-01', '2021-02-01', '2021-03-01', '2021-04-01',
```

```
'2021-05-01', '2021-07-01', '2021-08-01', '2021-09-01'],
dtype='datetime64[ns]', freq=None
)
```

Notice how the `to_datetime()` function properly parsed the entire list of different string representations and date types, such as Python's `Datetime` and NumPy's `datetime64`.

Similarly, you could have used the `pd.DatetimeIndex()` constructor directly, as shown:

```
parsed_dates = pd.Series(pd.DatetimeIndex(dates))
```

This would produce similar results.



In earlier versions of pandas, the `pd.to_datetime()` function could seamlessly parse a mix of date formats in a single call, even if they varied significantly in representation (e.g., `YYYY-MM-DD`, `MM/DD/YYYY`, or natural language formats such as `April 1, 2021`). However, with recent updates, pandas has adopted a stricter mechanism for handling mixed date formats.

This stricter behavior improves accuracy but requires users to handle mixed formats explicitly, such as by mapping `pd.to_datetime()` over individual entries, using the `DatetimeIndex` constructor, or explicitly passing a format.

More on that in the *How it works...* section.

4. The `DatetimeIndex` object gives access to many useful properties and methods to extract additional date and time properties. As an example, you can extract `day_name`, `month`, `year`, `days_in_month`, `quarter`, `is_quarter_start`, `is_leap_year`, `is_month_start`, `is_month_end`, and `is_year_start`. The following code shows how this can be done:

```
parsed_dates = pd.DatetimeIndex(dates)

print(f'Name of Day : {parsed_dates.day_name()}')
print(f'Month : {parsed_dates.month}')
print(f'Month Name : {parsed_dates.month_name()}')
print(f'Year : {parsed_dates.year}')
print(f'Days in Month : {parsed_dates.days_in_month}')
print(f'Quarter {parsed_dates.quarter}')
print(f'Is Quarter Start : {parsed_dates.is_quarter_start}')
print(f'Days in Month: {parsed_dates.days_in_month}')
print(f'Is Leap Year : {parsed_dates.is_leap_year}')
print(f'Is Month Start : {parsed_dates.is_month_start}')
print(f'Is Month End : {parsed_dates.is_month_end}')
print(f'Is Year Start : {parsed_dates.is_year_start}')
```

The preceding code produces the following results:

```
Name of Day : Index(
    ['Monday', 'Thursday', 'Friday', 'Monday',
     'Wednesday', 'Monday', 'Thursday', 'Sunday'],
    dtype='object')
Month : Index([1, 2, 3, 4, 5, 7, 8, 9], dtype='int32')
Month Name: Index(
    ['January', 'February', 'March', 'April',
     'May', 'July', 'August', 'September'],
    dtype='object')
Year : Index([2024, 2024, 2024, 2024, 2024, 2024, 2024, 2024],
             dtype='int32')
Days in Month : Index([31, 29, 31, 30, 31, 31, 31, 30],
                      dtype='int32')
Quarter Index([1, 1, 1, 2, 2, 3, 3, 3], dtype='int32')
Is Quarter Start : [ True False False  True False  True False False]
Days in Month: Index([31, 29, 31, 30, 31, 31, 31, 30],
                    dtype='int32')
Is Leap Year : [ True  True  True  True  True  True  True  True]
Is Month Start : [ True  True  True  True  True  True  True  True]
Is Month End : [False False False False False False False False]
Is Year Start : [ True False False False False False False False]
```

These properties and methods will be very useful when transforming your time-series datasets for analysis.

How it works...

The `pandas.to_datetime()` function is a powerful tool that can intelligently parse different date representations from strings. For example, strings such as `'2024-01-01'`, `'2/1/2024'`, `'03-01-2024'`, `'April 1, 2024'`, and `'20240501'` were parsed correctly in the previous section. Other date representations, such as `'April 1, 2024'` and `'1 April 2024'`, can be parsed using the `to_datetime()` function as well, and I'll leave it to you to explore additional examples that come to mind.

However, in recent versions of pandas, `to_datetime()` has adopted a stricter parsing mechanism when working with mixed date formats. If the function encounters a mix of incompatible formats, it will raise an error or produce missing values, depending on the `errors` parameter. For example, our date object contains mixed formats:

```
pd.to_datetime(
    dates,
    infer_datetime_format=True,
    errors='coerce'
)
```


In this case, if pandas cannot **uniformly** infer the format for all date strings in sequence, it will replace unparseable entries with NaT, indicating a missing datetime value. Here is an example output:

```
DatetimeIndex(['2024-01-01',      'NaT',      'NaT',      'NaT',
               'NaT', '2024-07-01', '2024-08-01', '2024-09-01'],
              dtype='datetime64[ns]', freq=None)
```

You will learn more about NaT in the *Performing data quality checks* recipe in *Chapter 6*.

Similarly, using `errors='ignore'` will bypass the parsing process for incompatible values and return the original input values as is. Here is an example:

```
parsed_dates = pd.to_datetime(
    dates,
    errors='ignore'
)

print(parsed_dates)
>>
Index(['2024-01-01',      '2/1/2024',      '03-01-2024',
       'April 1, 2024',      '20240501',      2024-07-01,
       2024-08-01 00:00:00, 2024-09-01 00:00:00],
      dtype='object')
```

When the `errors` parameter is set to `'ignore'`, pandas will not raise an error if it stumbles upon a date representation it cannot parse. Instead, the input values are passed as is.

Notice from the preceding output that the `to_datetime()` function returned an `Index` dtype object (representing strings or mixed types) instead of a `DatetimeIndex`.

In pandas, there are different representations to indicate missing values:

- `np.NaN` represents missing numeric values (not a number)
- `pd.NaT` represents missing datetime values (not a time)
- `np.nan` is used to represent missing scalar values (not available)

The `errors` parameter in `to_datetime` supports three options:

- `raise`: Raises an exception when parsing fails (it will error out).
- `coerce`: Will not cause it to raise an exception. Instead, it will just replace values with `pd.NaT` when it cannot parse an entry.
- `ignore`: Will also not cause it to raise an exception. Instead, it will just pass the original value when it cannot parse an entry.

`DatetimeIndex` provides access to several useful methods and attributes, which you explored in *step 4* of the previous *How to do it...* section. These include the following:

- `day_name()`: Returns the name of the day of the week (e.g., Monday, Tuesday)

- `month`: Provides the month component of the date as an integer (1 through 12)
- `month_name()`: Returns the full name of the month (e.g., January, February)
- `year`: Retrieves the year component of the date as an integer
- `days_in_month`: Gives the number of days in the month for the given date
- `quarter`: Indicates the quarter of the year for the date (1 through 4)
- `is_quarter_start`: Boolean value indicating whether the date is the first day of a quarter (True or False)
- `is_leap_year`: Boolean value indicating whether the year of the date is a leap year (True or False)
- `is_month_start`: Boolean value indicating whether the date is the first day of its month (True or False)
- `is_month_end`: Boolean value indicating whether the date is the last day of its month (True or False)
- `is_year_start`: Boolean value indicating whether the date is the first day of its year (True or False)

There's more...

An alternate way to generate a `DatetimeIndex` is by using the `pandas.date_range()` function. The following code specifies a starting date, the number of periods to generate, and a daily frequency (`freq='D'`):

```
pd.date_range(start='2024-1-01', periods=3, freq='D')
>>
DatetimeIndex(['2024-12-01', '2024-12-02', '2024-12-03'],
              dtype='datetime64[ns]', freq='D')
```

The `date_range()` function requires at least three of the following four parameters to be specified – `start`, `end`, `periods`, and `freq`. If you do not provide enough information, you will get a `ValueError` exception with the following message:

```
ValueError: Of the four parameters: start, end, periods, and freq, exactly
three must be specified
```

Let's explore different parameter combinations to examine how the function generates date ranges.

In the first example, specify a **start** date, an **end** date, and a daily **frequency**. The function will always return a range of equally spaced time points:

```
pd.date_range(start='2024-11-25',
              end='2024-12-02',
              freq='D')
>>
DatetimeIndex(['2024-11-25', '2024-11-26', '2024-11-27', '2024-11-28',
              '2024-11-29', '2024-11-30', '2024-12-01', '2024-12-02'],
              dtype='datetime64[ns]', freq='D')
```

In the second example, specify a **start** date, an **end** date, and the number of **periods**. Remember that the function will always return a range of equally spaced time points:

```
pd.date_range(start='2024-11-25',
              end='2024-12-01',
              periods=4)

>>
DatetimeIndex(['2024-11-25', '2024-11-27', '2024-11-29', '2024-12-01'],
              dtype='datetime64[ns]', freq=None)
```

In the following example, provide an **end** date, the number of **periods**, and the daily **frequency**:

```
pd.date_range(start='2024-11-25',
              periods=5,
              freq='D')

>>
DatetimeIndex(['2024-11-25', '2024-11-26', '2024-11-27', '2024-11-28',
              '2024-11-29'],
              dtype='datetime64[ns]', freq='D')
```

The `date_range()` function can work with a minimum of two parameters if the information is sufficient to generate equally spaced time points and infer the missing parameters. Here is an example of providing **start** and **end** dates only:

```
pd.date_range(start='2024-11-25', end='2024-12-02')

>>
DatetimeIndex(['2024-11-25', '2024-11-26', '2024-11-27', '2024-11-28',
              '2024-11-29', '2024-11-30', '2024-12-01', '2024-12-02'],
              dtype='datetime64[ns]', freq='D')
```

Notice that pandas was able to construct the date sequence using the **start** and **end** dates and default to **daily frequency** (`freq='D'`). Here is another example, with **start** date and **periods** only:

```
pd.date_range(start='2024-11-01', periods=3)

>>
DatetimeIndex(['2024-11-01', '2024-11-02', '2024-11-03'],
              dtype='datetime64[ns]', freq='D')
```

With **start** and **periods**, pandas has enough information to construct the date sequence and default to **daily frequency** (`freq='D'`).

Now, here is an example that lacks enough information on how to generate the sequence and will cause pandas to throw an error:

```
try:
    pd.date_range(start='2024-11-25', freq='D')
except ValueError:
```

```
print("Need additional parameters")
>>
Need additional parameters
```

Note that with just a **start** date and **frequency**, pandas does not have enough information to construct the date sequence. Therefore, adding either **periods** or the **end** date would resolve the issue.

Let's put everything we have learned from generating a `DatetimeIndex` to extracting datetime properties. In the following example, you will use the `date_range()` function to create a `DataFrame` with one `Date` column. You will then create additional columns using the different properties and methods:

```
df = pd.DataFrame(pd.date_range(start='2024-11-25', periods=5),
                  columns=['Date'])

df['days_in_month'] = df['Date'].dt.days_in_month
df['day_name'] = df['Date'].dt.day_name()
df['month'] = df['Date'].dt.month
df['month_name'] = df['Date'].dt.month_name()
df['year'] = df['Date'].dt.year
df['days_in_month'] = df['Date'].dt.days_in_month
df['quarter'] = df['Date'].dt.quarter
df['is_quarter_start'] = df['Date'].dt.is_quarter_start
df['days_in_month'] = df['Date'].dt.days_in_month
df['is_leap_year'] = df['Date'].dt.is_leap_year
df['is_month_start'] = df['Date'].dt.is_month_start
df['is_month_end'] = df['Date'].dt.is_month_end
df['is_year_start'] = df['Date'].dt.is_year_start
```

The preceding code should produce the following `DataFrame`.

	Date	days_in_month	day_name	month	month_name	year	quarter	is_quarter_start	is_leap_year	is_month_start	is_month_end	is_year_start
0	2024-11-25	30	Monday	11	November	2024	4	False	True	False	False	False
1	2024-11-26	30	Tuesday	11	November	2024	4	False	True	False	False	False
2	2024-11-27	30	Wednesday	11	November	2024	4	False	True	False	False	False
3	2024-11-28	30	Thursday	11	November	2024	4	False	True	False	False	False
4	2024-11-29	30	Friday	11	November	2024	4	False	True	False	False	False

Figure 5.1: A time-series `DataFrame` with 5 rows and 12 columns



THE `Series.dt` ACCESSOR

In the previous example, you used the `.dt` accessor to access the datetime properties of the `Date` column. The `.dt` accessor is a pandas property that provides a convenient way of accessing a wide range of datetime properties of a `Series`. This is particularly useful for extracting specific information such as day names, months, years, and more.

The `date_range` function is a very useful tool for different scenarios, such as generating date series for a wide range of frequencies (e.g., daily, monthly, yearly), reindexing a `DataFrame` to identify missing data, and simplifying time-series analysis by creating evenly spaced time intervals.

See also

To learn more about pandas' `to_datetime()` function and the `DatetimeIndex` class, please check out these resources:

- pandas.DatetimeIndex documentation: <https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html>
- pandas.to_datetime documentation: https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html

Providing a format argument to DateTime

When working with datasets extracted from different data sources, you may encounter date columns stored as strings, whether from files or databases. In the previous recipe, *Working with DatetimeIndex*, you explored the `pandas.to_datetime()` function, which can parse various date formats with minimal input. However, for more granular control and to ensure dates are parsed correctly, you may need to specify the exact format of the date strings.

In this recipe, you will be introduced to the `strptime` and `strftime` methods, which are part of Python's standard library (`datetime`), as well as how to leverage the `format` argument in `pandas.to_datetime()` to handle different date formats. Additionally, you will learn how to convert strings representing dates into `datetime` or `date` object instances of the `datetime.datetime` or `datetime.date` class.

How to do it...

Python's `datetime` module contains the `strptime()` method to create `datetime` or `date` from a string that contains a date. You will first explore how you can do this in Python and then extend this to pandas:

1. The `datetime.strptime()` method requires you to specify the exact format of the string using format codes. The following are examples of parsing four different representations of **January 1, 2024**, each producing the same output – `datetime.datetime(2024, 1, 1, 0, 0)`:

```
dt.datetime.strptime('1/1/2024', '%m/%d/%Y')
dt.datetime.strptime('1 January, 2024', '%d %B, %Y')
dt.datetime.strptime('1-Jan-2024', '%d-%b-%Y')
dt.datetime.strptime('Saturday, January 1, 2024', '%A, %B %d, %Y')
>>
datetime.datetime(2024, 1, 1, 0, 0)
```

Note that the output is a `datetime` object, representing the year, month, day, hour, and minute. You can specify only the date representation with the `.date()` method:

```
dt.datetime.strptime('1/1/2024', '%m/%d/%Y').date()
>>
datetime.date(2024, 1, 1)
```

The output is a date object instead of datetime. You can print a readable version of datetime simply by using the `print()` function:

```
dt_1 = dt.datetime.strptime('1/1/2024', '%m/%d/%Y')
print(dt_1)
>>
2024-01-01 00:00:00
```

To only print the date portion without time, you can use the following:

```
print(dt_1.date())
>>
2024-01-01
```

2. The `pandas.to_datetime()` function provides an alternative way of parsing strings, offering greater flexibility and support for handling large datasets. The following are equivalent examples using `pandas.to_datetime()`, each producing the same output `Timestamp('2024-01-01 00:00:00')`:

```
pd.to_datetime('1/1/2024', format='%m/%d/%Y')
pd.to_datetime('1 January, 2024', format='%d %B, %Y')
pd.to_datetime('1-Jan-2024', format='%d-%b-%Y')
pd.to_datetime('Saturday, January 1, 2024', format='%A, %B %d, %Y')
>>
Timestamp('2024-01-01 00:00:00')
```

Similarly, you can print a readable representation of the `Timestamp` object:

```
dt_2 = pd.to_datetime('1/1/2024', format='%m/%d/%Y')
print(dt_2)
>>
2024-01-01 00:00:00
```

To print only the date portion without time, you can use the following:

```
print(dt_2.date())
>>
2024-01-01
```

3. The `pandas.to_datetime()` function has an advantage over Python's `datetime.strptime()` because it can intelligently infer and parse a variety of date representations, with minimal input or specifications. The following are examples where the `format` parameter is omitted; all four functions will produce the same expected output:

```
pd.to_datetime('1/1/2024')
pd.to_datetime('1 January, 2024')
pd.to_datetime('1-Jan-2024')
```

```
pd.to_datetime('Saturday, January 1, 2024')
>>
Timestamp('2024-01-01 00:00:00')
```

Unlike `datetime.strptime()`, which requires a specific string to parse dates, `pandas.to_datetime()` can automatically detect and parse many common date formats without the need for explicit formatting (in most cases). While it is generally good practice to provide a format string for clarity when parsing dates, one of the advantages of `to_datetime()` is its flexibility, making it easy to use, especially for exploratory data analysis.

How it works...

In this recipe, you used Python's `datetime.datetime` and `pandas.to_datetime()` methods to parse dates in string formats. When using Python's `datetime`, you had to use the `datetime.datetime.strptime()` function to specify the exact date format representation in the string using format codes (for example, `%d`, `%B`, and `%Y`).

For example, in `datetime.strptime('1 January, 2022', '%d %B, %Y')`, you provided the `%d`, `%B`, and `%Y` format codes in this exact order, including separators (spaces and commas) to match the formatting provided in the string. Let's break this down:

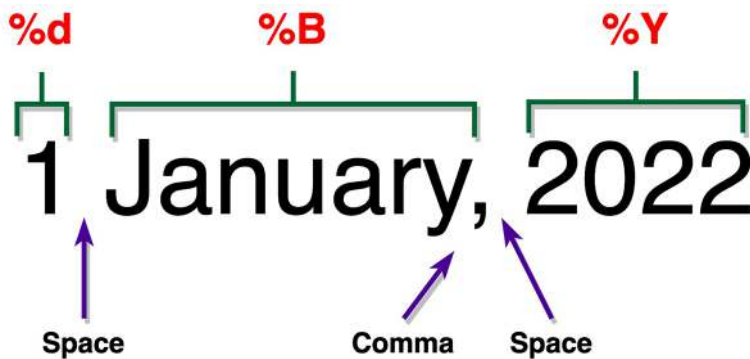


Figure 5.2: Understanding the format

`%d` indicates that the first value is a zero-padded digit representing the day of the month, followed by a space to display spacing between the digit and the next object.

`%B` is used to indicate that the second value represents the month's full name. Note that this was followed by a comma (,) to describe the exact format in the string, for example, "January,". Therefore, it is crucial to match the format in the strings you are parsing to include any commas, hyphens, backslashes, spaces, or whichever separator characters are used.

To adhere to the string format, there is a space after the comma (,), followed by `%Y` to reflect that the last value represents a four-digit year.



FORMAT DIRECTIVES

Remember, format codes are always prefixed with a percent sign (%) followed by the format code (a letter with or without a negative sign). This is referred to as a **formatting directive**. For example, lowercase y, as in %y, represents the year 22 without the century, while uppercase Y, as in %Y, represents the year 2022 with the century.

You can find a full list of Python directives here: <https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes>.

When using pandas' `to_datetime()` function, you were able to parse the same string objects as with `strptime()`. The biggest difference is that the `to_datetime()` function can accurately parse the strings without explicitly providing an argument to the `format` parameter. That is one of many advantages of using pandas for time-series analysis, especially when handling complex date and datetime scenarios.

There's more...

Now that you know how to use pandas' `to_datetime()` to parse string objects to `datetime`, let's apply this knowledge to transform a `DataFrame` column containing date information in string format into a `datetime` data type.

In the following code, you will create a small `DataFrame`:

```
df = pd.DataFrame({
    'Date': ['January 1, 2024', 'January 2, 2024', 'January 3, 2024'],
    'Sales': [23000, 19020, 21000]
})
df
>>
```

	Date	Sales
0	January 1, 2024	23000
1	January 2, 2024	19020
2	January 3, 2024	21000

Check the `DataFrame`'s structure and data types using `df.info()`:

```
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    Date      3 non-null      object
1   Sales      3 non-null      int64
dtypes: int64(1), object(1)
memory usage: 176.0+ bytes
```


Notice that the Date column is currently of type object (string). Let's parse the Date column into a datetime format and then assign it as a DataFrame index (DatetimeIndex):

```
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3 entries, 2022-01-01 to 2022-01-03
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype
---  -
0   Sales    3 non-null        int64
dtypes: int64(1)
memory usage: 48.0 bytes
```

Notice that the Date column has been successfully parsed into a DatetimeIndex. Since the Date column is now a DataFrame's index, it no longer exists as a separate column. The Sales column remains as the only data column in the DataFrame.

See also

- You can visit the official documentation page to learn more about pandas' `to_datetime()` function: https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html.

Working with Unix epoch timestamps

Unix epoch timestamps, sometimes referred to as **Unix time** or **POSIX time**, are a common way of representing dates and times as integers. This integer value represents the number of seconds elapsed since a specific reference point. For Unix-based timestamps, the reference point is January 1, 1970, at midnight (00:00:00 UTC). This date and time serve as the baseline, also known as the epoch, starting at zero. Every second beyond that time is simply an increment by one.

Many databases, applications, and systems store dates and times in this numeric format, making it mathematically straightforward to perform operations such as conversions, increments, and decrements. It is worth noting that **Unix epoch timestamps** are based on **Coordinated Universal Time (UTC)**, the international standard for timekeeping. Using UTC is an obvious choice when building globally distributed applications because it standardizes how dates and timestamps are stored, freeing you from concerns such as daylight saving or time zone differences across the globe. UTC is widely used in aviation systems, weather forecast systems, the International Space Station, and more.

At some point, you will likely encounter Unix epoch timestamps. To make more sense of these values, you will need to convert them into a human-readable format. This is exactly what we will cover in this recipe. You will discover how simple and efficient it can be to work with Unix epoch timestamps using pandas' built-in functions.

How to do it...

Before we start converting Unix timestamps into human-readable datetime objects, let's first build some intuition about storing dates and time as numeric objects (floating-point numbers):

1. Use the `time` module from Python's standard library to request the current time in seconds. This will be the time in seconds since the Unix epoch, starting from January 1, 1970, at 00:00:00 UTC:

```
import time
epoch_time = time.time()
print(epoch_time)
print(type(epoch_time))
>>
1732048268.954201
<class 'float'>
```

Copy the numeric value you have and visit <https://www.epoch101.com>. The website should display your current epoch time. If you scroll down, you can paste the number under Unix Epoch to convert it to a human-readable format. Make sure that you select **seconds as the unit**, as shown in the following figure:

The screenshot shows the 'Convert Unix Timestamp to Human Readable Format' interface. On the left, under 'Unix Epoch', the value '1732048268.954201' is entered in a text box. Below it, 'seconds' is selected with a radio button, and 'milliseconds' is unselected. A link '(need help converting from secs to millisec or vice versa?)' is present. A blue button labeled 'Convert to Human Readable →' is at the bottom left. On the right, under 'GMT', the output 'Tue, 19 Nov 2024 20:31:08 GMT' is shown in a text box. Below that, under 'Local', the output 'Tue Nov 19 2024 23:31:08 GMT+0300 (GMT+03:00)' is shown in a text box. At the bottom right, the text '354 days, 20 hours, 10 minutes, and 22 seconds ago' is displayed.

Figure 5.3: Converting a Unix timestamp into a human-readable format in both GMT and local time

The output will show both GMT and local time formats. In Figure 5.3, the GMT format is given as Tue, 19 Nov 2024 20:31:08 GMT and the local format as Tue Nov 19 2024, 23:31:08 GMT+0300 (GMT+03:00).

Let's see how pandas converts the epoch timestamp. The convenience here is that you will be using the same `pandas.to_datetime()` function that you should be familiar with by now, as you have used it in the previous two recipes from this chapter. This is one of the many conveniences you get when using pandas. For example, in the following code, you will use `pandas.to_datetime()` to parse the Unix epoch 1732048268.954201:

```
import pandas as pd
t = pd.to_datetime(1732048268.954201, unit='s')
print(t)
>>
2024-11-19 20:31:08.954200983
```

Note the need to specify `unit='s'` (units as seconds), which tells pandas that our timestamp is in seconds since the epoch. The output is similar to that in *Figure 5.3* for the GMT format.

If you want the datetime object to be time zone-aware – for example, the US/Pacific time zone – then you can use `tz_localize()`, such as `tz_localize('US/Pacific')`. If the original time zone of the data is unknown and you want to make it time zone-aware, then you can localize it to UTC using `tz_localize('UTC')`. The `tz_localize()` function attaches a time zone to the datetime object without altering the actual time values.

Note that Python's `time.time()` returns a Unix timestamp in UTC.

2. To convert the time zone of a time zone-aware datetime object to another time zone, use `tz_convert()`. If your datetime object is not time zone-aware, and to get a more accurate conversion, it is better to do it in two steps:
 - **Localize the time zone-naive datetime object by assigning it the UTC time zone using `tz_localize('UTC')`.** This ensures consistency, as UTC does not observe daylight saving time.
 - **Convert the time to the desired time zone using `tz_convert()`,** which modifies the time according to the rules of the target time zone.

The following code shows how this is done to convert to the Pacific time zone:

```
new_t = t.tz_localize('UTC').tz_convert('US/Pacific')
new_t
>>
Timestamp('2024-11-19 12:31:08.954200983-0800', tz='US/Pacific')
```

3. Let's put all of this together. You will convert a DataFrame that contains a datetime column in Unix epoch format into a human-readable format. Start by creating a new DataFrame with Unix epoch timestamps:

```
df = pd.DataFrame({
    'unix_epoch': [1641110340, 1641196740, 1641283140, 1641369540],
    'Sales': [23000, 19020, 21000, 17030]
})
df
>>
```

	unix_epoch	Sales
0	1641110340	23000
1	1641196740	19020
2	1641283140	21000
3	1641369540	17030

4. Create a new Date column by parsing the `unix_epoch` column into a datetime object (defaulting to GMT). Localize the time zone-naive object to UTC, then convert it into the desired time zone (e.g., 'US/Pacific'). Finally, set the Date column as the index:

```
df['Date'] = pd.to_datetime(df['unix_epoch'], unit='s')
df['Date'] = df['Date'].dt.tz_localize('UTC').dt.tz_convert('US/Pacific')
df.set_index('Date', inplace=True)
df
```

```
>>                                unix_epoch  Sales
Date
2022-01-01 23:59:00-08:00      1641110340   23000
2022-01-02 23:59:00-08:00      1641196740   19020
2022-01-03 23:59:00-08:00      1641283140   21000
2022-01-04 23:59:00-08:00      1641369540   17030
```

5. Inspect the DataFrame index and notice the additional time zone information:

```
print(df.index)
>>
DatetimeIndex(['2022-01-01 23:59:00-08:00', '2022-01-02 23:59:00-08:00',
               '2022-01-03 23:59:00-08:00', '2022-01-04 23:59:00-08:00'],
              dtype='datetime64[ns, US/Pacific]', name='Date', freq=None)
```

Note that since the Date column was of the `datetime` type (not `DatetimeIndex`), you had to use the `Series.dt` accessor to tap into the built-in methods and attributes for the datetime objects. In the last step, with `set_index()`, you converted `datetime` to a `DatetimeIndex` object (a DataFrame index). If you recall from the *Working with DatetimeIndex* recipe of this chapter, a `DatetimeIndex` object can access any of the `datetime` methods and attributes without using the `.dt` accessor.

6. If you do not need the time in your index (`DatetimeIndex`), given your data is daily and there is no use case for using time, then you can request just the date, as shown here:

```
df.index.date
>>
array([datetime.date(2022, 1, 1), datetime.date(2022, 1, 2),
       datetime.date(2022, 1, 3), datetime.date(2022, 1, 4)], dtype=object)
```

Notice how the output now displays the date without the time.

How it works...

Understanding Unix epoch time is more relevant as one engages in technical fields where precise and standardized time representation is critical.

So far, you have used pandas' `to_datetime()` to parse dates in string format to a datetime object by leveraging the `format` attribute (see the *Providing a format argument to DateTime* recipe). In this recipe, you used the same function, but instead of providing a value to `format`, you passed a value to the `unit` parameter, as in `unit='s'`.

The `unit` parameter tells pandas which unit to use when calculating the difference from the epoch start. In this case, the request was in seconds. However, there is another critical parameter that you do not need to adjust (in most cases), which is the `origin` parameter. For example, the default value is `origin='unix'`, which indicates that the calculation should be based on the Unix (or POSIX) time set to 01-01-1970 00:00:00 UTC.

This is what the actual code looks like:

```
pd.to_datetime(1635220133.855169, unit='s', origin='unix')
>>
Timestamp('2021-10-26 03:48:53.855169024')
```

You can modify `origin` to use a different reference date to calculate the datetime value. In the following example, `unit` is specified to be in days and `origin` is set at January 1, 2023:

```
pd.to_datetime(45, unit='D', origin='2024-11-1')
>>
Timestamp('2024-12-16 00:00:00')
```

The code calculates the date that is 45 days after the origin date (November 1, 2024).

When using `tz_localize` and `tz_convert`, these functions can work directly on any DataFrame that has a `DatetimeIndex`, eliminating the need for the `.dt` accessor as you have done earlier (required when working with datetime columns). Here is an illustration with a new DataFrame that has a `DatetimeIndex`:

```
df = pd.DataFrame({
    'unix_epoch': [1641113940, 1641117540, 1641121140, 1641124740],
    'count_shoppers': [12, 16, 18, 10]
})
df['Date'] = pd.to_datetime(df['unix_epoch'], unit='s')
df.set_index('Date', inplace=True)
print(df)
>>
```

	unix_epoch	count_shoppers
Date		
2022-01-02 08:59:00	1641113940	12
2022-01-02 09:59:00	1641117540	16
2022-01-02 10:59:00	1641121140	18
2022-01-02 11:59:00	1641124740	10

You can localize and convert the `DatetimeIndex` directly without the `.dt` accessor:

```
updated_df = df.tz_localize('UTC').tz_convert('US/Pacific')
print(updated_df)
>>
```

	unix_epoch	count_shoppers
Date		
2022-01-02 00:59:00-08:00	1641113940	12
2022-01-02 01:59:00-08:00	1641117540	16
2022-01-02 02:59:00-08:00	1641121140	18
2022-01-02 03:59:00-08:00	1641124740	10

There's more...

If you would like to store your `datetime` value in a Unix epoch, you can do this by subtracting 1970-01-01 and then floor-dividing by a unit of 1 second. Python uses `/` as the division operator, `//` as the floor division operator to return the floored quotient, and `%` as the modulus operator to return the remainder from a division.

Start by creating a new pandas `DataFrame`:

```
df = pd.DataFrame({
    'Date': pd.date_range('11-01-2024', periods=5),
    'order' : range(5)
})
print(df)
>>
```

	Date	order
0	2024-11-01	0
1	2024-11-02	1
2	2024-11-03	2
3	2024-11-04	3
4	2024-11-05	4

You can then perform the transformation, as follows:

```
df['Unix Time'] = (df['Date'] - pd.Timestamp("1970-01-01")) //
pd.Timedelta("1s")
df
>>
```

	Date	order	Unix Time
0	2024-11-01	0	1730419200
1	2024-11-02	1	1730505600
2	2024-11-03	2	1730592000
3	2024-11-04	3	1730678400
4	2024-11-05	4	1730764800

In the previous code, we subtracted the epoch from each date and then floor-divided by a unit of "1s" (one second).

You can validate this by converting it back using the following:

```
pd.to_datetime(df['Unix Time'], unit='s')
>>
0    2024-11-01
1    2024-11-02
2    2024-11-03
3    2024-11-04
4    2024-11-05
Name: Unix Time, dtype: datetime64[ns]
```

You have now generated your Unix epochs. There are different ways to achieve similar results. The preceding example is the recommended approach from pandas, which you can read more about here: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#from-timestamps-to-epoch.

You will learn more about `Timedelta` in the next recipe.

See also

- For more information on what you can do with the `to_datetime()` function, you can refer to the *Working with DatetimeIndex* recipe
- You can visit the official documentation page for pandas' `to_datetime()` function here: https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html

Working with time deltas

When working with time-series data, you may need to perform some calculations on datetime columns, such as adding or subtracting time. For example, you might want to add 30 days to a purchase date to determine when a return policy expires or calculate the end date of a product warranty. The `Timedelta` class and the `to_timedelta()` function make it easy to handle these scenarios, allowing you to derive new datetime objects by adding or subtracting at different ranges or increments, such as seconds, days, and weeks. This includes handling time zone-aware calculations.

You will explore two practical approaches in pandas to capture date/time differences – the `pandas.Timedelta` class and the `pandas.to_timedelta` function.

How to do it...

In this recipe, you will work with hypothetical sales data for a retail store. You will generate a sales `DataFrame`, which will contain items purchased from the store and their purchase date. You will then explore different scenarios using the `Timedelta` class and the `to_timedelta()` function:

1. Start by importing the pandas library and creating a DataFrame with two columns, `item` and `purchase_dt`. The purchase dates will be standardized to UTC:

```
df = pd.DataFrame({
    'item': ['item1', 'item2', 'item3', 'item4', 'item5', 'item6'],
    'purchase_dt': pd.date_range('2024-10-01', periods=6,
                                freq='D', tz='UTC')
})
```

The preceding code should generate the following DataFrame with six rows and two columns (`item` and `purchase_dt`):

	item	purchase_dt
0	item1	2024-10-01 00:00:00+00:00
1	item2	2024-10-02 00:00:00+00:00
2	item3	2024-10-03 00:00:00+00:00
3	item4	2024-10-04 00:00:00+00:00
4	item5	2024-10-05 00:00:00+00:00
5	item6	2024-10-06 00:00:00+00:00

Figure 5.4: The DataFrame with the item purchased and purchase datetime (UTC) data

2. Add another datetime column to represent the expiration date, which is 30 days from the purchase date:

```
df['expiration_dt'] = df['purchase_dt'] + pd.Timedelta(days=30)
```

The preceding code adds a third column (`expiration_dt`) to the DataFrame, reflecting dates 30 days after the purchase:

	item	purchase_dt	expiration_dt
0	item1	2024-10-01 00:00:00+00:00	2024-10-31 00:00:00+00:00
1	item2	2024-10-02 00:00:00+00:00	2024-11-01 00:00:00+00:00
2	item3	2024-10-03 00:00:00+00:00	2024-11-02 00:00:00+00:00
3	item4	2024-10-04 00:00:00+00:00	2024-11-03 00:00:00+00:00
4	item5	2024-10-05 00:00:00+00:00	2024-11-04 00:00:00+00:00
5	item6	2024-10-06 00:00:00+00:00	2024-11-05 00:00:00+00:00

Figure 5.5: The updated DataFrame with a third column reflecting the expiration date

3. Now, assume you are asked to create a special extended return policy that adds 35 days, 12 hours, and 30 minutes from the purchase date:

```
df['extended_dt'] = df['purchase_dt'] + \
    pd.Timedelta('35 days 12 hours 30 minutes')
```


The DataFrame now includes a fourth column (`extended_dt`), reflecting the new extended return dates:

	item	purchase_dt	expiration_dt	extended_dt
0	item1	2024-10-01 00:00:00+00:00	2024-10-31 00:00:00+00:00	2024-11-05 12:30:00+00:00
1	item2	2024-10-02 00:00:00+00:00	2024-11-01 00:00:00+00:00	2024-11-06 12:30:00+00:00
2	item3	2024-10-03 00:00:00+00:00	2024-11-02 00:00:00+00:00	2024-11-07 12:30:00+00:00
3	item4	2024-10-04 00:00:00+00:00	2024-11-03 00:00:00+00:00	2024-11-08 12:30:00+00:00
4	item5	2024-10-05 00:00:00+00:00	2024-11-04 00:00:00+00:00	2024-11-09 12:30:00+00:00
5	item6	2024-10-06 00:00:00+00:00	2024-11-05 00:00:00+00:00	2024-11-10 12:30:00+00:00

Figure 5.6: The updated DataFrame with a fourth datetime column reflecting the extended date

- Assume that you are asked to convert the time zone from UTC to the local time zone of the retailer store's headquarters, which is set in Los Angeles:

```
df[['purchase_dt', 'expiration_dt', 'extended_dt']] = \
    df[['purchase_dt', 'expiration_dt', 'extended_dt']].apply(
        lambda x: x.dt.tz_convert('US/Pacific'))
```

After converting from UTC to the US/Pacific time zone (Los Angeles), the datetime columns (`purchase_dt`, `expiration_dt`, and `extended_dt`) are updated with the new localized values. The DataFrame structure should remain the same – six rows and four columns – but now the data looks different, as shown in the following screenshot:

	item	purchase_dt	expiration_dt	extended_dt
0	item1	2024-09-30 17:00:00-07:00	2024-10-30 17:00:00-07:00	2024-11-05 04:30:00-08:00
1	item2	2024-10-01 17:00:00-07:00	2024-10-31 17:00:00-07:00	2024-11-06 04:30:00-08:00
2	item3	2024-10-02 17:00:00-07:00	2024-11-01 17:00:00-07:00	2024-11-07 04:30:00-08:00
3	item4	2024-10-03 17:00:00-07:00	2024-11-02 17:00:00-07:00	2024-11-08 04:30:00-08:00
4	item5	2024-10-04 17:00:00-07:00	2024-11-03 16:00:00-08:00	2024-11-09 04:30:00-08:00
5	item6	2024-10-05 17:00:00-07:00	2024-11-04 16:00:00-08:00	2024-11-10 04:30:00-08:00

Figure 5.7: The updated DataFrame where all datetime columns are in Los Angeles (US/Pacific) time

The `apply()` function ensures that the time zone conversion is applied across all three datetime columns in one step.

- Finally, calculate the time delta between the extended return date and the original expiration date. Since both columns are datetime objects, you can compute this with a simple subtraction:

```
df['exp_ext_diff'] = (df['extended_dt'] - df['expiration_dt'])
df
```

The resulting DataFrame should now have a fifth column, `exp_ext_diff`, which stores the difference between the extended date and the expiration date as a `Timedelta` object:

	item	purchase_dt	expiration_dt	extended_dt	exp_ext_diff
0	item1	2024-09-30 17:00:00-07:00	2024-10-30 17:00:00-07:00	2024-11-05 04:30:00-08:00	5 days 12:30:00
1	item2	2024-10-01 17:00:00-07:00	2024-10-31 17:00:00-07:00	2024-11-06 04:30:00-08:00	5 days 12:30:00
2	item3	2024-10-02 17:00:00-07:00	2024-11-01 17:00:00-07:00	2024-11-07 04:30:00-08:00	5 days 12:30:00
3	item4	2024-10-03 17:00:00-07:00	2024-11-02 17:00:00-07:00	2024-11-08 04:30:00-08:00	5 days 12:30:00
4	item5	2024-10-04 17:00:00-07:00	2024-11-03 16:00:00-08:00	2024-11-09 04:30:00-08:00	5 days 12:30:00
5	item6	2024-10-05 17:00:00-07:00	2024-11-04 16:00:00-08:00	2024-11-10 04:30:00-08:00	5 days 12:30:00

Figure 5.8: The updated DataFrame with a fifth column

Thanks to pandas' built-in capabilities, these types of transformations and calculations are not only simple but also highly efficient, eliminating the need for additional libraries when working with time series.

How it works...

Time deltas are very useful for calculating the difference between two date or time objects. In pandas, the `pandas.Timedelta` class is equivalent to Python's `datetime.timedelta` class, and it behaves very similarly. However, the advantage of pandas is that it includes a wide range of built-in functionality for working with DataFrames and time-series data. Let's dive into a quick experiment to demonstrate how pandas' `Timedelta` class is a subclass of Python's `timedelta` class:

```
import datetime as dt
import pandas as pd
pd.Timedelta(days=1) == dt.timedelta(days=1)
>> True
```

Let's validate that `pandas.Timedelta` is an instance of `datetime.timedelta`:

```
issubclass(pd.Timedelta, dt.timedelta)
>> True
dt_1 = pd.Timedelta(days=1)
dt_2 = dt.timedelta(days=1)
isinstance(dt_1, dt.timedelta)
>> True
isinstance(dt_1, pd.Timedelta)
>> True
```

This confirms that `pandas.Timedelta` inherits from `datetime.timedelta`, making it compatible with Python's `datetime` operations.

Python's `datetime.timedelta` class accepts integer values for these parameters – days, seconds, microseconds, milliseconds, minutes, hours, and weeks. On the other hand, `pandas.Timedelta` takes both integers and strings, as demonstrated in the following snippet:

```
pd.Timedelta(days=1, hours=12, minutes=55)
pd.Timedelta('1 day 12 hours 55 minutes')
pd.Timedelta('1D 12h 55min')
```

All three variations are equivalent and would produce the same result:

```
>> Timedelta('1 days 12:55:00')
```

Once you have defined your `Timedelta` object, you can use it to make calculations on date, time, or `datetime` objects:

```
week_td = pd.Timedelta('1W')
pd.to_datetime('1 JAN 2022') + week_td
>> Timestamp('2022-01-08 00:00:00')
```

In the preceding example, `week_td` represents a one-week `Timedelta` object, which can be added (or subtracted) from `datetime` to get the difference. By adding `week_td`, you are incrementing by one week. What if you want to add two weeks? You can use multiplication as well:

```
pd.to_datetime('1 JAN 2022') + 2*week_td
>> Timestamp('2022-01-15 00:00:00')
```

`pandas.to_timedelta` offers similar functionality to `pandas.Timedelta`. Here are examples that use some of the code tried earlier in the recipe and its equivalence using the `to_timedelta` function:

```
pd.Timedelta(days=30)
pd.to_timedelta('30 days')
>>
Timedelta('30 days 00:00:00')

pd.Timedelta('35 days 12 hours 30 minutes')
pd.to_timedelta('35 days 12 hours 30 minutes')
>>
Timedelta('35 days 12:30:00')

pd.Timedelta(2, unit="days")
pd.to_timedelta(2, unit="days")
>>
Timedelta('2 days 00:00:00')
```



Starting from **pandas 2.2.0**, the units 'H', 'T', 'S', 'L', 'U', and 'N' are deprecated and will be removed in future versions. Instead, pandas now supports the lowercase equivalents: 'h' (hours), 'min' (minutes), 's' (seconds), 'ms' (milliseconds), 'us' (microseconds), and 'ns' (nanoseconds). While this code uses the updated units, you might encounter older code that relies on the deprecated ones.

One major advantage of `to_timedelta` is its ability to process multiple inputs at once, whereas `Timedelta` is designed for creating individual timedelta objects. This makes `to_timedelta` ideal for working with list-like or a Series of time differences:

```
list_of_times = ["30 days", "2 hours", "1 day 4 hours 45 minutes"]
# This would raise an error
pd.Timedelta(list_of_times)
```

With `to_timedelta`, this would work seamlessly:

```
pd.to_timedelta(list_of_times)
>>
TimedeltaIndex(['30 days 00:00:00', '0 days 02:00:00', '1 days 04:45:00'],
dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter in `to_timedelta` allows for handling invalid inputs. The supported arguments include the following:

- `ignore`: Returns unparsed inputs (as is) without raising exceptions
- `raise`: Raises an exception (errors out) for unparsed inputs
- `coerce`: Converts unparsed inputs to `NaT` without raising exceptions

Here is an example to illustrate this:

```
invalid_time = ["1 days", "invalid time"]
pd.to_timedelta(invalid_time, errors="coerce")
>>
TimedeltaIndex(['1 days', NaT], dtype='timedelta64[ns]', freq=None)
```

There's more...

Using pandas' `Timedelta` is straightforward and makes working with large time-series DataFrames efficient without importing additional libraries, as it is built into pandas.

In the previous *How to do it...* section, you created a DataFrame and added additional columns based on the `timedelta` calculations. You can also add the `Timedelta` object to a DataFrame and reference it by its column. Let's see how this works.

First, construct the same DataFrame used earlier:

```
import pandas as pd
df = pd.DataFrame({
    'item': ['item1', 'item2', 'item3', 'item4', 'item5', 'item6'],
    'purchase_dt': pd.date_range('2024-10-01', periods=6, freq='D', tz='UTC')
})
```

This should produce the DataFrame shown in *Figure 5.4*. Now, you will add a new column that contains the `Timedelta` object (one week) and then use that column to add and subtract from the `purchase_dt` column:

```
df['1 week'] = pd.Timedelta('1W')
df['1_week_more'] = df['purchase_dt'] + df['1 week']
df['1_week_less'] = df['purchase_dt'] - df['1 week']
```

The preceding code should produce a DataFrame with three additional columns. The `1 week` column holds the `Timedelta` object, and because of that, you can reference the column to calculate any time differences you need:

	item	purchase_dt	1 week	1_week_more	1_week_less
0	item1	2024-10-01 00:00:00+00:00	7 days	2024-10-08 00:00:00+00:00	2024-09-24 00:00:00+00:00
1	item2	2024-10-02 00:00:00+00:00	7 days	2024-10-09 00:00:00+00:00	2024-09-25 00:00:00+00:00
2	item3	2024-10-03 00:00:00+00:00	7 days	2024-10-10 00:00:00+00:00	2024-09-26 00:00:00+00:00
3	item4	2024-10-04 00:00:00+00:00	7 days	2024-10-11 00:00:00+00:00	2024-09-27 00:00:00+00:00
4	item5	2024-10-05 00:00:00+00:00	7 days	2024-10-12 00:00:00+00:00	2024-09-28 00:00:00+00:00
5	item6	2024-10-06 00:00:00+00:00	7 days	2024-10-13 00:00:00+00:00	2024-09-29 00:00:00+00:00

Figure 5.9: The updated DataFrame with three additional columns

Now, check the data types for each column in the DataFrame:

```
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
---  -
0   item            6 non-null     object
1   purchase_dt     6 non-null     datetime64[ns, UTC]
2   1 week          6 non-null     timedelta64[ns]
3   1_week_more     6 non-null     datetime64[ns, UTC]
4   1_week_less     6 non-null     datetime64[ns, UTC]
dtypes: datetime64[ns, UTC](3), object(1), timedelta64[ns](1)
memory usage: 368.0+ bytes
```

Note that the `1 week` column has the `timedelta64` data type (our `Timedelta` object), which enables arithmetic operations on date, time, and datetime columns within the `DataFrame`.

In the *Working with DatetimeIndex* recipe, you explored the `pandas.date_range()` function, which generates a range of equally spaced time points based on the `start`, `end`, `period`, and `freq` parameters to produce a `DataFrame` with `DatetimeIndex`. Similarly, you can generate a `TimedeltaIndex` with a fixed frequency using the `pandas.timedelta_range()` function, which takes similar parameters to `pandas.date_range()`.

By default, the frequency (`freq`) for `timedelta_range()` is set to `'D'` (days).

Here is a quick example:

The output is as follows:

	item	purchase_dt	time_deltas
0	item1	2024-10-01 00:00:00+00:00	9 days 06:00:00
1	item2	2024-10-02 00:00:00+00:00	10 days 06:00:00
2	item3	2024-10-03 00:00:00+00:00	11 days 06:00:00
3	item4	2024-10-04 00:00:00+00:00	12 days 06:00:00
4	item5	2024-10-05 00:00:00+00:00	13 days 06:00:00

Figure 5.10: A `DataFrame` with a `Timedelta` column

See also

- To learn more about the `pandas.timedelta_range()` function, please refer to the official documentation here: https://pandas.pydata.org/docs/reference/api/pandas.timedelta_range.html.
- To learn more about the `pandas.Timedelta` class, please visit the official documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.Timedelta.html>.

Converting DateTime with time zone information

When working with time-series data that spans multiple time zones, things can quickly become more complex. For example, when developing data pipelines, building a data warehouse, or integrating data between systems, addressing time zones requires careful planning and consensus among project stakeholders. In Python, there are several libraries and modules dedicated to working with time zone conversion, including `pytz`, `dateutil`, and `zoneinfo`, to name a few.

Consider an example that highlights the importance of time zones in time-series data. Large companies that operate across continents often deal with data originating from multiple locations and time zones around the globe. Neglecting time zone differences can undermine effective decision-making. For instance, determining whether most customers visit your e-commerce site in the morning or evening – or whether shoppers browse during the day but make purchases in the evening after work – requires careful consideration of time zone differences. Time zones are critical for ensuring accuracy and relevance in such data-driven business decisions.

How to do it...

In this recipe, you will work with a hypothetical scenario – a small dataset that you will generate to represent website visits at different time intervals from various locations worldwide. The data will be standardized to UTC, and you will work with time zone conversions:

1. Start by importing the pandas library and creating the time-series DataFrame with `visit_dt` as the index, representing visit timestamps:

```
df = pd.DataFrame({
    'Location': ['Los Angeles',
                 'New York',
                 'Berlin',
                 'New Delhi',
                 'Moscow',
                 'Tokyo',
                 'Dubai'],
    'tz': ['US/Pacific',
           'US/Eastern',
           'Europe/Berlin',
           'Asia/Kolkata',
           'Europe/Moscow',
           'Asia/Tokyo',
           'Asia/Dubai'],
    'visit_dt': pd.date_range(start='22:00', periods=7, freq='45min'),
}).set_index('visit_dt')
```

This will produce a DataFrame where `visit_dt` is the index of the `DatetimeIndex` type, with two columns – `Location` (indicating the user's location) and `tz` (representing the corresponding time zone):

	Location	tz
visit_dt		
2025-10-07 22:00:00	Los Angeles	US/Pacific
2025-10-07 22:45:00	New York	US/Eastern
2025-10-07 23:30:00	Berlin	Europe/Berlin
2025-10-08 00:15:00	New Delhi	Asia/Kolkata
2025-10-08 01:00:00	Moscow	Europe/Moscow
2025-10-08 01:45:00	Tokyo	Asia/Tokyo
2025-10-08 02:30:00	Dubai	Asia/Dubai

Figure 5.11: The DataFrame with `visit_dt` in UTC as an index

2. Assume you need to convert the DataFrame to the company's headquarters time zone of Tokyo. If you apply `tz_convert()` directly against the DataFrame, you will get a `TypeError` exception:

```
df.tz_convert('Asia/Tokyo')
>>
TypeError: Cannot convert tz-naive timestamps, use tz_localize to
localize
```

That is because your time-series DataFrame's index is not time zone-aware. You will need to first localize the DataFrame to UTC using `tz_localize()` to make it time zone-aware:

```
df = df.tz_localize('UTC')
```

3. Now, convert the DataFrame's `visit_dt` index to Tokyo's time zone using the `tz_convert()` method:

```
df_hq = df.tz_convert('Asia/Tokyo')
```

The index (`visit_dt`) will now reflect Tokyo's local time:

	Location	tz
visit_dt		
2025-10-08 07:00:00+09:00	Los Angeles	US/Pacific
2025-10-08 07:45:00+09:00	New York	US/Eastern
2025-10-08 08:30:00+09:00	Berlin	Europe/Berlin
2025-10-08 09:15:00+09:00	New Delhi	Asia/Kolkata
2025-10-08 10:00:00+09:00	Moscow	Europe/Moscow
2025-10-08 10:45:00+09:00	Tokyo	Asia/Tokyo
2025-10-08 11:30:00+09:00	Dubai	Asia/Dubai

Figure 5.12: The DataFrame index converted to the Tokyo time zone

Note that you were able to access the `tz_localize()` and `tz_convert()` methods because the DataFrame's index is of type `DatetimeIndex`. If that is not the case, you will get a `TypeError` exception with the following message:

```
TypeError: index is not a valid DatetimeIndex or PeriodIndex
```

4. Next, localize each row to the appropriate time zone and add a new column called `local_dt`, which converts `visit_dt` (in UTC) to the user's local time zone based on the `tz` column:

```
df['local_dt'] = df.index
df['local_dt'] = df.apply(lambda x: pd.Timestamp.tz_convert(
    x['local_dt'], x['tz']), axis=1)
```


This resulting DataFrame will include a new column, `local_dt`, containing the localized time for each visit based on the `tz` column:

	Location	tz	local_dt
visit_dt			
2025-10-07 22:00:00+00:00	Los Angeles	US/Pacific	2025-10-07 15:00:00-07:00
2025-10-07 22:45:00+00:00	New York	US/Eastern	2025-10-07 18:45:00-04:00
2025-10-07 23:30:00+00:00	Berlin	Europe/Berlin	2025-10-08 01:30:00+02:00
2025-10-08 00:15:00+00:00	New Delhi	Asia/Kolkata	2025-10-08 05:45:00+05:30
2025-10-08 01:00:00+00:00	Moscow	Europe/Moscow	2025-10-08 04:00:00+03:00
2025-10-08 01:45:00+00:00	Tokyo	Asia/Tokyo	2025-10-08 10:45:00+09:00
2025-10-08 02:30:00+00:00	Dubai	Asia/Dubai	2025-10-08 06:30:00+04:00

Figure 5.13: The updated DataFrame with `local_dt` based on a localized time zone for each visit

You may wonder, what if we did not have a `tz` column? Where would we get the appropriate `tz` strings? These are called **time zone database names**. These are standard names, and you can find a subset of these in the Python documentation; or, for a more comprehensive list, you can visit this link: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

How it works...

Converting a time-series DataFrame from one time zone to another was achieved using `DataFrame.tz_convert()`, which takes a time zone string argument, such as `US/Pacific`. However, there are a few key assumptions to keep in mind when using `DataFrame.tz_convert()`:

- The DataFrame must have an index of the `DatetimeIndex` type
- `DatetimeIndex` must be time zone-aware

In the previous steps, you used the `DataFrame.tz_localize()` function to make the index time zone-aware. It is generally a good practice to standardize to **UTC** when working with multiple time zones or daylight saving time changes. This is because UTC remains consistent and does not vary based on location or daylight saving time, making it a reliable standard. Once the DataFrame is standardized to UTC, converting to other time zones is very straightforward.

In the example, the data was first localized to UTC and then converted to another time zone (Tokyo) in two steps. However, you can also do this in one step by chaining the two methods, as shown here:

```
df.tz_localize('UTC').tz_convert('Asia/Tokyo')
```

If your index is already time zone-aware, calling `tz_localize()` again will raise a `TypeError` exception with the following message:

```
TypeError: Already tz-aware, use tz_convert to convert
```

This error indicates that you do not need to localize the index again. Instead, you can directly convert it to another time zone using `tz_convert()`.

There's more...

Looking at the DataFrame in *Figure 5.12*, which was localized and converted to the Tokyo time zone, it might not be immediately clear whether the visit times are in the morning (A.M.) or evening (P.M.). To make the DataFrame more user-friendly, you can extract this information and display it in a new column using `strftime`, as discussed in the *Providing a format argument to DateTime* recipe.

You will reconstruct the same DataFrame, localize it to UTC, convert it to the headquarters' time zone, and then add a new column for A.M./P.M. formatting:

```
df = pd.DataFrame({
    'Location': ['Los Angeles',
                'New York',
                'Berlin',
                'New Delhi',
                'Moscow',
                'Tokyo',
                'Dubai'],
    'tz': ['US/Pacific',
          'US/Eastern',
          'Europe/Berlin',
          'Asia/Kolkata',
          'Europe/Moscow',
          'Asia/Tokyo',
          'Asia/Dubai'],
    'visit_dt': pd.date_range(start='22:00', periods=7, freq='45min'),
}).set_index('visit_dt').tz_localize('UTC').tz_convert('Asia/Tokyo')
```

We have combined the steps into one, and this should produce a DataFrame similar to the one in *Figure 5.12*.

Next, create a new column, `am/pm`, that displays either AM or PM, depending on whether the visit occurred in the morning or evening:

```
df['am/pm'] = df.index.strftime('%p')
```

The updated DataFrame should look like the following:

	Location	tz	am/pm
visit_dt			
2025-10-08 07:00:00+09:00	Los Angeles	US/Pacific	AM
2025-10-08 07:45:00+09:00	New York	US/Eastern	AM
2025-10-08 08:30:00+09:00	Berlin	Europe/Berlin	AM
2025-10-08 09:15:00+09:00	New Delhi	Asia/Kolkata	AM
2025-10-08 10:00:00+09:00	Moscow	Europe/Moscow	AM
2025-10-08 10:45:00+09:00	Tokyo	Asia/Tokyo	AM
2025-10-08 11:30:00+09:00	Dubai	Asia/Dubai	AM

Figure 5.14: The updated DataFrame with the am/pm column added

Adding this column makes it easier for users to quickly identify whether a visit occurred in the morning (AM) or evening (PM) at a glance, while retaining the `DatetimeIndex` with its time zone information.

See also

- To learn more about `tz_convert`, you can read the official documentation at https://pandas.pydata.org/docs/reference/api/pandas.Series.dt.tz_convert.html and https://pandas.pydata.org/docs/reference/api/pandas.Timestamp.tz_convert.html.

Working with date offsets

When working with time-series data, it is critical that you understand the context of the data you are working with, especially as it relates to the problem you are attempting to solve. For example, when analyzing manufacturing or sales data, you cannot assume that an organization's working days are Monday to Friday. Similarly, some businesses operate on the fiscal year instead of the standard calendar year. Additionally, you should consider factors such as holiday schedules, annual shutdowns, and other operational considerations.

This is where pandas' `offsets` can be handy. They can help transform your dates, in a calendar-aware way, into formats that are more meaningful and aligned with business requirements. For instance, you can add or subtract specific increments, such as days, months, or years, while adhering to complex calendar rules such as end-of-month adjustments (e.g., adding one month to January 31 results in February 30). Date offsets can also help correct data entries that may not align with the expected logical or business rules. They are particularly useful for business and finance applications, as, for example, an offset such as `BDay` can handle moving dates to the nearest business day.

Unlike `pandas.Timedelta`, `pandas.DateOffset` respects calendar-specific rules such as daylight saving time transitions, making it a perfect choice for time zone-aware operations.

In this recipe, you will work through a hypothetical example to explore how to leverage pandas' date offsets.

How to do it...

In this recipe, you will generate a time-series DataFrame to represent some daily logs of production quantities. The company, a US-based firm, would like to analyze data to better understand production capacity for future forecasting:

1. Start by importing the pandas and numpy libraries and generating a DataFrame with 16 days of random production data:

```
np.random.seed(10)
days = 16
df = pd.DataFrame({
    'prod_date': pd.date_range('2023-12-20', periods=days, freq='D'),
    'production' : np.random.randint(4, 20, days)
}).set_index('prod_date')
```

2. Add a column to represent the day name for each date:

```
df['day_name'] = df.index.day_name()
print(df)
>>
```

	production	day_name
prod_date		
2023-12-20	13	Wednesday
2023-12-21	17	Thursday
2023-12-22	8	Friday
2023-12-23	19	Saturday
2023-12-24	4	Sunday
2023-12-25	5	Monday
2023-12-26	15	Tuesday
2023-12-27	16	Wednesday
2023-12-28	13	Thursday
2023-12-29	17	Friday
2023-12-30	4	Saturday
2023-12-31	17	Sunday
2024-01-01	5	Monday
2024-01-02	14	Tuesday
2024-01-03	12	Wednesday
2024-01-04	13	Thursday

It is a good practice to do your best to understand the business context behind the data you are working with. If you see questionable data points, it would be difficult to determine whether a data point is acceptable or not without domain knowledge or that business context.

For this US-based company, working days are Monday to Friday. Confirm with stakeholders whether weekend production (Saturday and Sunday) reflects exceptions, such as emergencies, and clarify how holidays such as December 25th and January 1st should be treated. The business has requested that weekend production numbers not be included in the forecast and should instead be pushed to the next business day.

3. Handle weekend production by shifting these dates forward to the next business day using `pandas.offsets.BDay()`:
4. `pd.offsets.BDay(0)` ensures the following:
 - a. If the date is already a business day, it remains unchanged.
 - b. If the date falls on a weekend (Saturday or Sunday), it is adjusted forward to the next business day.
 - c. For example, 2023-12-23 (Saturday) was adjusted to 2023-12-25 (Monday). Similarly, 2023-12-24 (Sunday) was adjusted to 2023-12-25 (Monday).
5. Alternatively, you achieve the same result using the `rollforward()` method:

```
df['adj_bus_date'] = df.index + pd.offsets.BDay(0)
df['adj_bus_dname'] = df['adj_bus_date'].dt.day_name()
```

We used `dt.day_name()` to extract the name of the adjusted business day (`adj_day_name`).

6. Group production data by business day and aggregate the value to understand the impact of the adjustments:

```
cols = ['adj_bus_date', 'adj_bus_dname']
df_weekdays = df.groupby(cols)[['production']].sum()
df_weekdays.reset_index(level='adj_bus_dname', inplace=True)
print(df_weekdays)
>>
```

	adj_bus_dname	production
adj_bus_date		
2023-12-20	Wednesday	13
2023-12-21	Thursday	17
2023-12-22	Friday	8
2023-12-25	Monday	28
2023-12-26	Tuesday	15
2023-12-27	Wednesday	16
2023-12-28	Thursday	13
2023-12-29	Friday	17
2024-01-01	Monday	26
2024-01-02	Tuesday	14
2024-01-03	Wednesday	12
2024-01-04	Thursday	13

Now, the `df_weekdays` DataFrame excludes weekends (Saturday or Sunday) and aggregates values for the adjusted business days. Weekend production values are aggregated into their corresponding Monday's total after being shifted forward.

7. Finally, the business has made another request – they want to support higher-level reporting requirements, such as monthly or quarterly summaries. Add columns for `month_end` and `quarter_end` dates to align production data with reporting periods using pandas offsets:

```
df_weekdays['quarter_end'] = (df_weekdays.index +
                               pd.offsets.QuarterEnd())
df_weekdays['month_end'] = (df_weekdays.index +
                             pd.offsets.MonthEnd())
print(df_weekdays)
>>
```

	adj_bus_dname	production	quarter_end	month_end
adj_bus_date				
2023-12-20	Wednesday	13	2023-12-31	2023-12-31
2023-12-21	Thursday	17	2023-12-31	2023-12-31
2023-12-22	Friday	8	2023-12-31	2023-12-31
2023-12-25	Monday	28	2023-12-31	2023-12-31
2023-12-26	Tuesday	15	2023-12-31	2023-12-31
2023-12-27	Wednesday	16	2023-12-31	2023-12-31
2023-12-28	Thursday	13	2023-12-31	2023-12-31
2023-12-29	Friday	17	2023-12-31	2023-12-31
2024-01-01	Monday	26	2024-03-31	2024-01-31
2024-01-02	Tuesday	14	2024-03-31	2024-01-31
2024-01-03	Wednesday	12	2024-03-31	2024-01-31
2024-01-04	Thursday	13	2024-03-31	2024-01-31

Now, you have a DataFrame that should satisfy most of the reporting requirements of the business.

How it works...

Using date offsets makes it possible to increment, decrement, and transform your dates to new ranges following specific rules. There are several built-in offsets provided by pandas, each with its own behavior and use case, that can be applied to your dataset. Here are common offsets available in pandas:

- **BusinessDay** or **Bday**: Represents business days (Monday through Friday) by default, skipping weekends. It is used to adjust dates by adding or subtracting business days.
- **MonthBegin**: Rolls dates to the first calendar day of the month.
- **MonthEnd**: Rolls dates to the last calendar day of the month.
- **BusinessMonthBegin** or **BmonthBegin**: Adjusts dates to the first business day of the month.
- **BusinessMonthEnd** or **BmonthEnd**: Adjusts dates to the last business day of the month.
- **CustomBusinessDay** or **CDay**: Represents custom-defined business days (e.g., excluding holidays).

- `QuarterBegin`: Adjusts dates to the first day of the quarter.
- `QuarterEnd`: Adjusts dates to the last day of the quarter.
- `FY5253Quarter`: Adjusts dates based on a 4-4-5 fiscal quarter calendar.

For a more comprehensive list and their descriptions, you can visit the documentation here: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects.

Applying an offset in pandas is as simple as doing an addition or subtraction, as shown in the following example:

```
df.index + pd.offsets.BDay()  
df.index - pd.offsets.BDay()
```

There's more...

Following our example, you may have noticed that using the `BusinessDay` (`BDay`) offset does not account for holidays. So, what can be done to adjust for holidays such as Christmas and New Year's Day?

There are two approaches to handle holidays in pandas:

- Use an existing holiday offset
- Define a custom holiday calendar

Using an existing holiday offset

For this example, you can use the `USFederalHolidayCalendar` class, which includes standard US holidays such as New Year, Christmas, and Thanksgiving. So, let's see how this works.

First, import the additional modules needed and make a copy of the `df_weekdays` DataFrame from the previous section:

```
from pandas.tseries.holiday import (  
    USFederalHolidayCalendar,  
    AbstractHolidayCalendar,  
    Holiday,  
    nearest_workday  
)  
from pandas.tseries.offsets import CDay  
df = df_weekdays[['adj_bus_dname', 'production']].copy()
```

The `USFederalHolidayCalendar` class contains built-in holiday rules that you can inspect:

```
USFederalHolidayCalendar.rules
```

The output includes a total of 11 holiday definitions and rules.

To apply these rules, use the `CustomerBusinessDay` or `CDay` offset. We already imported `CDay`, which is what we will use:

```
def holiday_offset(date):
    us_holiday_offset = CDay(calendar=USFederalHolidayCalendar())
    return us_holiday_offset.rollforward(date)

df['adj_holiday_date'] = df.index.to_series().map(holiday_offset)
df['adj_holiday_dname'] = df['adj_holiday_date'].dt.day_name()
print(df)

>>
      adj_bus_dname  production adj_holiday_date adj_holiday_dname
adj_bus_date
2023-12-20      Wednesday          13      2023-12-20      Wednesday
2023-12-21      Thursday          17      2023-12-21      Thursday
2023-12-22        Friday           8      2023-12-22        Friday
2023-12-25        Monday          28      2023-12-26        Tuesday
2023-12-26        Tuesday          15      2023-12-26        Tuesday
2023-12-27      Wednesday          16      2023-12-27      Wednesday
2023-12-28      Thursday          13      2023-12-28      Thursday
2023-12-29        Friday          17      2023-12-29        Friday
2024-01-01        Monday          26      2024-01-02        Tuesday
2024-01-02        Tuesday          14      2024-01-02        Tuesday
2024-01-03      Wednesday          12      2024-01-03      Wednesday
2024-01-04      Thursday          13      2024-01-04      Thursday

df['USFederalHolidays'] = df.index + pd.offsets.CDay(
    calendar=USFederalHolidayCalendar())
df
```

The `adj_holiday_date` column adjusts each date in the DataFrame to account for both weekends and US federal holidays by shifting them forward to the next valid business day.

Defining a custom holiday calendar

For businesses with unique holiday schedules, you can define a custom holiday calendar using `AbstractHolidayCalendar` and the `Holiday` classes, as well as the `nearest_workday` function. Start by defining a custom class, `MyHolidayCalendar`:

```
class MyHolidayCalendar(AbstractHolidayCalendar):
    rules = [
        Holiday("Cust New Years", month=1, day=1,
                observance=nearest_workday),
        Holiday("Cust Christmass Day", month=12, day=25,
                observance=nearest_workday)
    ]

mycal = MyHolidayCalendar()
```


Similar to how you applied the `USFederalHolidayCalendar` class using the `CDay` offset, you will apply your custom objects. The `custom_holiday_offset` function will be similar to the `holiday_offset` function used earlier:

```
def custom_holiday_offset(date):
    cust_holiday_offset = CDay(calendar=mycal)
    return cust_holiday_offset.rollforward(date)

df['cust_holiday_date'] = df.index.to_series().map(custom_holiday_offset)
df['cust_holiday_dname'] = df['cust_holiday_date'].dt.day_name()
```

You will get the following output:

	adj_bus_dname	production	adj_holiday_date	adj_holiday_dname	cust_holiday_date	cust_holiday_dname
adj_bus_date						
2023-12-20	Wednesday	13	2023-12-20	Wednesday	2023-12-20	Wednesday
2023-12-21	Thursday	17	2023-12-21	Thursday	2023-12-21	Thursday
2023-12-22	Friday	8	2023-12-22	Friday	2023-12-22	Friday
2023-12-25	Monday	28	2023-12-26	Tuesday	2023-12-26	Tuesday
2023-12-26	Tuesday	15	2023-12-26	Tuesday	2023-12-26	Tuesday
2023-12-27	Wednesday	16	2023-12-27	Wednesday	2023-12-27	Wednesday
2023-12-28	Thursday	13	2023-12-28	Thursday	2023-12-28	Thursday
2023-12-29	Friday	17	2023-12-29	Friday	2023-12-29	Friday
2024-01-01	Monday	26	2024-01-02	Tuesday	2024-01-02	Tuesday
2024-01-02	Tuesday	14	2024-01-02	Tuesday	2024-01-02	Tuesday
2024-01-03	Wednesday	12	2024-01-03	Wednesday	2024-01-03	Wednesday
2024-01-04	Thursday	13	2024-01-04	Thursday	2024-01-04	Thursday

Figure 5.15: The updated DataFrame with adjusted holiday columns

Now, the `cust_holiday_date` column adjusts dates based on the custom holiday rules.

If a holiday such as Christmas falls on a weekend, the holiday date will be observed on the nearest weekday (Friday or Monday), as determined by the `nearest_workday` rule.

If you are curious about the `nearest_workday` function and how it was used in both the `USFederalHolidayCalendar` rules and your custom holiday, then the following code illustrates how it works:

```
nearest_workday(pd.to_datetime('2024-11-30'))
>>
Timestamp('2024-11-29 00:00:00')

nearest_workday(pd.to_datetime('2024-12-1'))
>>
Timestamp('2024-12-02 00:00:00')
```

As illustrated, the `nearest_workday` function finds the nearest weekday (Friday or Monday) if a holiday falls on a weekend. It determines whether the day is a weekday or not, and based on that, it will either use the day before (if it falls on a Saturday) or the day after (if it falls on a Sunday). There are other rules available as well, including the following:

- `next_Monday`: Shifts dates forward to the next Monday
- `next_monday_or_Tuesday`: Shifts dates forward to the next Monday or Tuesday
- `next_workday`: Moves forward to the next valid workday
- `previous_Friday`: Moves backward to the previous Friday
- `previous_workday`: Moves backward to the previous valid workday
- `sunday_to_Monday`: Specifically shifts Sundays to Mondays

`USFederalHolidayCalendar` is useful for standard US holidays such as New Year's Day or Christmas. For businesses with unique holiday schedules, defining a custom calendar using `AbstractHolidayCalendar` allows flexibility to include specific rules.

See also

- For more insight regarding `pandas.tseries.holiday`, you can view the actual code, which highlights all the classes and functions and can serve as an excellent reference, at <https://github.com/pandas-dev/pandas/blob/master/pandas/tseries/holiday.py>.

Working with custom business days

Companies have different working days worldwide, influenced by the region or territory they belong to. For example, when working with time-series data, and depending on the analysis you need to make, knowing whether certain transactions fall on a workday or weekend can make a difference. For instance, if you are performing anomaly detection and know that certain types of activities can only be done during working hours, any activities beyond these boundaries may trigger further analysis.

In this recipe, you will see how you can customize an offset to fit your requirements when doing an analysis that depends on defined business days and non-business days.

How to do it...

In this recipe, you will create custom business days and holidays for a company headquartered in Amman, Jordan. In Jordan, the working week is from Sunday to Thursday, whereas Friday and Saturday are weekends. Additionally, Labor Day is on May 1 and Independence Day is on May 25 of each year:

1. You will start by importing `pandas` and defining the workdays for Jordan:

```
import pandas as pd
from pandas.tseries.holiday import AbstractHolidayCalendar, Holiday
from pandas.tseries.offsets import CustomBusinessDay
```

- You will need a custom class, `JordanHolidayCalendar`, that contains a list of rules for Jordan's Independence Day and Labor Day holidays:

```
class JordanHolidayCalendar(AbstractHolidayCalendar):
    rules = [
        Holiday('Jordan Independence Day', month=5, day=25),
        Holiday('Jordan Labor Day', month=5, day=1)
    ]
```

- You can use the new class to return a list of holiday occurrences within a defined date range. For example, you can check for 2024 and 2025:

```
JordanHolidayCalendar().holidays(start='2024-01-01',
                                   end='2025-12-31')

>>
DatetimeIndex(['2024-05-01', '2024-05-25', '2025-05-01', '2025-05-25'],
              dtype='datetime64[ns]', freq=None)
```

Keep in mind that `JordanHolidayCalendar().holidays()` without a date range would by default return all the holidays from 1970 to 2200. This default range ensures compatibility with any potential future extensions of your analysis:

```
JordanHolidayCalendar().holidays()

>>
DatetimeIndex([
    '1970-05-01', '1970-05-25', '1971-05-01', '1971-05-25',
    '1972-05-01', '1972-05-25', '1973-05-01', '1973-05-25',
    '1974-05-01', '1974-05-25',
    ...
    '2196-05-01', '2196-05-25', '2197-05-01', '2197-05-25',
    '2198-05-01', '2198-05-25', '2199-05-01', '2199-05-25',
    '2200-05-01', '2200-05-25'],
              dtype='datetime64[ns]', length=462, freq=None)
```

The list contains 462 entries. The list could be much bigger if we had defined more holidays. Thus, it is a good practice when using `holidays()` to define a start and end range.

- Next, define a `weekmask` to represent the working days as a string of date abbreviations:

```
jo_mask = "Sun Mon Tue Wed Thu"
```

- The definitions specify that the working days are Sunday to Thursday. Alternatively, you can define it as a string of binary values (1 for business day, 0 for weekend):

```
jo_mask = "1111100"
```

6. Define a new instance of CustomBusinessDay with Jordan's working days and custom holidays:

```
jo_busdays = CustomBusinessDay(
    holidays=JordanHolidayCalendar().holidays(start='2024-01-01',
                                                end='2025-12-31'),
    weekmask=jo_mask
)
```

The start and end for holidays() are added to reduce the list of holiday occurrences to be from 2024 to 2025. Even though in our recipe we will only be generating date ranges for 2024, this illustrates that the purpose of holidays is just defining a list of occurrences that will be used as a reference when generating dates.

7. You can validate that the rules were registered properly:

```
JordanHolidayCalendar().rules
>>
[Holiday: Jordan Independence Day (month=5, day=25, ),
 Holiday: Jordan Labor Day (month=5, day=1, )]
```

8. You can also check the weekmask:

```
jo_busdays.weekmask
>>
'Sun Mon Tue Wed Thu'
```

9. Check the two rules:

```
JordanHolidayCalendar().rules[0].dates('2024-01-01', '2024-12-31')
>>
DatetimeIndex(['2024-05-25'], dtype='datetime64[ns]', freq='<DateOffset:
years=1>')

JordanHolidayCalendar().rules[1].dates('2024-01-01', '2024-12-31')
>>
DatetimeIndex(['2024-05-01'], dtype='datetime64[ns]', freq='<DateOffset:
years=1>')
```

10. Now you can use pd.date_range to generate 20 **business days** starting from May 1st, 2024. The generated dates will exclude weekends (Friday and Saturday) and the two national holidays:

```
days = 20
df = pd.DataFrame({'Date': pd.date_range(
    '2024-5-1', periods=days, freq=jo_busdays)})
```

11. To make it easier to determine whether things are working as expected, add a new column that represents the day name:

```
df['Day_name'] = df.Date.dt.day_name()
print(df)
```

12. The generated time series should have the new custom business days and a holiday rule for Jordan:

	Date	Day_name
0	2024-05-02	Thursday
1	2024-05-05	Sunday
2	2024-05-06	Monday
3	2024-05-07	Tuesday
4	2024-05-08	Wednesday
5	2024-05-09	Thursday
6	2024-05-12	Sunday
7	2024-05-13	Monday
8	2024-05-14	Tuesday
9	2024-05-15	Wednesday
10	2024-05-16	Thursday
11	2024-05-19	Sunday
12	2024-05-20	Monday
13	2024-05-21	Tuesday
14	2024-05-22	Wednesday
15	2024-05-23	Thursday
16	2024-05-26	Sunday
17	2024-05-27	Monday
18	2024-05-28	Tuesday
19	2024-05-29	Wednesday

The example can be further extended to include different countries and holidays to fit the type of analysis you are working with. To extend this example to other countries, define additional rules in the `AbstractHolidayCalendar` class or create separate calendars for each country.

How it works...

This recipe builds on the *Working with date offsets recipe* but focuses on customizing offsets. The pandas library provides several offsets that can take a custom calendar, holiday, and weekmask. These include the following:

- `CustomBusinessDay` or `Cday`: Customizes business days by defining specific holidays and work-weeks
- `CustomBusinessMonthEnd` or `CBMonthEnd`: Rolls forward or backward to the last business day of the month based on custom rules

- `CustomBusinessMonthBegin` or `CBMonthBegin`: Rolls forward or backward to the first business day of the month based on custom rules
- `CustomBusinessHour`: Defines custom business hours within specified workdays

They behave like any other offset; the only difference is that they allow you to create your own rules. For instance, if a business operates on a non-standard workweek or observes unique holidays, using a custom offset ensures that date calculations align with actual business operations.

In the recipe, you imported the `CustomBusinessDay` class and created an instance of it to create a custom frequency for business days, taking into account weekends and holidays. The following code is what you used:

```
jo_busdays = CustomBusinessDay(  
    holidays=JordanHolidayCalendar().holidays(start='2024-01-01',  
                                                end='2025-12-31'),  
    weekmask=jo_mask  
)
```

In our example, when defining workdays, you used a list of abbreviated weekday names. This is called `weekmask`, and it's used in both pandas and NumPy when customizing weekdays. `weekmask` defines the working days using abbreviated weekday names (e.g., 'Sun Mon Tue Wed Thu' for Sunday through Thursday). This ensures that weekends (Friday and Saturday) are excluded.

In pandas, you can also define a custom holiday calendar by extending the `AbstractHolidayCalendar` class and including specific dates or rules for holidays, as demonstrated in the earlier code for Jordan Independence Day.

There's more...

Let's extend the previous example by adding custom business hours to the `DataFrame`. This involves creating another custom offset, similar to `CustomBusinessDay`, but tailored to include specific business hours:

```
jo_busshours = pd.offsets.CustomBusinessHour(  
    start="7:30",  
    end="16:30",  
    holidays=JordanHolidayCalendar().holidays(start='2024-01-01',  
                                                end='2025-12-31'),  
    weekmask=jo_mask)
```

Here, you are applying the same rules: the custom holidays and `weekmask` for the custom workdays, ensuring that the custom business hours respect the defined workweek and holidays. The working hours are defined by providing the **start** and **end** times (in 24-hour format).

The following is a usage example of the custom business hours you just created to generate a range of timestamps during business hours within a specified date range:

```
start = '2024-05-1'
end = '2024-05-29'

business_hours = pd.date_range(start=start, end=end, freq=jo_busshours)
print(business_hours[0:10])
>>
DatetimeIndex(['2024-05-02 07:30:00', '2024-05-02 08:30:00',
               '2024-05-02 09:30:00', '2024-05-02 10:30:00',
               '2024-05-02 11:30:00', '2024-05-02 12:30:00',
               '2024-05-02 13:30:00', '2024-05-02 14:30:00',
               '2024-05-02 15:30:00', '2024-05-05 07:30:00'],
              dtype='datetime64[ns]', freq='cbh')
```

Note that custom offsets such as `CustomBusinessDay` and `CustomBusinessHour` can be applied to both time zone-naïve and time zone-aware datetime objects. However, making a `DataFrame` time zone-aware depends on the specific use case. For scenarios where time zones are critical – such as global analysis or daylight saving changes – localizing your `DataFrame` to a specific time zone (`tz.localize()`) and then converting it (`tz.convert()`) can help ensure the accuracy of time-based calculations. By doing this, you can avoid errors associated with time zone differences and get consistent and reliable results.

```
# Localize to UTC
df['Date'] = df['Date'].dt.tz_localize('UTC')

# Convert to another time zone (e.g., Asia/Amman)
df['Date'] = df['Date'].dt.tz_convert('Asia/Amman')
```

For more information on working with different time zones, refer to the *Converting DateTime with time zone information* recipe.

See also

- To learn more about pandas' `CustomBusinessDay`, you can read the official documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.tseries.offsets.CustomBusinessDay.html>.
- To learn more about pandas' `CustomBusinessHours`, you can read the official documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.tseries.offsets.CustomBusinessHour.html>.

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



6

Handling Missing Data

As a data scientist, data analyst, or business professional working with datasets, you may have already realized that expecting a perfectly clean dataset is overly optimistic. More often than not, the data you work with suffers from flaws such as **missing values**, **erroneous values**, **duplicate records**, **insufficient samples**, or the presence of **outliers**.

Time series data is no different. Before diving into any analysis or modeling workflow, it is crucial to investigate and understand your dataset. Spotting these issues early is essential to ensure accurate analysis and reliable model performance. Moreover, understanding the **business context** around your time series data is critical for identifying and addressing data issues successfully. Here are some examples:

- Stock market data has different patterns, assumptions, and potential sources for missing data compared to COVID-19 tracking data or sensor readings
- COVID-19 tracking data (or other infectious diseases) may have inconsistencies due to reporting delays or differences across regions
- Domain knowledge helps you differentiate between acceptable irregularities (e.g., missing weekend stock prices) and actual data quality issues (e.g., missing weekday stock prices)

Having this type of intuition or domain knowledge will allow you to anticipate potential issues and define what would be considered acceptable or problematic. Always aim to understand the context around the data. Ask questions such as the following:

- Why was the data collected in the first place?
- How was the data collected?
- What business rules, logic, or transformations have been applied to the data?
- Were these modifications applied during the data acquisition process or within the systems that generated the data?

Two common challenges during data preparation are **handling missing values** and **dealing with outliers**. You will dive into outlier detection in *Chapter 7*, and *Chapter 13*. In this chapter, you will focus on missing data and explore techniques for handling missing data through **imputation** and **interpolation**.

Here is the list of recipes that we will cover in this chapter:

- Performing data quality checks
- Handling missing data with univariate imputation using pandas
- Handling missing data with univariate imputation using scikit-learn
- Handling missing data with multivariate imputation
- Handling missing data with interpolation

Technical requirements

To follow along with this chapter, you can download the Jupyter notebook and necessary datasets from the GitHub repository:

- **Jupyter notebooks:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch6/Chapter_6.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch6>

This chapter extensively uses pandas 2.3.3 (released September 29, 2025), which is the latest version as of this writing. You can use a more recent version if available. Additionally, the following libraries will be utilized:

- numpy = 2.3.3
- matplotlib = 3.10.6
- statsmodels = 0.14.5
- scikit-learn = 1.7.2
- SciPy = 1.16.2

If you are using **pip**, then you can install these packages from your terminal with the following command:

```
pip install matplotlib numpy statsmodels scikit-learn scipy
```

In this chapter, two datasets will be used extensively for the imputation and interpolation recipes:

- **CO2 Emissions** dataset: Annual CO2 emissions data (1750-2020) from **Our World in Data** (<https://ourworldindata.org/co2-emissions>)
- **e-Shop Clickstream** dataset: Daily clickstream data for an online clothing store from April 2008 to August 2008. That data is from the **UCI machine learning repository** (<https://archive.ics.uci.edu/dataset/553/clickstream+data+for+online+shopping>)

The CO2 emissions dataset, with its long-term trend, is ideal for testing imputation and interpolation over decades. On the other hand, the e-Shop clickstream data, with its daily fluctuations, is perfect for exploring imputation in fast-moving, behavioral time series.

For demonstration purposes, both datasets have been modified to include missing observations (by removing observations). The original versions are also provided for comparison when evaluating the different techniques.

Throughout this chapter, you will follow a structured process for handling missing data:

1. **Ingest** the data into a pandas DataFrame.
2. **Identify** and analyze missing data.
3. **Impute** missing values.
4. **Evaluate** imputed values against the original dataset.
5. **Visualize** and compare the performance of different imputation methods.

These steps can be translated into reusable functions to streamline the process. You will create three key utility functions to assist in the process:

- A `read_dataset` function to read the CSV file into a DataFrame and parse a specified date column
- A `rmse_score` function to evaluate results by computing the **RMSE** score for each imputation method and comparing it to the original data
- A `plot_dfs` function to compare the original dataset with the imputed version visually

Start by loading the standard libraries that you will be using throughout this chapter:

```
import pandas as pd
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
```

For convenience and cleaner code, the helper functions used in this chapter are stored in a separate `utils.py` script. You can import them as follows:

```
from utils import read_dataset, plot_dfs, rmse_score
```

These utility functions handle three essential tasks in this chapter:

- `read_dataset` loads a CSV file, parses the date column, and prepares the dataset for analysis.
- `plot_dfs` visualizes and compares the original and imputed versions of a time series, enabling a quick visual assessment of imputation performance.
- `rmse_score` computes the **Root Mean Squared Error (RMSE)** between the original and imputed datasets, allowing a quantitative comparison across different techniques.

Each of these functions will be used throughout this chapter to simplify the workflow and ensure consistency when visualizing and evaluating imputation methods.

Understanding missing data

Missing data can occur due to various reasons, such as unexpected power outages, accidentally unplugged devices, defective sensors, survey respondents declining to answer a question, or intentional removal for privacy or compliance reasons. In other words, missing data is inevitable in real-world datasets.

Generally, missing data is very common, yet sometimes it is not given the proper level of attention in terms of formulating a strategy on how to handle the situation. Improper handling of missing data can lead to biased analysis, unreliable models, and incorrect conclusions. Common approaches to handling missing data may include the following:

- **Dropping rows with missing data:** This is a straightforward approach where observations with missing data are deleted. However, this can lead to the loss of significant information. Things can be worse if you have limited data in the first place – for example, if collecting the data is a complex and expensive process. Additionally, the drawback of deleting records prematurely is the loss of potential patterns in missing data. You will not be able to know if the missing data was due to **censoring** (an observation is only partially collected) or due to **bias** (e.g., high-income participants declining to share their total household income in a survey).
- **Tagging missing data:** Another approach may involve tagging the rows with missing data by adding a new column describing or labeling the missing data. For example, suppose you know that there was a power outage on a particular day. In that case, you can add “*Power Outage*” rather than “*Missing Data*” when the cause is unknown. This approach adds additional context and allows for further analysis. Additionally, by differentiating between missing data causes, it can also inform downstream workflows.
- **Estimating missing values (imputation):** Imputing missing values is the focus of this chapter. The goal of imputation is to substitute for the missing values (fill in). The techniques for estimating missing values can range from simple methods (e.g., mean or median imputation) to more advanced models (e.g., statistical or machine learning methods). The challenge here is that since the true values are missing, it can be difficult to determine the accuracy of the imputed values.

In this chapter, you will use RMSE as a metric to evaluate the different imputation techniques.

The process to calculate the RMSE can be broken down into a few simple steps:

1. **Calculate the error for each observation:** This is the difference between the actual (original) value and the imputed (estimated) value.
2. **Square the errors to avoid negative values:** Errors can be both negative and positive, which may cancel each other out when summed. To avoid having a zero summation, we square the errors (differences).
3. **Compute the Mean Squared Error (MSE):** We sum up all the errors and divide by the total number of observations to compute the mean.
4. **Take the square root of the MSE to get the RMSE.**

The RMSE equation can be written as:

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (x_i - \hat{x}_i)^2}{N}}$$

Where $\hat{x}_i$ is the imputed value, x_i is the actual (original) value, and N is the number of observations.



Understanding RMSE limitations in imputation

RMSE is a commonly used metric to evaluate the performance of *predictive* models (e.g., comparing different regression models). Generally, a *lower* RMSE is desirable; it tells us that the model can fit the dataset. Simply stated, it tells us the average distance (error) between the predicted value and the actual value. You want this distance minimized.

However, when comparing different imputation methods, we want our imputed values to resemble, as closely as possible, the actual data, which contains random effects (uncertainty). This means we are not seeking a perfect prediction, and thus, a lower RMSE score does not necessarily indicate a better imputation method, as it might overlook the randomness and variability inherent in the original dataset. In this chapter, the use of RMSE is combined with visualization to help illustrate how the different techniques compare and work.

As a reminder, we have intentionally removed some values (synthetic missing values) but retained the original data (ground truth) to compare against when using RMSE.

Performing data quality checks

Missing data refers to values that were not captured or observed in the dataset. These missing values can occur in specific features (columns) or across entire observations (rows). When ingesting data using pandas, missing values are typically represented as NaN (**Not a Number**), NaT (**Not a Time**), or NA.

In some cases, datasets may include placeholder values from the source system instead of leaving the cells empty to represent missing observations. For example, you may encounter numeric placeholders such as 99999 or 0, or strings like "missing", "?", or "N/A". It is important to investigate placeholder values carefully. For instance, when missing values are represented by zeros (0), you need to be cautious and investigate further to determine whether those zeros represent legitimate values or are indicative of missing data (placeholders).

In this recipe, you will explore how to identify the presence of missing data.

Getting ready...

You can download the Jupyter notebooks and requisite datasets from the GitHub repository. Please refer to the *Technical requirements* section of this chapter.

You will be using two datasets from the Ch6 folder:

- `clicks_missing_multiple.csv`
- `co2_missing.csv`

How to do it...

The pandas library provides convenient methods for discovering missing data and for summarizing data in a DataFrame:

1. Start by reading the two CSV files (`co2_missing.csv` and `clicks_missing_multiple.csv`) using the `read_dataset()` function:

```
folder = Path('../..../datasets/Ch6/')
co2_file = Path('co2_missing.csv')
ecom_file = Path('clicks_missing_multiple.csv')

co2_df = read_dataset(folder,
                      co2_file,
                      index=True,
                      date_col='year')

ecom_df = read_dataset(folder,
                      ecom_file,
                      index=True,
                      date_col='date')

ecom_df.head()
```

This should display the first five rows from the `ecom_df` DataFrame:



	price	location	clicks
date			
2008-04-01	43.155647	2.0	18784
2008-04-02	43.079056	1.0	24738
NaT	43.842609	NaN	15209
NaT	NaN	1.0	14018
NaT	43.941176	1.0	11974

Figure 6.1: First five rows from the `ecom_df` DataFrame showing `NaN` and `NaT`

The output from the preceding code shows that there are five missing values from the source dataset. `NaN` is how pandas represents empty **numeric** values. `NaT` is how pandas represents missing **Datetime** values.

2. To count the number of missing values in both DataFrames, you can use the `DataFrame.isnull()` or `DataFrame.isna()` methods. Both methods will return `True` (if missing) or `False` (if not missing) for each value. To **count** missing values for each column, you can use `DataFrame.isnull().sum()` or `DataFrame.isna().sum()`.

3. The reason `.sum()` works for counting missing values is that in Python, Booleans (`True` or `False`) are a subtype of integers. `True` is equivalent to `1`, and `False` is equivalent to `0`. To validate this concept, try the following:

```
isinstance(True, int)
>> True
int(True)
>> 1
int(False)
>> 0
```

This allows us to use `.sum()` to count how many `True` values there are (which represent missing values) in our Boolean mask created by `isnull()`.

Now, let's get the total number of missing values for each DataFrame:

```
co2_df.isnull().sum()
>>
co2      25
dtype: int64

ecom_df.isnull().sum()
>>
price      1
location   1
clicks     14
dtype: int64
```

You can use either `isnull().sum()` or `isna().sum()` since these methods can be used interchangeably. The `isnull()` method is simply an alias of `isna()`.

4. In the previous step, you may have noticed that the `year` column from `co2_df` and the `date` column from `ecom_df` were not included in the result set when counting missing values. This is because when you call `isnull()` or `isna()` methods on a DataFrame, they only check for missing values in the DataFrame's columns and not in the index. Since our `read_dataset()` function had set the `year` and `date` columns as the DataFrame **index** (specifically, `DatetimeIndex`), any missing values in the index are ignored by default. If you want to include the index in your missing value checks, you have a couple of options: Use `.index.isnull()` or `.index.isna()` to check missing values directly in the index. Alternatively, you can reset the index to return it to a regular column so it can be included in `.isnull().sum()`, as shown here:

```
co2_df.reset_index(inplace=True)
ecom_df.reset_index(inplace=True)
```


Now, if you run `isnull().sum()`, you should see the year column from `co2_df` and the date column from `ecom_df` included in the counts, giving you a complete picture of missing values across your entire DataFrame:

```
co2_df.isnull().sum()
>>
year      0
co2      25
dtype: int64

ecom_df.isnull().sum()
>>
date       4
price      1
location    1
clicks     14
dtype: int64
```

From the results, `co2_df` has 25 missing values in the `co2` column, while `ecom_df` has 20 missing values in total (4 from the `date` column, 1 from the `price` column, 1 from the `location` column, and 14 from the `clicks` column).

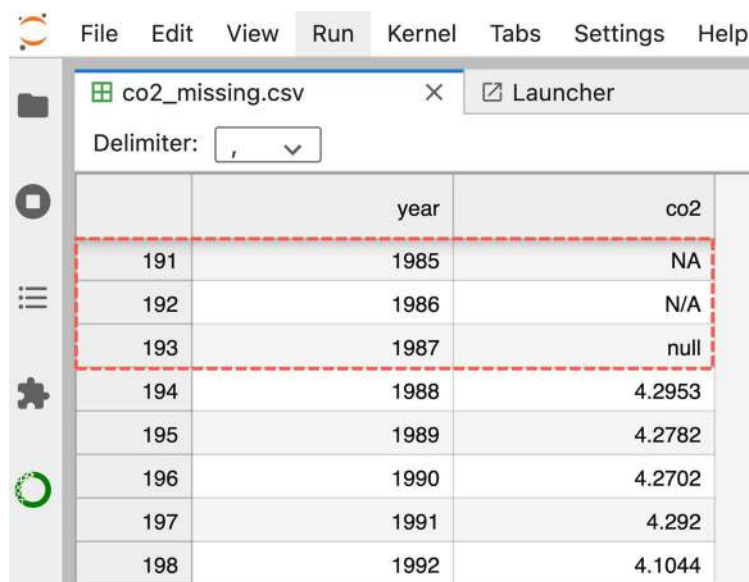
5. To get the total number of missing values in a DataFrame, simply chain another `sum()` function to the end of the statement:

```
print(ecom_df.isnull().sum().sum())
>> 20
```

Similarly, for `co2_df`, you can chain another `sum()`:

```
print(co2_df.isnull().sum().sum())
>> 25
```

6. If you inspect the `co2_missing.csv` file using a text/code editor (such as Excel, VSCode, or JupyterLab) and scroll down to rows 191–193, you will notice that there are string placeholder values in there: NA, N/A, and null:



	year	co2
191	1985	NA
192	1986	N/A
193	1987	null
194	1988	4.2953
195	1989	4.2782
196	1990	4.2702
197	1991	4.292
198	1992	4.1044

Figure 6.2: `co2_missing.csv` shows string values that were converted to NaN (missing) by `pandas`

Figure 6.2 shows the three string values. Interestingly, `pandas.read_csv()` automatically interpreted the three string values as NaN. This is the default behavior in `read_csv()`, which can be modified using the `na_values` parameter. To see how `pandas` represents these values, you can run the following command:

```
print(co2_df[190:195])
>>
      year      co2
190 1985-01-01  NaN
191 1986-01-01  NaN
192 1987-01-01  NaN
193 1988-01-01  4.2953
194 1989-01-01  4.2782
```

Note that in the CSV view (Figure 6.2), the first row starts from 1, while in `pandas`, the first row starts from zero (0). Hence, the rows 191, 192, and 193 in `pandas` are represented by the indices 190, 191, and 192, respectively.

7. If all you need is to check whether the DataFrame contains any missing values, use `isnull().values.any()`. This will output `True` if there are any missing values in the DataFrame:

```
print(ecom_df.isnull().values.any())
>> True

print(co2_df.isnull().values.any())
>> True
```

8. So far, `isnull()` has helped identify all the missing values in the DataFrames. But what if the missing values were masked or replaced by other placeholder values, such as `?`, `0`, or `99999`. These placeholder values are not inherently treated as missing (`NaN`) in pandas since they technically hold data. However, domain knowledge or prior insights can help us identify these placeholders as missing data. For example, in the CO2 emission dataset, annual emissions should always be greater than 0, so a value of `0` would indicate missing data.
9. Similarly, in the clickstream dataset, the number of clicks should be numeric. If the column is not numeric, it should prompt an investigation as to why pandas could not parse the column as such. This issue often arises when string values (e.g., `?`) are present in the column.

To gain deeper insight into the schema and data types of the DataFrame, you can use the `DataFrame.info()` method. This method displays the schema, total records, column names, data types (dtypes), count of non-missing values per column, the index type, and the DataFrame's total memory usage:

```
ecom_df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 135 entries, 0 to 134
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   date        131 non-null   datetime64[ns]
1   price       134 non-null   float64
2   location    134 non-null   float64
3   clicks      121 non-null   object
dtypes: datetime64[ns](1), float64(2), object(1)
memory usage: 4.3+ KB

co2_df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 226 entries, 0 to 225
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype
```

```

---  -----  -----
0   year      226 non-null    datetime64[ns]
1   co2       201 non-null    float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 3.7 KB

```

The summary for `co2_df` confirms that there are 25 missing values in the `co2` column (226 total records minus the 201 non-null values).

However, the summary for `ecom_df` shows that the `clicks` column is of type object, indicating mixed types instead of the expected `float64`. Let's investigate further using basic summary statistics.

10. To generate summary statistics for a DataFrame, use the `DataFrame.describe()` method:

```
co2_df.describe(include='all')
```

The output is as follows:

	year	co2
count	226	201.000000
mean	1906-11-27 01:29:12.212389376	1.590015
min	1750-01-01 00:00:00	0.000000
25%	1851-04-02 06:00:00	0.076400
50%	1907-07-02 12:00:00	0.935100
75%	1963-10-01 18:00:00	2.807600
max	2020-01-01 00:00:00	4.907900
std	NaN	1.644182

Figure 6.3: `co2_df` summary statistics indicating zero values present in the data

By using `include='all'`, instead of the default value `include=None`, the summary statistics include all columns in the DataFrame; both numeric and non-numeric. This means you will see statistics for every column, with the type of summary depending on the data type: numeric columns receive quantitative measures such as mean, standard deviation, and percentiles, while object or categorical columns get frequency-based summaries such as counts, unique values, and the most frequent value. If a particular statistic doesn't apply to a column, you will see `NaN` in that spot.

The summary statistics for the `co2_df` DataFrame confirms that we have zero values under the `co2` column (`min = 0.00`). As pointed out earlier, domain knowledge tells us that `0` represents missing data (null). These zero values will need to be replaced with `NaN` proper handling during the imputation process.

Next, check the summary statistics for `ecom_df`:

```
ecom_df.describe(include='all')
```

The output is as follows:

		date	price	location	clicks
count		131	134.000000	134.000000	121
unique		NaN	NaN	NaN	119
top		NaN	NaN	NaN	?
freq		NaN	NaN	NaN	2
mean	2008-06-08 22:32:03.664122112		43.480221	1.694030	NaN
min	2008-04-01 00:00:00		42.207018	1.000000	NaN
25%	2008-05-07 12:00:00		43.038050	1.000000	NaN
50%	2008-06-09 00:00:00		43.498842	1.000000	NaN
75%	2008-07-11 12:00:00		43.889935	2.000000	NaN
max	2008-08-13 00:00:00		45.801613	5.000000	NaN
std		NaN	0.610578	1.118724	NaN

Figure 6.4: `ecom_df` summary statistics indicating the ? value in the clicks column

The summary statistics for the `ecom_df` DataFrame indicate the presence of a ? value in the clicks column. This explains why pandas did not parse the column as numeric and interpreted it as having mixed types. These ? values will need to be replaced with NaN to be treated as missing values during the imputation process.

11. Replace placeholder values (0 in `co2_df` and ? in `ecom_df`) with NaN types using the `DataFrame.replace()` method:

```
co2_df.replace(0, np.NaN, inplace=True)
ecom_df.replace('?', np.NaN, inplace=True)
ecom_df['clicks'] = ecom_df['clicks'].astype('float')
```

Columns with mixed types (e.g., containing both strings and numbers) are treated as object dtype in pandas. This is why, after replacing ' ? ' with `np.NaN` in the clicks column, we needed to explicitly convert it to a float using `.astype('float')` to allow for proper numeric operations in subsequent analysis.

Validate the changes by running `DataFrame.isnull().sum()`. You should notice that the missing value counts have increased:

```
co2_df.isnull().sum()
>>
year      0
```

```
co2      35
dtype: int64

ecom_df.isnull().sum()
>>
date      4
price     1
location   1
clicks    16
dtype: int64
```

The updated counts reflect the additional missing values after replacing the placeholder values, providing a more accurate representation of the dataset's missing data.

How it works...

When reading the CSV files using `pandas.read_csv()`, the default behavior is to recognize and parse certain string values, such as NA, N/A, and null, as NaN (missing). Thus, once these values are converted to NaN, pandas can infer the `float64` (numeric) type for the `co2` column based on the remaining non-null values.

This behavior is enabled by two parameters: `na_values` and `keep_default_na`:

- The `na_values` parameter, by default, includes a predefined list of strings that pandas interprets as NaN. The list includes #N/A, #N/A N/A, #NA, -1.#IND, -1.#QNAN, -NaN, -nan, 1.#IND, 1.#QNAN, <NA>, N/A, NA, NULL, NaN, n/a, nan, and null.

You can append to this list by providing additional values to the `na_values` parameter. Additionally, `keep_default_na` is set to `True` by default, thus using (appending) `na_values` with the default list for parsing.

- The `keep_default_na` parameter is set to `True` by default, meaning the default `na_values` list is used for parsing missing values. If you set `keep_default_na=False` without providing new values for `na_values`, then pandas will not interpret the default string values (e.g., NA, N/A, and null) as missing. Instead, these values will remain as is, and the `co2` column will be parsed as an object (string) type.

In essence, pandas gives you flexible options to handle missing values during import: you can use its default missing value detections, customize which values should be treated as missing, or you can choose to keep all values as they appear in the source data (meaning nothing is treated as missing).

Here is an example:

```
co2_df = pd.read_csv(folder/co2_file,
                     keep_default_na=False)
co2_df.isna().sum()
>>
```

```

year      0
co2       0
dtype: int64

co2_df.shape
>> (226, 2)

```

In this case, no records were lost (226 total records), but pandas did not detect any NaN (missing) values. Let's inspect the structure of the DataFrame:

```

co2_df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 226 entries, 0 to 225
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0   year    226 non-null      int64
1   co2     226 non-null      object
dtypes: int64(1), object(1)
memory usage: 3.7+ KB

```

Notice that the co2 column is now of type object instead of float64. This is because pandas treated the placeholder strings as valid values rather than converting them to NaN. Now, let's inspect rows 190 to 194:

```

print(co2_df[190:195])
>>
      year    co2
190  1985     NA
191  1986    N/A
192  1987   null
193  1988  4.2953
194  1989  4.2782

```

Finally, let's inspect how the missing values are represented:

```

co2_df.iloc[132:139]

```

You will notice all seven rows contain blank values (empty string).

In this recipe, you explored the `.isna()` method. Once the data is read into a DataFrame or Series, you get access to both `.isna()` and `.isnull()` methods for detecting missing values. These methods are interchangeable and return True for missing values and False otherwise.

To get the counts of missing values for each column, you can chain the `sum()` function. To calculate the total number of missing values across all columns, you can chain another `sum()`:

```
co2_df.isnull().sum()
co2_df.isnull().sum().sum()
```

This approach provides a straightforward way to detect and quantify missing values in your dataset, which is an essential step in preparing data for analysis or modeling.

There's more...

If you know in advance that the dataset will always have specific placeholder values, such as `?`, which should be converted to `NaN` (or any other value), then you can utilize the `na_values` parameter in the `pd.read_csv()` function. This approach can reduce the number of steps needed to clean the data after creating the `DataFrame`.

For example, the following code automatically replaces all instances of `?` with `NaN` during the data ingestion process:

```
ecom_df = pd.read_csv(folder/ecom_file,
                      parse_dates=['date'],
                      na_values=['?'])
```

This ensures that all instances of `?` are interpreted as `NaN` without requiring additional processing steps, streamlining the data cleaning workflow.

See also

- To learn more about the `na_values` and `keep_default_na` parameters from `pandas.read_csv()`, visit the official documentation here: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
- To learn more about the `DataFrame.isna()` function, visit the official documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.isna.html>

Handling missing data with univariate imputation using pandas

Generally, there are two main approaches to imputing missing data: **univariate imputation** and **multivariate imputation**. In this recipe, you will explore univariate imputation techniques available in `pandas`.

In univariate imputation, only the non-missing values within a **single variable** (column or feature) are used to estimate and replace the missing values for that variable. For example, if a dataset contains a column for sales with missing values, you could apply **mean imputation**, a univariate imputation method, by calculating the average sales (from non-missing values) and using it to fill the missing sales observations. Here, the imputation relies solely on the values in the `Sales` column (a single column) without considering other variables in the dataset.

Some commonly used univariate imputation techniques include the following:

- **Mean imputation:** Replacing missing values with the mean of the non-missing values in the column.
- **Forward fill:** Imputing using the last non-missing observation to fill subsequent missing values (forward). This can be referred to as **Last Observation Carried Forward (LOCF)**. This technique is useful in time series where missing values are likely to resemble prior observations, and when values are expected to remain stable or change incrementally.
- **Backward fill:** Imputing using the next non-missing observation to fill preceding missing values (backward). This can be referred to as **Next Observation Carried Backward (NOCB)**. This technique is useful in time series where future observations are more likely to represent the missing values.

To better understand these techniques, you will use two datasets containing missing data and two corresponding original datasets without missing data. The datasets will allow you to evaluate the accuracy of the imputation methods both visually and numerically.

Getting ready

You can download the Jupyter notebooks and required datasets from the GitHub repository. Refer to the *Technical requirements* section of this chapter.

The following datasets are available in the Ch6 folder of the GitHub repository:

- `clicks_original.csv`: Contains the original clickstream data without missing values.
- `clicks_missing.csv`: A modified version of the clickstream data with missing values.
- `co2_original.csv`: Contains the original CO2 emissions data without missing values.
- `co2_missing_only.csv`: A modified version of the CO2 emissions data with missing values.

The original datasets will serve as the baseline for comparison, while the datasets with missing data will be used to test imputation techniques.

How to do it...

You will start by reading all four CSV files using the `read_dataset()` function created in the *Technical requirements* section of this chapter. The original versions of the datasets will serve as a baseline to compare the results of the imputations, helping you gain a better intuition on how different techniques perform. The *RMSE* will be used as a numerical measure to evaluate and compare the imputation accuracy of each technique, as well as the visualization of the outputs to complement the evaluation:

1. Use the `read_dataset()` function to read the four datasets:

```
folder = Path('../../datasets/Ch6/')

co2_original = read_dataset(
    folder, 'co2_original.csv', 'year', index=True)
co2_missing = read_dataset(
    folder, 'co2_missing_only.csv', 'year', index=True)
```

```
clicks_original = read_dataset(
    folder, 'clicks_original.csv', 'date', index=True)
clicks_missing = read_dataset(
    folder, 'clicks_missing.csv', 'date', index=True)
```

2. Visualize the original and missing versions of the CO2 datasets using the `plot_dfs` function created in the *Technical requirements* section. Provide the column name (`co2`) with missing values to the function:

```
plot_dfs(co2_original,
         co2_missing,
         'co2',
         title="Annual CO2 Emission per Capita",
         xlabel="Years",
         ylabel="x100 million tons")
```

The `plot_dfs` function will produce two plots: the original CO2 dataset without missing values, and the altered dataset with missing values.

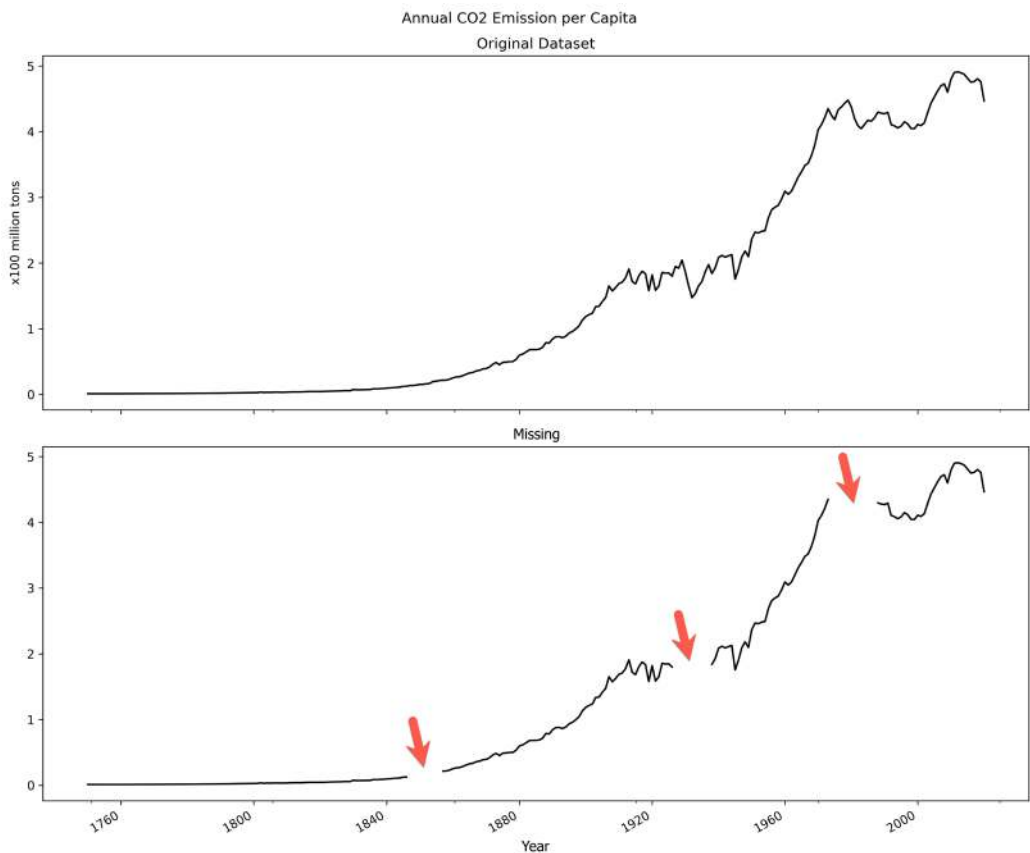


Figure 6.5: CO2 dataset showing a comparison between the missing values and the original

In *Figure 6.5*, you can observe a noticeable upward trend in CO2 levels over time. There is missing data in three different spots.

- Now, visualize the clickstream dataset and provide the column name (clicks) with missing values to the `plot_dfs` function:

```
plot_dfs(clicks_original,
         clicks_missing,
         'clicks',
         title="Page Clicks per Day",
         xlabel="date",
         ylabel="# of clicks")
```

The `plot_dfs` function will produce two plots: the original clickstream dataset without missing values, and the altered dataset with missing values.

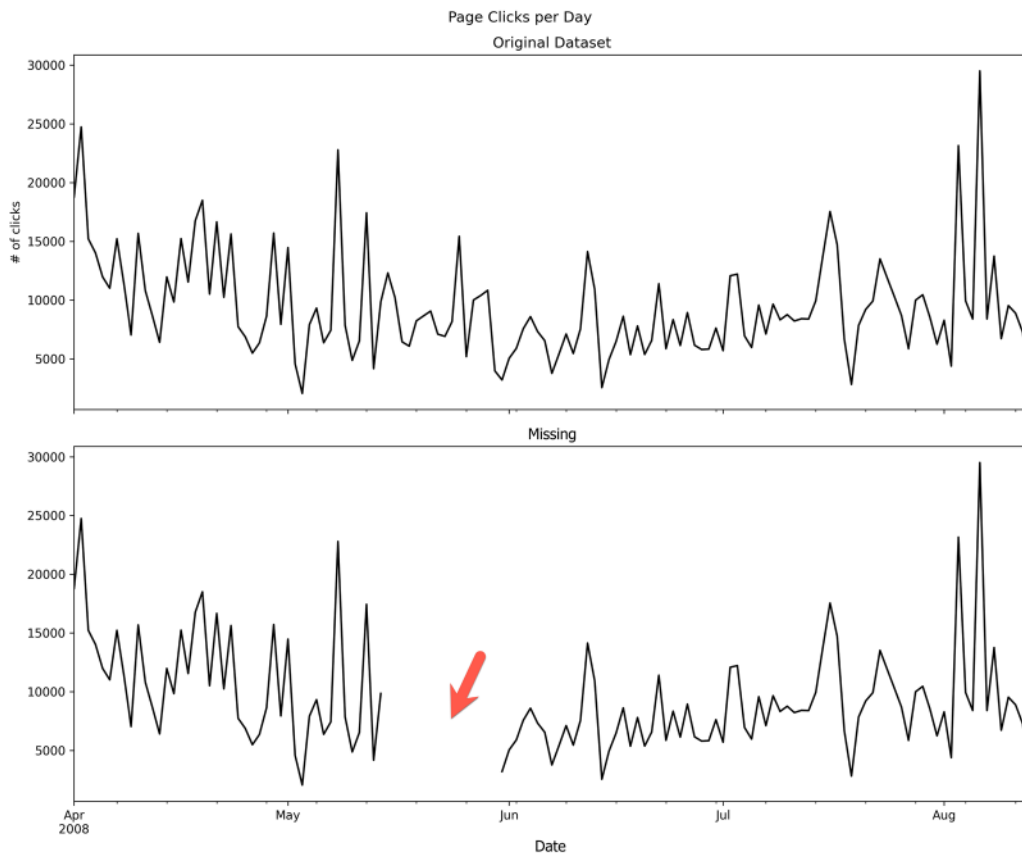


Figure 6.6: Clickstream dataset showing a comparison between the missing values and the original

In *Figure 6.6*, you can see missing data from May 15 to May 30. You can confirm this by running the following code:

```
clicks_missing[clicks_missing['clicks'].isna()]
```

4. Now you are ready to perform your first imputation. You will use the `.fillna()` method for mean imputation. The function takes a value parameter that can accept a numeric or a string value to substitute for all the NaN instances. Additionally, you will use `.ffill()` for forward fill, and `.bfill()` for backward fill.
5. Let's impute the missing values and append the results as new columns to the DataFrame. Start with the CO2 DataFrame:

```
co2_missing['ffill'] = co2_missing['co2'].ffill()
co2_missing['bfill'] = co2_missing['co2'].bfill()
co2_missing['mean'] = co2_missing['co2'].fillna(
    co2_missing['co2'].mean()
)
```

Use the `rmse_score` function created earlier, in the *Technical requirements* section, to get the scores:

```
_ = rmse_score(co2_original, co2_missing, 'co2')
>>
RMSE for ffill: 0.05873012599267133
RMSE for bfill: 0.05550012995280968
RMSE for mean: 0.7156383637041684
```

Both forward fill and backward fill methods maintain the trend of the dataset, as seen in the plots. However, backward fill outperformed forward fill in terms of RMSE for the CO2 dataset. This is likely because backward fill leverages future observations that align more closely with the missing values, preserving the overall trend and minimizing deviation.

Next, visualize the results using the `plot_dfs` function:

```
plot_dfs(co2_original, co2_missing, 'co2')
```

The preceding code produces the results that follow:

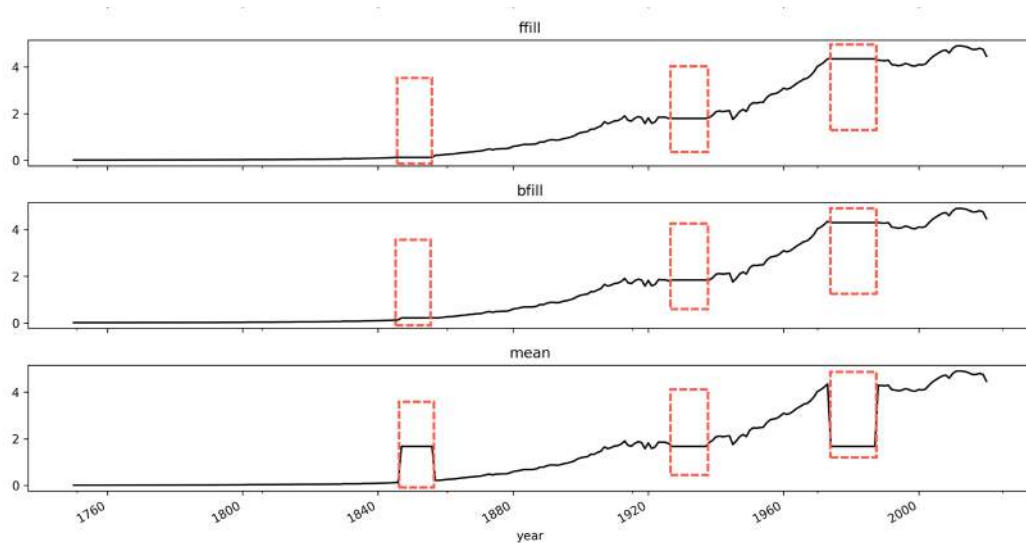


Figure 6.7: Comparison between the three imputation methods for the CO2 DataFrame

Compare the results in Figure 6.7 with the original data in Figure 6.5. Notice that both forward fill (ffill) and backward fill (bfill) perform better than mean imputation, as evidenced by the RMSE scores and visual representation.

- Now, perform the same imputation methods on the clickstream DataFrame:

```
clicks_missing['ffill'] = clicks_missing['clicks'].ffill()
clicks_missing['bfill'] = clicks_missing['clicks'].bfill()
clicks_missing['mean'] = clicks_missing['clicks'].fillna(
    clicks_missing['clicks'].mean()
)
```

Now, calculate the RMSE scores:

```
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
>>
RMSE for ffill: 1034.1210689204554
RMSE for bfill: 2116.6840489225033
RMSE for mean: 997.7600138929953
```

Interestingly, for the clickstream dataset, the mean imputation had the lowest RMSE score, in contrast to the results from the CO2 dataset. This may be because the clickstream data does not exhibit strong trends, making the mean a simple yet effective choice for filling gaps without introducing significant bias. Let's visualize the results to get another perspective on performance:

```
plot_dfs(clicks_original, clicks_missing, 'clicks')
```

You get the plots that follow:

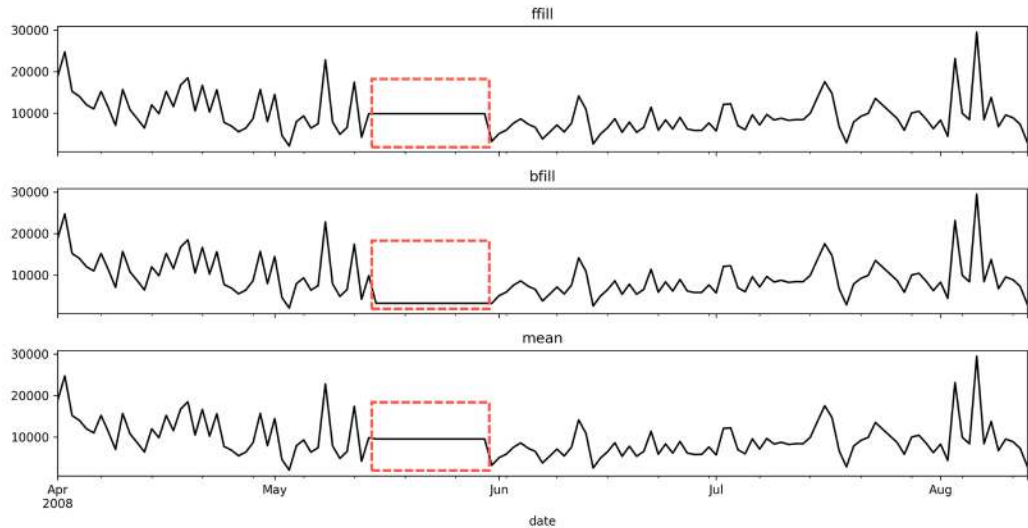


Figure 6.8: Comparison between the three imputation methods for the clickstream DataFrame

Compare the results in Figure 6.8 with the original data in Figure 6.6. From imputing two different datasets (CO2 and clickstream), you observe there is no *one-size-fits-all strategy* when it comes to handling missing data. Instead, each dataset requires a different strategy. Therefore, you should always inspect your results and align the outputs with the expectations based on the nature of your data.

How it works...

The effectiveness of different imputation methods, such as `.fillna()`, `.bfill()`, and `.ffill()`, is closely tied to the temporal structure and patterns in the dataset, such as trends, seasonality, or randomness, for example:

- **Backward fill (bfill)** leveraging future observations can be advantageous for datasets with strong trends or seasonality. However, keep in mind that using future values for backfilling (bfill) can introduce lookahead bias or data leakage in predictive modeling tasks.
- **Forward fill (ffill)** is suitable for datasets where continuity from past values is assumed.
- For datasets without temporal dependencies (e.g., clickstream data), **mean imputation** can be a reasonable choice as it avoids introducing trends or biases.

In the previous section, you explored how these methods work:

- **Using `DataFrame.fillna()` with a value:** This is the simplest imputation method. You replaced all missing values in a column with a scalar numeric value, such as the **mean** of the non-missing values.

- **Backward filling** with `.bfill()`: This method filled gaps by propagating the next valid observation (after the missing spot) backward.
- **Forward filling** with `.ffill()`: This method filled gaps by propagating the last valid observation (before the missing spot) forward.

Each technique is suited to specific use cases, and the choice of method should align with the dataset's characteristics and the goals of your analysis.

There's more...

The value parameter in `.fillna()` can accept not just a scalar but also a Python **dictionary**, a pandas **Series**, or even a pandas **DataFrame**. Let's explore these options with practical examples.

Using a Python dictionary

You can use a Python dictionary to impute missing values for multiple columns by specifying a mapping of column names to imputation values or strategies. Here is how it can be done.

Start by reading the `clicks_missing_more.csv` file:

```
clicks_missing = read_dataset(folder, 'clicks_missing_more.csv',
                               'date', index=True)

clicks_missing.isna().sum()
```

This should produce the following:

```
price      5
location   6
clicks     16
dtype: int64
```

Here, we have three columns with missing values (`price`, `location`, and `clicks`). We can use a dictionary to define a mapping in which each **key-value** pair corresponds to a column in the `clicks_missing` DataFrame. We can define different statistical measures (**median**, **mean**, and **mode**) for the imputation strategy for each column. This is illustrated in the following code:

```
values = {'clicks': clicks_missing['clicks'].median(),
          'price': clicks_missing['price'].mean(),
          'location': clicks_missing['location'].mode()[0]}
```

Note that when using `mode()`, we need to access the first element with `[0]` since pandas' `mode` function can return multiple values as a **Series**, not a single scalar value like `mean()` or `median()`.

Apply the dictionary using `.fillna()`:

```
clicks_missing.fillna(value=values, inplace=True)

clicks_missing.isna().sum()
```

The preceding code should produce the following results indicating all three columns have their missing values filled.

```
price      0
location   0
clicks     0
dtype: int64
```

The `inplace=True` argument modifies the `clicks_missing` DataFrame directly, ensuring the changes are applied without the need to reassign the DataFrame.

Using another DataFrame

You can also use another pandas DataFrame (or Series) to impute missing values, provided the columns match between the two DataFrames. This allows you to map values from one DataFrame to another.

In the following example, you will read the `clicks_missing_more.csv` file and the `clicks_original.csv` file for demonstration:

```
clicks_missing = read_dataset(folder, 'clicks_missing_more.csv',
                               'date', index=True)
clicks_original = read_dataset(folder, 'clicks_original.csv',
                               'date', index=True)
```

You will use the `clicks_original` DataFrame to impute missing values in the `clicks_missing` DataFrame:

```
clicks_missing.fillna(value=clicks_original, inplace=True)
clicks_missing.isna().sum()
```

The preceding code should produce the following results, indicating all three columns have their missing values filled:

```
price      0
location   0
clicks     0
dtype: int64
```

Here, the missing values in `clicks_missing` are replaced with the corresponding values from `clicks_original`.

See also

- To learn more about `DataFrame.fillna()`, visit the official documentation page here: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html>.

In the next recipe, you will perform similar univariate imputation, but this time using the `scikit-learn` library.

Handling missing data with univariate imputation using scikit-learn

scikit-learn is a widely popular machine learning library in Python, known for its extensive range of algorithms and tools for classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. Among its preprocessing capabilities, scikit-learn provides support for handling missing data, including both **univariate** and **multivariate** imputation techniques.

In this recipe, you will learn how to use scikit-learn's `SimpleImputer` for univariate imputation. `SimpleImputer` allows you to replace missing values in a dataset with statistical measures such as the **mean**, **median**, or **most frequent** value. This approach is useful when dealing with datasets where missing values need to be estimated based on non-missing values in the same variable (column or feature).

Getting ready

You can download the Jupyter notebooks and required datasets from the GitHub repository. This recipe will utilize the three functions prepared in the *Technical requirements* section of this chapter (`read_dataset`, `rmse_score`, and `plot_dfs`).

The following datasets are available in the Ch6 folder of the GitHub repository:

- `clicks_original.csv`: Contains the original clickstream data without missing values
- `clicks_missing.csv`: A modified version of the clickstream data with missing values
- `co2_original.csv`: Contains the original CO2 emissions data without missing values
- `co2_missing_only.csv`: A modified version of the CO2 emissions data with missing values

The original datasets will serve as the baseline for comparison, while the datasets with missing data will be used to test imputation techniques.

How to do it...

You will start by importing the libraries and then read all four CSV files:

1. Use the `SimpleImputer` class from scikit-learn to perform univariate imputation. Use the `read_dataset()` function to read the four datasets:

```
from sklearn.impute import SimpleImputer

folder = Path('.././datasets/Ch6/')
co2_original = read_dataset(
    folder, 'co2_original.csv', 'year', index=True)
co2_missing = read_dataset(
    folder, 'co2_missing_only.csv', 'year', index=True)
clicks_original = read_dataset(
    folder, 'clicks_original.csv', 'date', index=True)
clicks_missing = read_dataset(
    folder, 'clicks_missing.csv', 'date', index=True)
```

2. `SimpleImputer` accepts different values for the `strategy` parameter, such as `mean`, `median`, and `most_frequent`. Let's explore all three strategies and see how they compare.

Create a list of tuples for each method:

```
strategy = [
    ('Mean Strategy', 'mean'),
    ('Median Strategy', 'median'),
    ('Most Frequent Strategy', 'most_frequent')
]
```

You can loop through the `strategy` list to apply the different imputation strategies. The `SimpleImputer` class provides a `fit_transform` method, which combines two steps: fitting the imputer to the data (`.fit`), and then transforming the data (`.transform`).

Since **scikit-learn** estimators generally expect features as columns in a 2D array (even for a single feature), `SimpleImputer` requires a 2D NumPy array as input. You will need to use the `Series.values` property followed by the `.reshape(-1, 1)` method to create a 2D NumPy array. This transforming reshapes the 1D array from shape (226,) to a 2D array of shape (226, 1), which is a column vector:

```
co2_vals = co2_missing['co2'].values.reshape(-1,1)
clicks_vals = clicks_missing['clicks'].values.reshape(-1,1)
```

Next, loop through the list of strategies:

```
for s_name, s in strategy:
    co2_missing[s_name] = (
        SimpleImputer(missing_values=pd.NA,
                       strategy=s).fit_transform(co2_vals))
    clicks_missing[s_name] = (
        SimpleImputer(missing_values=pd.NA,
                       strategy=s).fit_transform(clicks_vals))
```

After this step, both the `clicks_missing` and `co2_missing` DataFrames will have three additional columns – one for each of the imputation strategies applied.

Notice the `missing_values` parameter takes a **placeholder** for the missing values. For null values, you can pass either `pd.NA` or `np.nan`.

3. Evaluate each strategy using the `rmse_score` function. Start with the CO2 dataset. You should get an output similar to the following:

```
_ = rmse_score(co2_original, co2_missing, 'co2')
>>
RMSE for Mean Strategy: 0.7156383637041684
RMSE for Median Strategy: 0.8029421606859859
RMSE for Most Frequent Strategy: 1.1245663822743381
```

For the clickstream data, you should get an output similar to the following:

```
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
>>
RMSE for Mean Strategy: 997.7600138929953
RMSE for Median Strategy: 959.3580492530756
RMSE for Most Frequent Strategy: 1097.6425985146868
```

Notice how the RMSE score differs between the two datasets. For example, the mean strategy performed best on the CO2 data, while the median strategy produced the best result for the clickstream data.

4. Visualize the results using `plot_dfs`. Start with the CO2 dataset:

```
plot_dfs(co2_original, co2_missing, 'co2')
```

This will produce the following plots:

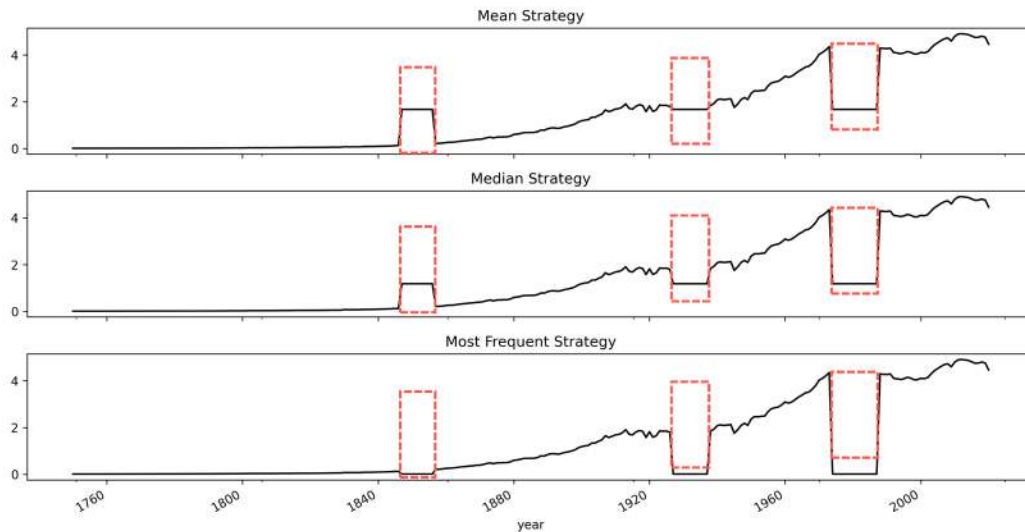


Figure 6.9: Comparing three `SimpleImputer` strategies for the CO2 dataset

Compare these results with the original data in Figure 6.5.

5. For the clickstream dataset, you should use the following:

```
plot_dfs(clicks_original, clicks_missing, 'clicks')
```

This should plot all three strategies:

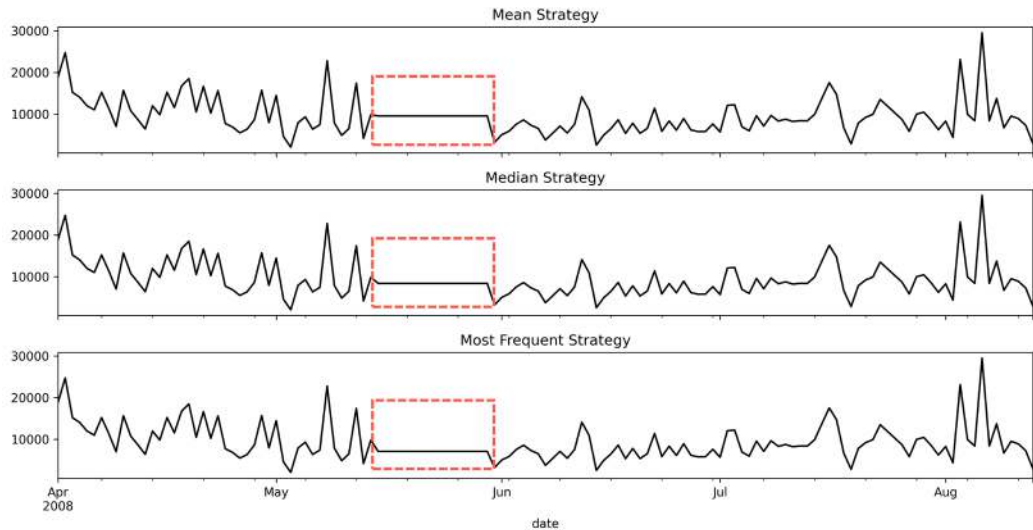


Figure 6.10: Comparing three SimpleImputer strategies for the clickstream dataset

Compare this with the original data in Figure 6.6.

SimpleImputer provides basic strategies that may be suitable for some data but not optimal for other data. The advantage of these simple imputation strategies (including the ones from the previous *Handling missing data with univariate imputation using pandas recipe*) is that they are fast and straightforward to implement.

How it works...

You used the SimpleImputer class from scikit-learn to implement three basic strategies for imputing missing values: mean, median, and most_frequent (mode). These are univariate imputation techniques, meaning that only one feature or column was used to compute the imputed value (mean, median, or mode).

The SimpleImputer class has three **parameters** that you should understand:

- **missing_values:** By default, this is set to `np.nan`, but it can also accept `pd.NA` or other placeholder values. NumPy's `np.nan` and pandas' `NaN` are both used to represent missing values in Python. They are functionally equivalent in pandas, as shown in the following example:

```
test = pd.Series([np.nan, np.nan, np.nan])
test
>>
0    NaN
1    NaN
2    NaN
dtype: float64
```

The output shows `dtype: float64` because missing values (NaN) inherently make the Series or DataFrame `dtype float64` in pandas, even if the other values are integers.

`SimpleImputer` will replace all occurrences of `missing_values`. You can update this parameter to handle other missing value representations, such as `pandas.NA`, integers, floats, or even strings.

- **Strategy:** This determines the imputation method. The default value is the string "mean", but it can also be "median", "most_frequent", or "constant".
- **fill_value:** This parameter is used to specify the replacement value when the strategy is set to "constant". The value can be either a string or a numeric value. This is similar to `fill.na()` in pandas.

When using scikit-learn for preprocessing tasks like imputation, the typical workflow involves two steps:

1. **Fitting the imputer:** First, you **fit** the imputer to the data using the `.fit()` method. This step involves “**training**” the imputer, which in the context of imputation means calculating the necessary statistics (such as mean, median, etc.) from the provided data. The fitting process is usually done on the training dataset.
2. **Applying the transform:** After fitting, you apply the imputer to the data using the `.transform()` method. This step performs the actual imputation by replacing missing values with the computed statistics.

In our example, these two steps were combined into one using the `.fit_transform()` method. This convenient approach computes the necessary statistics and performs the imputation in a single step, which is especially useful during the initial stages of data preprocessing.

The pandas `.fillna()` method provides similar functionality to scikit-learn's `SimpleImputer`. For instance, the **mean** strategy can be achieved using the pandas DataFrame `.mean()` method combined with `.fillna()`.

The following example illustrates this and compares the two outcomes from scikit-learn and pandas:

```
avg = co2_missing['co2'].mean()

co2_missing['pandas_fillna'] = co2_missing['co2'].fillna(avg)

cols = ['co2', 'Mean Strategy', 'pandas_fillna']
_ = rmse_score(co2_original, co2_missing[cols], 'co2')
>>
RMSE for Mean Strategy: 0.7156383637041684
RMSE for pandas_fillna: 0.7156383637041684
```

The output shows that both methods produce identical results.

The pandas `.fillna()` method has an added advantage: it can scale imputation across the entire `DataFrame` with ease (column by column). For example, if you have a `sales_report_data` `DataFrame` containing multiple columns with missing data, you can perform mean imputation for all columns with a single line of code:

```
sales_report_data.fillna(sales_report_data.mean(), inplace=True)
```

This ability to handle multiple columns simultaneously makes `.fillna()` particularly useful when working with larger datasets. Here is an example:

```
import pandas as pd
import numpy as np

data = {
    'A': [1, 2, np.nan, 4],
    'B': [np.nan, 2, 3, 4],
    'C': [1, np.nan, np.nan, 4]
}
sales_report_data = pd.DataFrame(data)
print(sales_report_data)

>>
   A    B    C
0  1.0  NaN  1.0
1  2.0  2.0  NaN
2  NaN  3.0  NaN
3  4.0  4.0  4.0
```

Next, apply the mean imputation across all columns:

```
sales_report_data.fillna(sales_report_data.mean(), inplace=True)
print(sales_report_data)

>>
   A    B    C
0  1.000000  3.0  1.0
1  2.000000  2.0  2.5
2  2.333333  3.0  2.5
3  4.000000  4.0  4.0
```

This demonstrates that missing values in each column are replaced by their respective column means. Note that the `.mean()` method only applies to numeric columns. Non-numeric columns will remain unchanged during this operation.

There's more...

The `add_indicator` option in scikit-learn's `SimpleImputer` enhances the imputation process by adding a `MissingIndicator` transform to the output. This adds an additional binary column that indicates whether the original data was missing (1) or observed (0). Encoding missing information as a feature (column) can provide additional insights, especially in cases where the pattern of missingness itself correlates with the target variable.

The following example demonstrates how to enable this feature with `add_indicator=True`. You will use `.fit()` followed by `.transform()`:

```
co2_missing = read_dataset(folder, 'co2_missing_only.csv',
                             'year', index=True)
co2_vals = co2_missing['co2'].values.reshape(-1,1)
imputer = SimpleImputer(strategy='mean', add_indicator=True)
imputer.fit(co2_vals)
imputer.get_params()
```

The `get_params()` method outputs the following:

```
>>
{'add_indicator': True,
 'copy': True,
 'fill_value': None,
 'keep_empty_features': False,
 'missing_values': nan,
 'strategy': 'mean'}
```

You can then use `.transform()` to perform the imputation and include the missing indicator as an additional column. The final output will include two additional columns to the original `co2_missing` `DataFrame`:

```
co2_missing[['imputed', 'indicator']] = (imputer.transform(co2_vals))
print(co2_missing.loc['1855-01-01':].head())
```

The preceding code should produce the following output:

	co2	imputed	indicator
year			
1855-01-01	NaN	1.673262	1.0
1856-01-01	NaN	1.673262	1.0
1857-01-01	0.2163	0.216300	0.0
1858-01-01	0.2192	0.219200	0.0
1859-01-01	0.2320	0.232000	0.0

Notice how the indicator column aligns with the original dataset, where zero (0) represents observed values, and one (1) indicates missing values. This feature is particularly useful when the presence or absence of data itself carries meaningful information. For instance, it can be treated as a feature in machine learning models to improve predictive performance.

See also

- To learn more about scikit-learn's `SimpleImputer` class, visit the official documentation page here: <https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html#sklearn.impute.SimpleImputer>.

So far, you have been dealing with univariate imputation. A more powerful approach is **multivariate imputation**, which you will learn about in the next recipe.

Handling missing data with multivariate imputation

Previously, we discussed two approaches to imputing missing data: **univariate imputation** and **multivariate imputation**.

As demonstrated in the earlier recipes, **univariate imputation** involves using a single variable (column) to substitute for missing data, without accounting for relationships with other variables in the dataset. While univariate imputation techniques are usually faster and easier to implement, they often oversimplify the problem and may not yield the best results in complex datasets.

In contrast, a **multivariate imputation** considers multiple variables within the dataset to estimate missing values. The idea is straightforward: by leveraging the relationship and correlation between variables, multivariate imputation can lead to more accurate and meaningful estimates, especially in datasets where variables are strongly interconnected. In other words, including more variables in the imputation process improves the predictability of missing values.

In simple terms:

- **Univariate imputation** handles missing values for a single variable independently (in isolation), without considering relationships with other variables in the dataset
- **Multivariate imputation** assumes that there is some synergy among variables in the dataset, and collectively, they can provide better estimates to fill in the missing values

In this recipe, you will be working with the clickstream dataset, which contains multiple variables (**clicks**, **price**, and **location** columns) to perform **multivariate imputation** for missing values in the **clicks** column. You will be introduced to two imputation methods in scikit-learn that leverage relationships between variables to impute missing data:

- `IterativeImputer` uses regression models iteratively to predict and impute missing values
- `KNNImputer` applies proximity-based methods to estimate missing values by calculating the mean or median of the nearest neighbors

`IterativeImputer` is ideal for datasets with complex relationships between variables, while `KNNImputer` works well for datasets with localized patterns or clustering in data.

Getting ready

You can download the Jupyter notebooks and required datasets from the GitHub repository.

This recipe will utilize the three functions prepared in the *Technical requirements* section of this chapter (`read_dataset`, `rmse_score`, and `plot_dfs`).

How to do it...

In this recipe, you will use **scikit-learn** for the multivariate imputation. The library provides two powerful tools: `IterativeImputer` and `KNNImputer`.

Using IterativeImputer

The `IterativeImputer` class allows you to pass a **regressor** to its **estimator** parameter, enabling you to predict the missing values based on other variables (columns) in the dataset:

1. Start by importing the necessary libraries, methods, and classes:

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import ExtraTreesRegressor, BaggingRegressor
from sklearn.linear_model import (ElasticNet,
                                  LinearRegression,
                                  BayesianRidge)
from sklearn.neighbors import KNeighborsRegressor
```

2. Load the two clickstream datasets into DataFrames:

```
folder = Path('../datasets/Ch6/')
clicks_original = read_dataset(folder, 'clicks_original.csv', 'date')
clicks_missing = read_dataset(folder, 'clicks_missing.csv', 'date')
```

3. With `IterativeImputer`, you can test different **regressors** (estimators). Let's create a list of regressors to evaluate with `IterativeImputer`:

```
regressors = [
    ('bayesianRidge', BayesianRidge()),
    ('extra_trees', ExtraTreesRegressor(n_estimators=50,
                                         random_state=42)),
    ('bagging', BaggingRegressor(n_estimators=50, random_state=42)),
    ('elastic_net', ElasticNet()),
    ('linear_regression', LinearRegression()),
    ('knn', KNeighborsRegressor(n_neighbors=3))
]
```

Setting `random_state=42` for the ensemble models (`ExtraTreeRegressor` and `BaggingRegressor`) ensures reproducible results by fixing the random number generator seed. This is important for **tree-based** and **bagging** models, which use randomization in their training process. Without a fixed random state, you would get slightly different imputation results each time you run the code.

4. Loop through the list of regressors to train the imputation models and apply them to the dataset. Use the `fit()` method to train the model and the `transform()` method to impute missing values for the `clicks` column. You will do this for each regressor in order to compare performance. You will append the results from each regressor as a new column to the `clicks_missing` DataFrame. This will make it easier when comparing the RMSE score and plotting for visual comparison:

```
# Select the relevant columns ('price', 'location', 'clicks')
# These are the first three columns and 'clicks' is the third (index 2)
clicks_vals = clicks_missing.iloc[:,0:3].to_numpy()

for reg_name, reg in regressors:
    est = IterativeImputer(
        random_state=15,
        estimator=reg).fit(clicks_vals)
    clicks_missing[reg_name] = est.transform(clicks_vals)[: , 2]
```

5. Use the `rmse_score` function to calculate the RMSE for each regressor:

```
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
```

This should print the following scores:

```
RMSE for bayesianRidge: 949.4393973455852
RMSE for extra_trees: 1404.1501333341182
RMSE for bagging: 1265.5100992417892
RMSE for elastic_net: 945.40752093431
RMSE for linear_regression: 938.941983142718
RMSE for knn: 1336.8798392251822
```

Bayesian Ridge, **ElasticNet**, and **Linear Regression** produce similar results with lower RMSE scores compared to others.

6. Plot the imputed results for a visual comparison between the different regressors:

```
plot_dfs(clicks_original, clicks_missing, 'clicks')
```

The output is as follows:

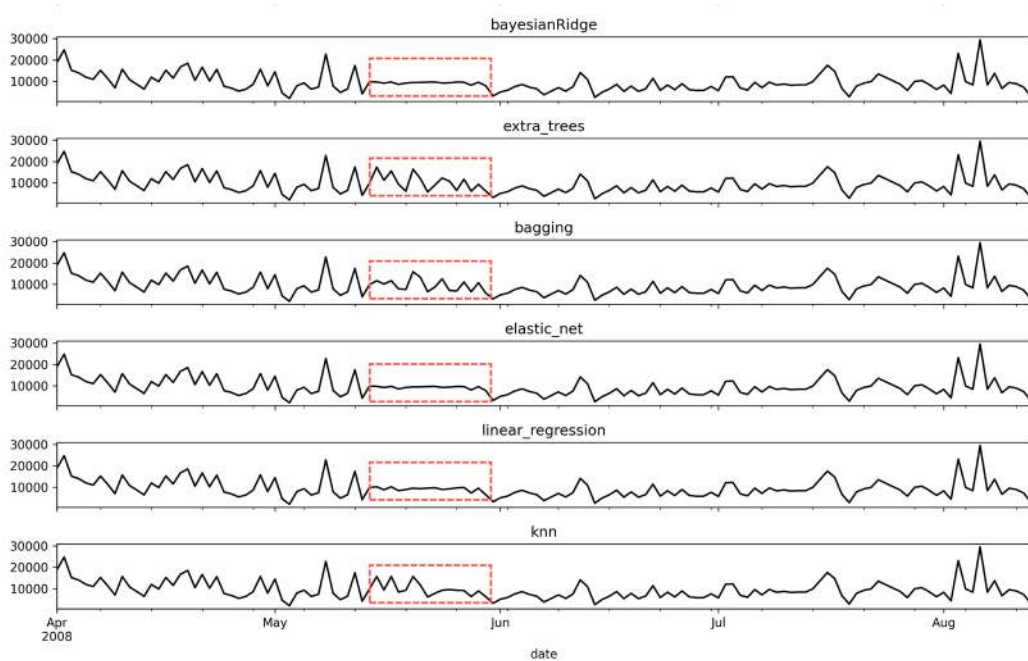


Figure 6.11: Comparing different estimators using *IterativeImputer*

Compare the results in Figure 6.11 with the original data in Figure 6.6.

While RMSE scores are useful for comparing methods, they don't always tell the full story about the quality of imputation. For more details, see the *How it works...* section.

Using KNNImputer

For an alternative approach to multivariate imputation, use the `KNNImputer` class provided by `scikit-learn`. `KNNImputer` estimates missing values by averaging the values of the nearest neighbors in the feature space.

1. Start by importing the `KNNImputer` class:

```
from sklearn.impute import KNNImputer
```

2. Fit and transform the dataset using `KNNImputer`, and append the results to a new column (`knn_imputer`) in the `clicks_missing` `DataFrame`. You can use `fit_transform` to do this in one step:

```
knn_imputer = KNNImputer(n_neighbors=3)
clicks_missing['knn_imputer'] = knn_imputer.fit_transform(
    clicks_vals
)[: , 2]
```

3. Calculate the RMSE using `rmse_score`:

```
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
>>
RMSE for bayesianRidge: 949.4393973455852
RMSE for extra_trees: 1404.1501333341182
RMSE for bagging: 1265.5100992417892
RMSE for elastic_net: 945.40752093431
RMSE for linear_regression: 938.941983142718
RMSE for knn: 1336.8798392251822
RMSE for knn_imputer: 1336.8798392251822
```

Notice that `KNNImputer` produced the same RMSE score as `KNeighborsRegressor` when used with `Iterative Imputer`.

4. Create a plot to compare the results from `KNNImputer` and `IterativeImputer` with `KNeighborsRegressor` visually:

```
plot_dfs(clicks_original,
         clicks_missing[['clicks', 'knn', 'knn_imputer']], 'clicks')
```

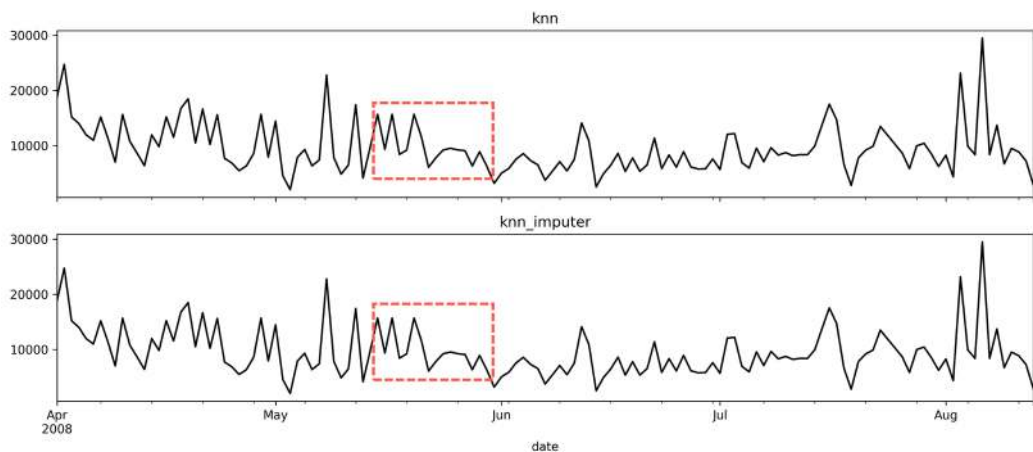


Figure 6.12: Comparing `KNNImputer` (`knn_imputer`) with `IterativeImputer` with `KNeighborsRegressor` (`knn`)

Even though the results from using `KNeighborsRegressor` in `IterativeImputer` produced similar results to `KNNImputer`, they work differently. `KNNImputer` is a non-iterative approach and uses local information from neighboring samples to impute missing values. Using `IterativeImputer` with `KNeighborsRegressor` involves building predictive models for each feature iteratively.

Generally, `KNNImputer` is ideal for datasets with localized relationships, where neighboring samples share similar values. On the other hand, `IterativeImputer` is better suited for datasets where missing values depend on more complex relationships among multiple variables.

How it works...

The `IterativeImputer` class from the `scikit-learn` library is inspired by the R `MICE` package (**M**ulti-**v**ariate **I**mputation by **C**hained **E**quations), a well-known implementation for imputing missing data (<https://www.jstatsoft.org/article/view/v045i03>). `MICE` uses a chained equations approach, where each variable with missing values is modeled iteratively as a function of other variables. `scikit-learn`'s implementation, `IterativeImputer`, typically returns a single imputation, whereas `MICE` in R is designed to produce multiple imputations. You can read more about the differences here: <https://scikit-learn.org/stable/modules/impute.html#id2>.



It is worth noting that `scikit-learn`'s `IterativeImputer` is still in experimental mode. This means its behavior or default parameters may change in future releases of `scikit-learn`. To use it, you have to explicitly enable it by importing the experimental module using the following:

```
from sklearn.experimental import enable_iterative_imputer
```

In the next section, you will use another implementation of `MICE` from the `statsmodels` library.

One of the key benefits of `IterativeImputer` is its flexibility in allowing the use of different regressors, such as the following:

- **BayesianRidge:** A regression model that uses Bayesian inference to estimate the distribution of coefficients, which helps handle multicollinearity and provides uncertainty estimates
- **ExtraTreesRegressor:** An ensemble-based regression model that uses randomized decision trees to improve prediction accuracy and reduce overfitting
- **BaggingRegressor:** A bootstrap aggregation model that combines predictions from multiple base regressors (e.g., decision trees) to improve stability and accuracy
- **ElasticNet:** A linear regression model that combines L1 (Lasso) and L2 (Ridge) regularization to handle correlated features and prevent overfitting
- **LinearRegression:** A simple linear regression model that predicts a target variable by fitting a linear relationship between the features and the target
- **KNeighborsRegressor:** A regression model that predicts the target value based on the average (or weighted average) of the *k*-nearest neighbors in feature space

At the beginning of the chapter, we discussed that using RMSE for evaluating imputation methods can be somewhat misleading. This is because, unlike predictive modeling, our objective of imputation is not necessarily to achieve the “best” score (i.e., the lowest RMSE value); instead, it is to fill missing data in a way that closely resembles the true nature and variability of the original dataset.

While RMSE does have limitations, it can still provide valuable insights when comparing different imputation methods. It helps us understand which method estimates the missing values more closely to their actual values, based on the available data.

However, it's crucial to recognize that a lower RMSE doesn't always mean a more "accurate" imputation in the context of real-world data. This is because real datasets often contain noise and randomness, which some imputation methods might fail to capture, especially those that produce the lowest RMSE scores. Methods such as `BayesianRidge`, `ElasticNet`, and `Linear Regression` might yield lower RMSE values but could *over-smooth* the data (see *Figure 6.11* for those three estimators), failing to reflect the inherent randomness and variability present in real datasets.

When using the imputed data for downstream tasks such as predictive modeling, we need to acknowledge that some level of imperfection in the imputed values is acceptable. This is because we often don't know the true nature of the missing data, and our aim is to create a dataset that provides a "good enough" representation for model training and analysis. In essence, the objective is to achieve a balance – an imputation that provides a reasonable estimate of missing values while preserving the overall characteristics of the data, including its randomness and variability.

There's more...

The `statsmodels` library offers an implementation of MICE that is closer to the R implementation. Unlike `IterativeImputer`, `statsmodels`' MICE allows for multiple imputations using chained equations.

You will append the results from the `statsmodels` MICE imputation to the `clicks_missing` `DataFrame` as an additional column, allowing you to compare the results with those from `IterativeImputer`. Start by importing the required libraries:

```
from statsmodels.imputation.mice import MICEData
import statsmodels.api as sm
```

Next, use the `MICEData` class to wrap the `clicks_missing` `DataFrame`. Specify the columns to include for MICE imputation and create an instance of `MICEData`.

```
cols = ['price', 'location', 'clicks']

# Create a MICEData object with specified columns from the dataset
mice_data = MICEData(clicks_missing[cols],
                      k_pmm=20,
                      perturbation_method='gaussian')
```

The values used for `k_pmm` and `perturbation_method` are the default values in `statsmodels`:

- `k_pmm`: Specifies the number of nearest neighbors used during **Predictive Mean Matching (PMM)**, which helps in maintaining realistic imputations
- `perturbation_method`: 'gaussian' adds noise from a Gaussian distribution to improve the robustness of imputations

Set up the imputation model for the `clicks` column:

```
mice_data.set_imputer('clicks', formula='price + location + clicks',
                      model_class=sm.OLS)
```

The imputation model uses **Ordinary Least Squares (OLS)** regression to predict missing values for the specified variable ('clicks') based on other variables ('price', 'location', and 'clicks' itself).

Now, perform 20 iterations of imputation:

```
mice_data.update_all(n_iter=20)
```

The `update_all()` method refines these predictions over multiple iterations (20 in this case), improving the accuracy of imputations.

Append the imputed values to the `clicks_missing` DataFrame and calculate the RMSE:

```
clicks_missing['MICE'] = mice_data.data['clicks'].values.tolist()
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
>>
RMSE for bayesianRidge: 949.4393973455851
RMSE for extra_trees: 1427.0064509984215
RMSE for bagging: 1284.2159560597736
RMSE for elastic_net: 945.40752093431
RMSE for linear_regression: 938.9419831427184
RMSE for knn: 1336.8798392251822
RMSE for knn_imputer: 1336.8798392251822
RMSE for MICE: 989.7082510630202
```

Finally, visualize the results for a final comparison. This will include some of the imputations from `IterativeImputer`:

```
cols = ['clicks', 'bayesianRidge', 'bagging', 'knn', 'MICE']
plot_dfs(clicks_original, clicks_missing[cols], 'clicks')
```

The output should look as follows:

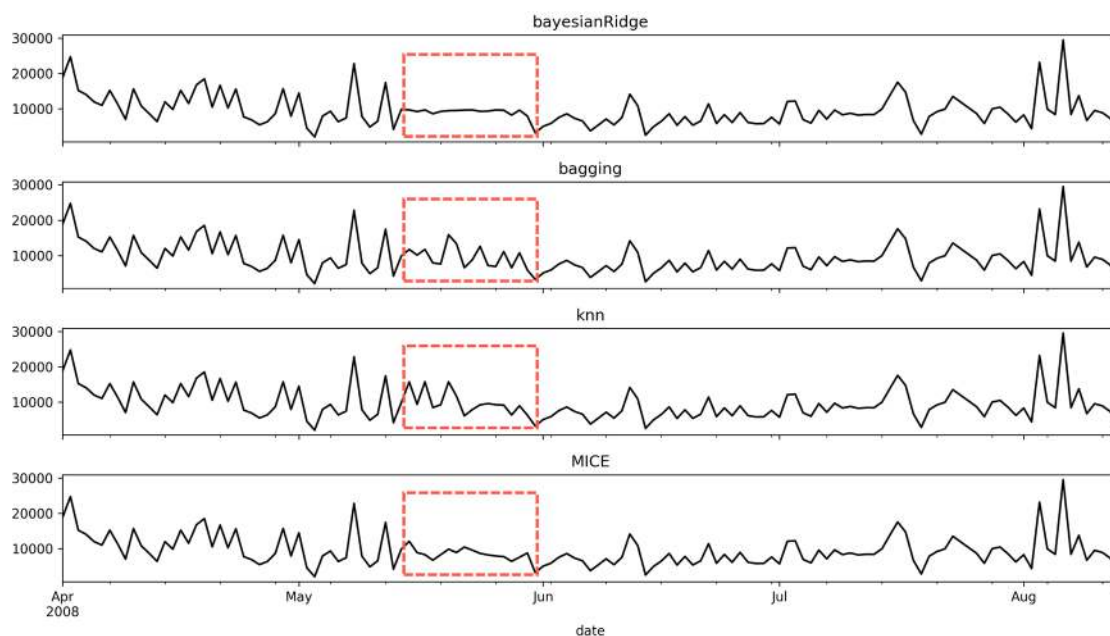


Figure 6.13: Comparing the statsmodels MICE implementation with scikit-learn's IterativeImputer

Compare the results with the original data in Figure 6.6.

Overall, multivariate imputation techniques generally produce better results than univariate methods, especially for datasets with complex relationships among variables. While univariate imputers (e.g., SimpleImputer) are faster and easier to use, multivariate approaches like statsmodels' MICE and Scikit-Learn's IterativeImputer are better suited for datasets where variables are highly correlated.

There is always a trade-off between simplicity, speed, and the quality of imputations. The choice of method depends on the dataset, the level of accuracy required, and the downstream use case.

See also

- To learn more about IterativeImputer, please visit the official documentation page here: <https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html#sklearn.impute.IterativeImputer>
- To learn more about statsmodels MICE implementation, please visit the official documentation page here: <https://www.statsmodels.org/dev/imputation.html>

Handling missing data with interpolation

Interpolation is another widely used technique for imputing missing values, particularly in time series datasets. The pandas library provides the `DataFrame.interpolate()` method, offering flexible and powerful options for **univariate imputation**.

For instance, **linear interpolation** imputes missing data by drawing a straight line between the two points surrounding the missing value. In a time series context, this means the missing value is estimated as a linear line based on the prior (past) value and the next (future) value. On the other hand, **polynomial interpolation** uses a curved line to estimate missing values between two points, enabling it to capture a non-linear relationship between points.

Examples of some of the interpolation methods available in pandas include the following:

- **Linear:** Straight-line interpolation between points
- **Nearest:** Uses the nearest available value for imputation
- **Polynomial/Quadratic:** Fits a polynomial curve (e.g., quadratic for order 2) through the points
- **Spline:** Fits a smooth curve through the points using spline functions

While **spline** and **polynomial** interpolation both use polynomials, spline fits **piecewise polynomials** to ensure smoothness at **knots** (points of transition between polynomial pieces), whereas polynomial fits a single polynomial across all points. As a result, spline interpolation can adapt better to non-linear patterns in localized regions, while polynomial interpolation may overfit or exaggerate trends, especially with higher orders.

Each method applies a unique mathematical approach to impute missing values. For example, linear interpolation is ideal for datasets with consistent trends, while spline interpolation is better suited for data with smooth, non-linear patterns.

Moreover, the interpolation capabilities of pandas can be extended further through the SciPy library, which provides additional options for **univariate** and **multivariate** interpolations.

In this recipe, you will use the pandas `DataFrame.interpolate()` function to explore various interpolation methods, including linear, polynomial, quadratic, nearest, and spline, to handle missing data.

Getting ready

You can download the Jupyter notebooks and required datasets from the GitHub repository.

This recipe will utilize the three functions prepared in the *Technical requirements* section of this chapter (`read_dataset`, `rmse_score`, and `plot_dfs`).

You will be using four datasets from the Ch6 folder: `clicks_original.csv`, `clicks_missing.csv`, `co2_original.csv`, and `co2_missing_only.csv`.

How to do it...

You will perform multiple interpolations on two different datasets and then compare the results using RMSE and visualization:

1. Start by reading the CSV files into DataFrames:

```
folder = Path('.././datasets/Ch6/')
co2_original = read_dataset(
    folder, 'co2_original.csv', 'year', index=True)
```

```
co2_missing = read_dataset(
    folder, 'co2_missing_only.csv', 'year', index=True)
clicks_original = read_dataset(
    folder, 'clicks_original.csv', 'date', index=True)
clicks_missing = read_dataset(
    folder, 'clicks_missing.csv', 'date', index=True)
```

2. The method parameter in `DataFrame.interpolate()` specifies the interpolation technique to apply. For a list of available interpolation methods, you can reference the documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html>

Create a list of the interpolation methods to be tested: linear, quadratic, nearest, and cubic:

```
interpolations = [
    'linear',
    'quadratic',
    'nearest',
    'cubic'
]
```

3. Loop through the list to run different interpolations using `.interpolate()`, and append a new column for each output to enable comparison:

```
for intp in interpolations:
    co2_missing[intp] = co2_missing['co2'].interpolate(method=intp)
    clicks_missing[intp] = clicks_missing[
        'clicks'
    ].interpolate(method=intp)
```

4. There are two additional interpolation methods we would like to add: `spline` and `polynomial`. To use these methods, you need to provide an integer value for the order parameter. You can try different order values for `spline` and `polynomial`. Start with `order=2` for the `spline` and `polynomial` methods as a balanced choice to model smooth, non-linear patterns without **overfitting**:

```
co2_missing['spline'] = \
    co2_missing['co2'].interpolate(method='spline', order=2)
co2_missing['polynomial'] = \
    co2_missing['co2'].interpolate(method='polynomial', order=2)

clicks_missing['spline'] = \
    clicks_missing['clicks'].interpolate(method='spline', order=2)
clicks_missing['polynomial'] = \
    clicks_missing['clicks'].interpolate(method='polynomial', order=2)
```

Note that the quadratic and cubic interpolation methods are essentially convenient shortcuts for polynomial interpolation with `order=2` and `order=3`, respectively.

Use the `rmse_score` function to compare the results from the different interpolation strategies. Start with CO2 data:

```
_ = rmse_score(co2_original, co2_missing, 'co2')
>>
RMSE for linear: 0.05507291327761665
RMSE for quadratic: 0.08367561505614354
RMSE for nearest: 0.05385422309469095
RMSE for cubic: 0.08373627305833194
RMSE for spline: 0.18786023475414115
RMSE for polynomial: 0.08367561505614354
```

Linear interpolation is ideal for datasets with consistent trends, while spline methods are better suited for data with smooth but non-linear patterns. For the CO2 data, the spline method has the highest RMSE because it introduces excessive variation when modeling smooth but linear trends, which isn't ideal for datasets with consistent growth patterns.

Notice that polynomial and quadratic produced the same RMSE. This is because quadratic interpolation is actually a special case of polynomial interpolation where the order is set to 2. When you use polynomial interpolation with `order=2`, you're essentially fitting a quadratic function, resulting in identical imputation values and therefore identical RMSE scores.

Now, let's check the clickstream data:

```
_ = rmse_score(clicks_original, clicks_missing, 'clicks')
>>
RMSE for linear: 1329.1448378562811
RMSE for quadratic: 5224.641260626975
RMSE for nearest: 1706.1853705030173
RMSE for cubic: 6199.304875782831
RMSE for spline: 5222.9229934486475
RMSE for polynomial: 5224.641260626975
```

Quadratic and cubic methods have higher RMSE scores for the clickstream dataset due to their tendency to exaggerate trends, which introduces errors in regions with sparse or inconsistent data.

5. Lastly, visualize the results to gain a better idea of how each interpolation worked. Start with the CO2 dataset:

```
cols = ['co2', 'linear', 'nearest', 'polynomial', 'cubic', 'spline']
plot_dfs(co2_original, co2_missing[cols], 'co2')
```

This should plot the selected columns:

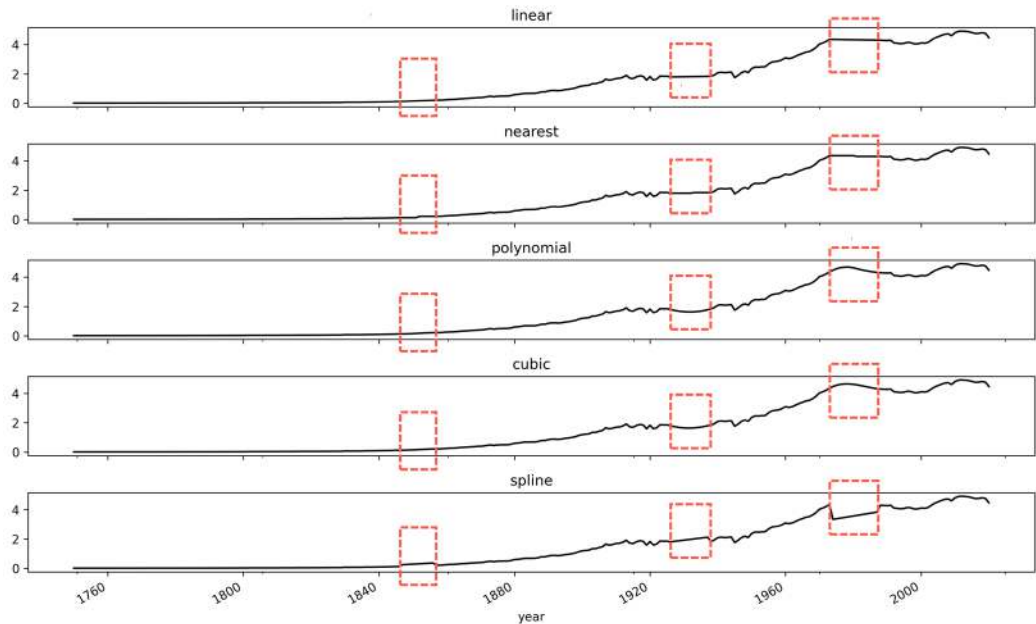


Figure 6.14: Comparing the different interpolation strategies on the CO2 dataset

Compare the results with the original data in Figure 6.5.

In Figure 6.14, the linear and nearest methods provide simpler interpolations that closely follow the original trend. The polynomial and spline methods introduce unnecessary variations, especially in regions with larger gaps.

Now, create the plots for the clickstream dataset:

```
cols = ['clicks', 'linear', 'nearest', 'polynomial', 'cubic', 'spline']
plot_dfs(clicks_original, clicks_missing[cols], 'clicks')
```

This should plot the selected columns:

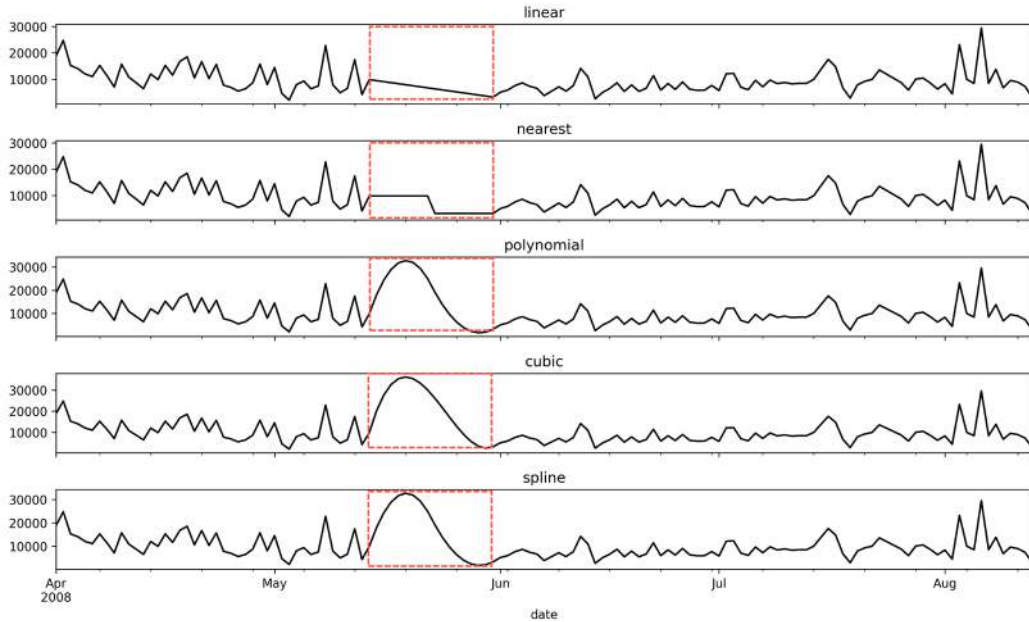


Figure 6.15: Comparing the different interpolation strategies on the clickstream dataset

Compare the results with the original data in Figure 6.6.

In Figure 6.15, the polynomial, cubic, and spline methods exaggerate the curve due to their higher flexibility, even with a polynomial order of 2. This results in overfitting (failing to generalize well by capturing noise rather than the underlying trend), especially in regions with sparse data. On the other hand, the linear method attempts to draw a straight line, which avoids overfitting and provides a simpler imputation.

Linear interpolation is ideal for datasets with steady trends, such as CO2 emissions, while spline or cubic interpolation may be better suited for datasets with smooth but non-linear patterns, such as certain clickstream data.

This highlights the trade-off between complexity and accuracy when selecting an interpolation method. While more complex methods, such as spline or polynomial interpolation, can capture non-linear trends, they may overfit in datasets with noise or sparse data. Simpler methods, such as linear or nearest interpolation, provide more stable and interpretable results for such cases.

How it works...

The interpolation technique estimates missing values by detecting patterns in neighboring data points to predict what the missing values should be. The simplest form is **linear interpolation**, which assumes a straight line between two neighboring data points. In contrast, **polynomial interpolation** uses a curve fitted across multiple adjacent data points, allowing it to capture more complex patterns between them. Each interpolation method uses a different function and mechanism to predict the missing data.

In pandas, you used the `DataFrame.interpolate` function to perform these operations. By default, this function applies linear interpolation (`method = "linear"`). Additional parameters provide control over the imputation process, such as specifying different methods or handling edge cases.

The `limit` parameter allows you to set the maximum number of consecutive NaN values to fill. Recall in the previous recipe, *Performing data quality checks*, that the clickstream dataset had 16 consecutive missing points. By setting a limit, you can control how many consecutive NaN values are imputed, preventing excessive filling that might not accurately reflect underlying trends.

For instance, you can limit the number of consecutive NaN to 5:

```
clicks_missing['clicks'].isna().sum()
>> 16

example = clicks_missing['clicks'].interpolate(limit = 5)
example.isna().sum()
>> 11
```

In this example, only 5 data points were imputed according to the specified limit, leaving 11 missing values unfilled.

Let's look at a comparison of the data before and after interpolation with a limit of 5.

Here is the before:

```
clicks_missing['clicks']['2008-05-15:'].head(10)
>>
date
2008-05-15    NaN
2008-05-16    NaN
2008-05-17    NaN
2008-05-18    NaN
2008-05-19    NaN
2008-05-20    NaN
2008-05-21    NaN
2008-05-22    NaN
2008-05-23    NaN
2008-05-24    NaN
Name: clicks, dtype: float64
```

Here is the after interpolation with `limit=5`:

```
example['2008-05-15:'].head(10)
>>
date
2008-05-15    9450.0
```

```
2008-05-16    9059.0
2008-05-17    8668.0
2008-05-18    8277.0
2008-05-19    7886.0
2008-05-20         NaN
2008-05-21         NaN
2008-05-22         NaN
2008-05-23         NaN
2008-05-24         NaN
Name: clicks, dtype: float64
```

This approach helps maintain data integrity by avoiding over-reliance on potentially unreliable interpolations for long sequences of missing data.

Lastly, it's worth noting that linear interpolation in pandas assumes the data points are equally spaced and disregards the numerical values of the index. In contrast, methods such as 'index', 'time', 'nearest', 'zero', 'slinear', 'quadratic', and 'cubic' consider the numerical index values when imputing missing data. This distinction can have a significant impact on the results, especially for datasets with non-sequential or categorical indices.

Choosing the right interpolation method is key to ensuring it aligns with the structure and behavior of your dataset.

There's more...

Other libraries also offer interpolation, including the following:

- **SciPy** provides a comprehensive selection of both univariate and multivariate interpolation techniques: <https://docs.scipy.org/doc/scipy/reference/interpolate.html>.
- **NumPy** offers basic interpolation options, with the most widely used being the `numpy.interp()` function. This function performs simple linear interpolation on one-dimensional arrays, making it a suitable choice for straightforward applications: <https://numpy.org/doc/stable/reference/generated/numpy.interp.html?highlight=interp#numpy.interp>.

See also

- To learn more about `DataFrame.interpolate`, visit the official documentation page here: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html>.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



7

Outlier Detection Using Statistical Methods

In addition to missing data, as discussed in *Chapter 6*, a common data issue you may face is the presence of **outliers**. Identifying and properly handling outliers is a crucial step in time-series analysis, as even a small number of extreme values can significantly impact forecasting models, distort statistical properties, lead to biased parameter estimates, and result in inaccurate forecasts that could impact decisions. Outliers can appear in several forms: point outliers, collective outliers, or contextual outliers. For example, a **point outlier** occurs when a data point deviates significantly from the rest of the population—sometimes referred to as a **global outlier**. **Collective outliers**, which are groups of observations, differ from the population and don't follow the expected pattern. Lastly, **contextual outliers** occur when an observation is considered an outlier based on a particular condition or context, such as deviation from neighboring data points. Note that with contextual outliers, the same observation may not be considered an outlier if the context changes.

In this chapter, you will be introduced to a handful of practical statistical techniques that cover **parametric** and **non-parametric** methods. Parametric methods, such as z-score, assume that the data follows a specific distribution (e.g., Gaussian), while non-parametric methods, such as the Tukey method, do not make such assumptions, making them more robust to deviations from normality. In *Chapter 13*, you will dive into more advanced machine learning and deep learning-based techniques.

In the literature, you will find another popular term, anomaly detection, which can be synonymous with outlier detection. The methods and techniques to identify outlier or anomaly observations are similar; the difference lies in the context and the actions that follow once these points have been identified. For example, an outlier transaction in financial transactions may be referred to as an anomaly and trigger a fraud investigation to stop it from re-occurring. Under a different context, survey data with outlier data points may simply be removed by the researchers once they examine the overall impact of keeping versus removing such points. Sometimes you may decide to keep these outlier points if they are part of the natural process. In other words, they are legitimate and so you may opt to use robust statistical methods that are not influenced by outliers.

Another concept, known as **change point detection** (CPD), relates to outlier detection. In CPD, the goal is to anticipate abrupt and impactful fluctuations (increasing or decreasing) in the time-series data. CPD covers specific techniques, for example, cumulative sum (CUSUM) and Bayesian online change point detection (BOCPD). Detecting change is vital in many situations. For example, a machine may break if the internal temperature reaches a certain point, or you may want to understand whether a discounted price increased sales. This distinction between outlier detection and CPD is vital since sometimes you may want the latter. Where the two disciplines converge, depending on the context, sudden changes may indicate the potential presence of outliers (anomalies).

The recipes that will be covered in this chapter are as follows:

- Resampling time-series data
- Detecting outliers using visualizations
- Detecting outliers using the Tukey method
- Detecting outliers using a z-score
- Detecting outliers using a modified z-score
- Detecting outliers using the Hampel filter

Technical requirements

You can download the Jupyter notebooks and the needed datasets from the GitHub repository:

- **Jupyter Notebook:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch7/Chapter_7.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch7>

Throughout the chapter, you will be using a dataset from the **Numenta Anomaly Benchmark** (NAB), which provides outlier detection benchmark datasets. For more information about NAB, please visit their GitHub repository here: <https://github.com/numenta/NAB>.

The *New York Taxi dataset* captures the number of NYC taxi passengers at a specific timestamp. The data contains known anomalies that are provided to evaluate the performance of our outlier detectors. The dataset contains 10,320 records between *July 1, 2014*, and *May 31, 2015*. The observations are captured in 30-minute intervals, which translates to `freq = '30min'`.

To prepare for the outlier detection recipes, start by loading the libraries that you will be using throughout the chapter:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
plt.rcParams["figure.figsize"] = [12, 5]
```

Load the `nyc_taxi.csv` data into a pandas DataFrame as it will be used throughout the chapter:

```
file = Path("../datasets/Ch7/nyc_taxi.csv")
nyc_taxi = pd.read_csv(file,
```

```
index_col='timestamp',  
parse_dates=True)  
  
nyc_taxi.index.freq = '30min'
```

You can store the known dates containing outliers, also known as ground-truth labels:

```
nyc_dates = [  
    "2014-11-01",  
    "2014-11-27",  
    "2014-12-25",  
    "2015-01-01",  
    "2015-01-27"  
]
```

If you investigate these dates to gain more insight into their significance, you will find similar information to the following summary:

- *Saturday, November 1, 2014*, was before the New York Marathon, and the official marathon event was on Sunday, November 2, 2014
- *Thursday, November 27, 2014*, was Thanksgiving Day
- *Thursday, December 25, 2014*, was Christmas Day
- *Thursday, January 1, 2015*, was New Year's Day
- *Tuesday, January 27, 2015*, was the North American blizzard, where all vehicles were ordered off the street from January 26 to January 27, 2015

You can plot the time-series data to gain an intuition on the data you will be working with for outlier detection:

```
nyc_taxi.plot(title="NYC Taxi", alpha=0.6);
```

This should produce a time series with a 30-minute frequency:

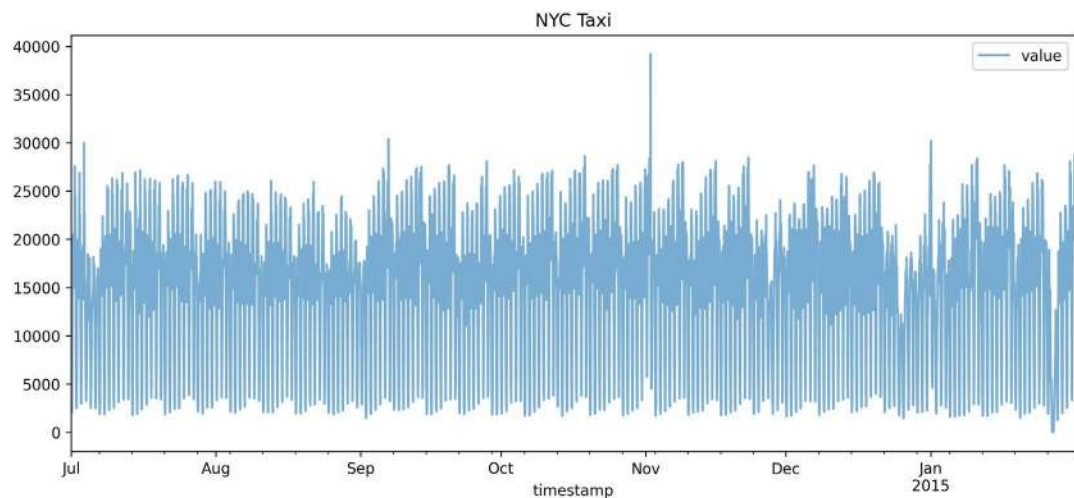


Figure 7.1: Plot of the New York City taxi time-series data

For consistency and cleaner code, the visualization functions used in this chapter are stored in a separate `utils.py` script. You can import them as follows:

```
from utils import plot_outliers, plot_zscore
```

These two utility functions will help visualize outliers and interpret statistical detection results:

- `plot_outliers` highlights anomalous points within your time-series data, making it easier to spot unusual patterns and assess whether they correspond to known events or potential anomalies.
- `plot_zscore` plots standardized z-scores along with threshold lines, providing an intuitive way to see how statistical thresholds isolate extreme values.

Both of these functions will be used throughout this chapter to visualize and validate the performance of different outlier detection methods.

As we proceed with the outlier detection recipes, the goal is to see how the different techniques capture outliers and compare them to the ground-truth labels.

The nature of outliers in time-series data

The presence of outliers requires special handling and further investigation before hastily jumping to decisions on how to handle them. First, you will need to detect and spot their existence, which this chapter is all about. Domain knowledge can be instrumental in determining whether these identified points are outliers, their impact on your analysis, and how you should deal with them.

Outliers can indicate bad data due to a random variation in the process, known as noise, or due to a data entry error, faulty sensors, a bad experiment, or natural variation. Outliers are usually undesirable if they seem synthetic, for example, bad data. On the other hand, if outliers are a natural part of the process, you may need to rethink removing them and opt to keep these data points. In such circumstances, you can rely on **non-parametric** statistical methods that do not make assumptions about the underlying distribution.

Generally, outliers can cause side effects when building a model based on strong assumptions about the data distribution; for example, the data is from a Gaussian (normal) distribution. Statistical methods and tests based on assumptions of the underlying distribution are referred to as **parametric methods** and can be significantly impacted by outliers.

There is no universal protocol for dealing with outliers, and the magnitude of their impact will vary. For example, sometimes you may need to test your model with outliers and again without outliers to understand the overall impact on your analysis. In other words, not all outliers are created—nor should they be treated—equally. However, as stated earlier, having domain knowledge is essential when dealing with these outliers.

Now, before using a dataset to build a model, you will need to test for the presence of such outliers so you can further investigate their significance. Spotting outliers is usually part of the data cleansing and preparation process before going deep into your analysis.

A common approach to handling outliers is to delete these data points and not have them be part of the analysis or model development. Alternatively, you may wish to replace the outliers using similar techniques highlighted in *Chapter 6*, such as imputation and interpolation. Other methods, such as smoothing the data, could minimize the impact of outliers. Smoothing, such as exponential smoothing, is discussed in *Chapter 9*. These smoothing techniques can preserve the temporal patterns and seasonality in time-series data while reducing the influence of outlier observations that might otherwise derail your forecasting models. You may also opt to keep the outliers and use more resilient algorithms to their effect.

There are many well-known methods for outlier detection. The area of research is evolving, ranging from basic statistical techniques to more advanced approaches leveraging neural networks and deep learning. In statistical methods, you have different tools that you can leverage, such as the use of visualizations (box plots, **quantile-quantile (QQ)** plots, histograms, and scatter plots), **z-score**, **interquartile range (IQR)**, and **Tukey fences**, and statistical tests such as Grubbs' test, the Tietjen-Moore test, or the generalized extreme Studentized deviate (ESD) test. These are basic, easy-to-interpret, and effective methods.

In the first recipe, you will be introduced to a crucial time-series transformation technique known as resampling before diving into outlier detection.

Resampling time-series data

A typical transformation that is done on time-series data is **resampling**. The process implies changing the frequency or level of granularity of the data.

Usually, you will have limited control over how the time series is generated in terms of frequency. For example, the data can be generated and stored in small intervals, such as milliseconds, minutes, or hours. In some cases, the data can be in larger intervals, such as daily, weekly, or monthly. The frequency at which you analyze your data can significantly impact your ability to detect certain types of outliers.

The need for resampling time series can be driven by the nature of your analysis and at what granular level you need your data to be. For instance, you can have daily data, but your analysis requires the data to be weekly, and thus you will need to resample. This process is known as **downsampling**. When you are downsampling, you will need to provide some level of aggregation, such as mean, sum, min, or max, to name a few. On the other hand, some situations require you to resample your data from daily to hourly. This process is known as **upsampling**. When upsampling, you may introduce null rows, which you can fill using imputation or interpolation techniques. See *Chapter 6*, where both imputation and interpolation methods were discussed in more detail.

In the context of **outlier detection**, resampling can significantly impact which anomalies you might discover. For example, when downsampling, you might smooth out short-term outliers by aggregating them with normal data points. This is especially evident in our NYC taxi dataset—a brief spike in passengers might disappear when aggregated into daily totals.

On the other hand, downsampling can actually make some outliers more visible. Holiday anomalies such as Christmas Day (December 25, 2014) may stand out more clearly when viewed at a weekly frequency rather than 30-minute intervals.

When upsampling, the interpolation methods you choose to fill gaps can create new patterns that might either highlight or obscure contextual outliers. For instance, upsampling our taxi data from 30 minutes to 15 minutes using forward fill would duplicate the passenger counts, potentially affecting how anomalies appear.

Resampling also serves as an important preprocessing step by ensuring your data has uniform intervals, which many outlier detection algorithms require. What appears unusual in hourly data might simply be part of a normal pattern when viewed daily.

The choice of resampling method (mean, median, sum, etc.) and frequency should be guided by your analysis objectives and the types of outliers you're looking for. It's often helpful to try multiple approaches—examining your data at different granularity levels can reveal outliers that might otherwise remain hidden.

In this recipe, you will explore how resampling is done using the pandas library.

How to do it...

Resampling transforms your time-series data by changing its frequency, which can have significant implications for outlier detection.

In this recipe, you will work with the `nyc_taxi` DataFrame, created earlier in the *Technical requirements* section. The data captures the number of passengers in *30-minute* intervals.

Resampling can be a valuable preprocessing step for outlier detection. By aggregating data to a lower frequency (downsampling), you can smooth out short-term noise and make it easier to identify unusual patterns or global outliers. You will first learn how downsampling works, then move on to upsampling:

1. **Downsample the data to a daily frequency:** Currently, you have 10,320 records, and when you resample the data to daily frequency, you will need to aggregate the data. In this example, you will use the `.mean()` function. This will reduce the number of samples to 215 records. The term *downsampling* indicates that the number of samples went down, and more concretely, we are decreasing the frequency of the data.

Inspect the first five rows of the original DataFrame:

```
nyc_taxi.head()
>>

```

timestamp	value
2014-07-01 00:00:00	10844
2014-07-01 00:30:00	8127
2014-07-01 01:00:00	6210
2014-07-01 01:30:00	4656
2014-07-01 02:00:00	3820

```
print(nyc_taxi.index.freq)
>>
<30 * Minutes>
```

Resampling is done using the `DataFrame.resample()` function. For daily, you will use `'D'` as the date offset rule, followed by `.mean()` as the method of aggregation to get daily averages:

```
df_downsampled = nyc_taxi.resample('D').mean()
df_downsampled.head()
>>
           value
timestamp
2014-07-01  15540.979167
2014-07-02  15284.166667
2014-07-03  14794.625000
2014-07-04  11511.770833
2014-07-05  11572.291667
```

Notice how `DatetimeIndex` is now at a daily frequency, and the number of passengers now reflects the daily average. Inspect the first `DatetimeIndex` to check its frequency using the `.freq` property:

```
print(df_downsampled.index.freq)
>>
<Day>
```

Check the number of records now after downsampling:

```
df_downsampled.shape
>>
(215, 1)
```

Indeed, now you have 215 records.

2. **Resample the `nyc_taxi` data again, but this time as a three-day frequency:** You can do this by using the `'3D'` offset string. This time, use the `.sum()` method instead for a cumulative view:

```
df_downsampled = nyc_taxi.resample('3D').sum()
df_downsampled.head()
>>
           value
timestamp
2014-07-01  2189749
2014-07-04  1658320
2014-07-07  2111672
2014-07-10  2335629
2014-07-13  2139711
```


Check the frequency of `DatetimeIndex`:

```
print(df_downsampled.index.freq)
>>
<3 * Days>
```

If you use `df_downsampled.shape`, you will notice the number of records reduced to 72.

- Now, change the frequency to three business days instead. The default in pandas is Monday to Friday. In the *Working with custom business days* recipe in *Chapter 5*, you learned how to create custom business days. For now, you will use the default definition of business days. If you observe the output from the previous step, 2014-07-13 falls on a Sunday. Using '3B' as the `DateOffset` value will push it to the following Monday, 2014-07-14:

```
df_downsampled = nyc_taxi.resample('3B').sum()
df_downsampled.head()
>>
```

	value
timestamp	
2014-07-01	2189749
2014-07-04	3021425
2014-07-09	3799847
2014-07-14	2174540
2014-07-17	3711232

What is interesting about the output is how it skips five days from 2014-07-04 to 2014-07-09, and then again from 2014-07-09 to 2014-07-14. The reason is the **business day** rule, which specifies that we have two days of the week as weekends. Since the function is calendar-aware, it knows a weekend is coming after the first three-day increment, so it adds a two-day weekend to skip them, and thus, it makes a five-day jump. It starts from 2014-07-04, then moves to 2014-07-09, and from 2017-07-09, it moves to 2017-07-14. Similarly, it moves from 2014-07-17 to 2014-07-22, and so on.

- Lastly, let's upsample the data from a 30-minute interval (frequency) to a 15-minute frequency: This will create an empty entry (NaN) between every other entry. You will use the 'min' offset string:

```
df_upsampled = nyc_taxi.resample('15min').mean()
df_upsampled.head()
>>
```

	value
timestamp	
2014-07-01 00:00:00	10844.0
2014-07-01 00:15:00	NaN
2014-07-01 00:30:00	8127.0
2014-07-01 00:45:00	NaN
2014-07-01 01:00:00	6210.0

Notice that upsampling creates NaN rows. Unlike *downsampling*, when *upsampling*, you need to give instructions on how to fill the NaN rows. You might be wondering why we used `.mean()` here. The simple answer is that it would not matter whether you used `.sum()`, `.max()`, or `.min()`, for example. It is also because pandas `resample()` always requires an aggregation function, even if it's a placeholder before imputation. You will need to augment the missing rows using imputation or interpolation techniques when you *upsample*. For example, you can specify an imputation value using `.fillna()` or by using the `.ffill()` or `.bfill()` method:

```
df_upsampled = nyc_taxi.resample('15min').ffill()
df_upsampled.head()
>>
```

timestamp	value
2014-07-01 00:00:00	10844
2014-07-01 00:15:00	10844
2014-07-01 00:30:00	8127
2014-07-01 00:45:00	8127
2014-07-01 01:00:00	6210

The first five records show the use of forward filling. For more information on using `.fillna()`, `.bfill()`, or `.ffill()`, or imputation in general, refer to *Chapter 6*.

Overall, resampling in pandas is very convenient and straightforward. This can be a handy tool when you want to change the frequency of your time series.

How it works...

The `DataFrame.resample()` method allows you to group rows in a specified time frame, for example, by day, week, month, year, or any datetime attribute. The way `.resample()` works is by grouping the data using `DatetimeIndex` and the frequency provided; hence, this method is specific to time-series DataFrames.

The `.resample()` function works in a very similar manner to the `.groupby()` function; the difference is that `.resample()` is specific to time-series data and groups at `DatetimeIndex`. When you call `.resample()` on a `DataFrame`, it returns a `Resampler` object, which is similar to the `GroupBy` object.

The resampling process follows these steps:

- The data is grouped based on the specified frequency
- For **downsampling**, an aggregation function (such as `.mean()`, `.sum()`, `.min()`, or `.max()`) must be applied to combine multiple values into one
- For **upsampling**, new rows are created with NaN values, which can then be filled using methods such as `.ffill()`, `.bfill()`, or custom interpolation

The offset strings used in resampling follow pandas' `DateOffset` aliases. Some common ones include the following:

- `'D'`: Calendar day
- `'B'`: Business day
- `'W'`: Weekly
- `'ME'`: Month end
- `'MS'`: Month start
- `'QE'`: Quarter end
- `'QS'`: Quarter start
- `'YE'`: Year end
- `'YS'`: Year start
- `'h'`: Hourly
- `'min'`: Minute
- `'S'`: Second

You can also specify multiples of these frequencies, as demonstrated with `'3D'` for every three days or `'15min'` for every 15 minutes.

For outlier detection purposes, downsampling to daily frequency (which we'll use in upcoming recipes) offers a strategic advantage. While downsampling does smooth out the data by combining multiple observations, it doesn't necessarily eliminate evidence of outliers. In fact, when using the *mean* as your aggregation method, extreme values continue to influence the daily average.

For instance, if there were an extreme outlier during one of the 48 30-minute intervals in our NYC taxi data on a particular day, the *daily mean* would still reflect this abnormality.

This preservation of outlier information is particularly valuable for our analysis. Rather than trying to identify which specific 30-minute interval contained an anomaly, our focus in the upcoming recipes will be on detecting *which days exhibited unusual passenger patterns overall*. The daily frequency provides a more manageable scale for applying statistical methods such as z-score and Tukey fences while still retaining the essential signal of anomalous behavior.

It's worth noting that your choice of aggregation method should align with your outlier detection goals. The mean is particularly sensitive to outliers, which makes it suitable when you want outliers to remain visible after downsampling. Other aggregation methods, such as the *median*, would be more robust against outliers, potentially masking the very anomalies you're trying to detect. Similarly, *sum* aggregation would amplify absolute differences, while *min* or *max* would highlight only the extreme points within each time window.

This relationship between resampling strategy and outlier visibility underscores why resampling is an important first step in our outlier detection workflow, setting the foundation for the statistical techniques we'll explore in subsequent recipes.

There's more...

You can supply more than one aggregation at once when downsampling using the `.agg()` function. For example, by using 'ME' for month end, you can supply the `.agg()` function with a list of aggregations you want to perform:

```
nyc_taxi.resample('ME').agg(['mean',
                             'min',
                             'max',
                             'median',
                             'sum'])
```



The 'M' string is being deprecated and will be removed in a future version. It should be replaced with 'ME'.

This should produce a DataFrame with five columns, one for each aggregation method specified. The index column, a timestamp column, will be grouped at monthly intervals:

	value				
	mean	min	max	median	sum
timestamp					
2014-07-31	14994.084677	1769	29985	16625.5	22311198
2014-08-31	14580.438844	1841	26062	16184.0	21695693
2014-09-30	15623.374306	1431	30373	17244.5	22497659
2014-10-31	16086.851478	1691	28626	17767.5	23937235
2014-11-30	15492.125000	1683	39197	17287.0	22308660
2014-12-31	14813.428763	1459	27804	16587.0	22042382
2015-01-31	14399.790995	8	30236	16061.0	21426889

You can change to month start instead by using 'MS'. For example, this will produce *2014-07-01* instead (the beginning of each month).

You'll notice that the resulting DataFrame has a hierarchical column index (`MultiIndex`). If you prefer simpler column names, you can rename them after the aggregation:

```
# Rename the columns
result.columns = ['mean', 'min', 'max', 'median', 'sum']
```

This gives you a DataFrame with single-level column names.

This multiple aggregation approach can be particularly valuable for outlier detection as it allows you to examine different statistical properties simultaneously. For instance, looking at the January 2015 data in our example, we can see a minimum value of 8 passengers, which stands out dramatically compared to other months' minimums (all above 1,400). This corresponds to the North American blizzard we mentioned earlier.

See also

- To learn more about pandas' `DataFrame.resample()`, please visit the official documentation here: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.resample.html>.

Detecting outliers using visualizations

In time-series analysis, outlier detection is particularly important as anomalous data points can significantly impact forecasting accuracy and model performance. Using visualizations is an effective step in identifying potential outliers at a glance before applying more complex statistical methods.

There are two general approaches for using statistical techniques to detect outliers—**parametric** and **non-parametric** methods:

- **Parametric** methods assume you know the underlying distribution of the data, such as a normal distribution
- On the other hand, in **non-parametric** methods, you make no such assumptions about the data's distribution, making them particularly useful for exploratory data analysis

Histograms and **box plots** are basic non-parametric techniques that can provide insight into the distribution of the data and the presence of outliers. Box plots, also known as box and whisker plots, provide a five-number summary: the *minimum*, *first quartile* (25th percentile), *median* (50th percentile), *third quartile* (75th percentile), and *maximum*. There are different implementations for how far the whiskers extend; for example, the whiskers can extend to the minimum and maximum values. In common statistical software, including Python's **Matplotlib** and **Seaborn** libraries, the whiskers extend to what is called *Tukey's lower and upper fences*, which are based on the IQR.

Any data point outside these boundaries is considered a potential outlier. You will dive into the actual calculation and implementation in the *Detecting outliers using the Tukey method* recipe. For now, let's focus on the visualization aspect of the analysis.

In this recipe, you will use Seaborn, a powerful Python visualization library based on Matplotlib, to explore several visualization techniques, including histograms, box plots, boxen plots, and violin plots—each offering unique insights into your time-series data and potential outliers.

Getting ready

You can download the Jupyter notebooks and required datasets from the GitHub repository. Please refer to the *Technical requirements* section of this chapter. You will be using the `nyc_taxi` DataFrame that you loaded earlier in the *Technical requirements* section, which contains NYC taxi passenger counts recorded at 30-minute intervals.

For this recipe, you'll use Seaborn, a statistical data visualization library that simplifies the creation of informative and aesthetically pleasing plots built on top of Matplotlib. Its integration with pandas DataFrames makes it great for time-series visualization.

You will be using Seaborn version *0.13.2*, which is the latest version as of the time of writing. You can use a more recent version if available, as the core functionality used in this recipe should be stable across recent Seaborn versions.

To install Seaborn using pip, use the following:

```
pip install seaborn
```

To install Seaborn using conda, use the following:

```
conda install seaborn -c conda-forge
```

How to do it...

In this recipe, you will explore different plots available from Seaborn, including `histplot()`, `displot()`, `boxplot()`, `boxenplot()`, and `violinplot()`.

You will notice that these plots tell a similar story, but visually, each plot represents the information differently. Eventually, you will develop a preference toward some of these plots for your own use when investigating your data:

1. Start by importing the Seaborn library to begin exploring how these plots work to help you detect outliers:

```
import seaborn as sns
```

2. Recall that the `nyc_taxi` DataFrame contains passenger counts recorded every 30 minutes. Keep in mind that every analysis or investigation is unique, and so should be your approach to align with the problem you are solving. This also means that you will need to consider your data preparation approach, for example, determine what transformations you need to apply to your data.

For this recipe, the goal is to find which days have outlier observations, not at which interval within the day. To achieve this, you will *resample* the data to a daily frequency using the mean aggregation (downsampling). Even though such a transformation will smooth out the data, you will not lose too much of the details as it pertains to finding outliers since the mean is sensitive to outliers. In other words, extreme values (outliers) on specific intervals (there are 48 intervals in a day), even after resampling, will still influence the daily average—the mean will still carry that information.

In this step, you will downsample the data to a daily frequency using the `.resample()` method and using 'D' as the offset string. This will reduce the number of observations from 10,320 to 215 (which is $10320/48 = 215$):

```
tx = nyc_taxi.resample('D').mean()
```

3. To visualize the known outliers, use the `plot_outliers` function introduced in the *Technical requirements* section. This function plots the time-series data and marks the outliers with red X markers. The `nyc_dates` list contains the known outlier dates (ground truth):

```
known_outliers= tx.loc[nyc_dates]
plot_outliers(known_outliers, tx, 'Known Outliers', labels=True)
```

This produces a time-series plot (*Figure 7.2*) with the known outliers highlighted. The `labels=True` argument adds the date labels to the outliers for better interpretability.

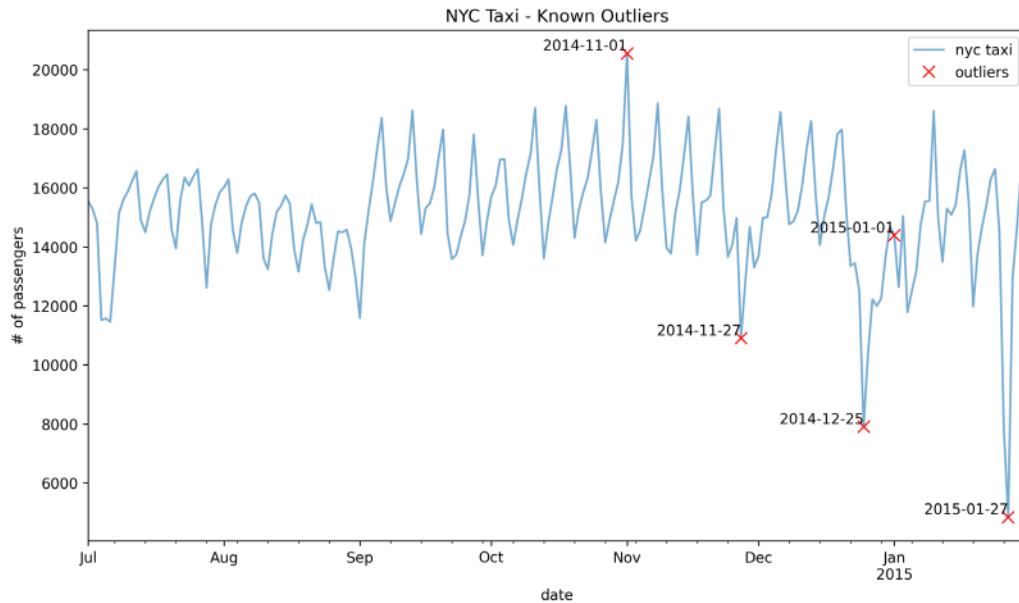


Figure 7.2: Plotting the NYC taxi data after downsampling with ground-truth labels (outliers)

4. Start by creating a histogram to inspect the distribution of daily average passenger counts using the `histplot()` function:

```
sns.histplot(tx);
```

This should produce the following:

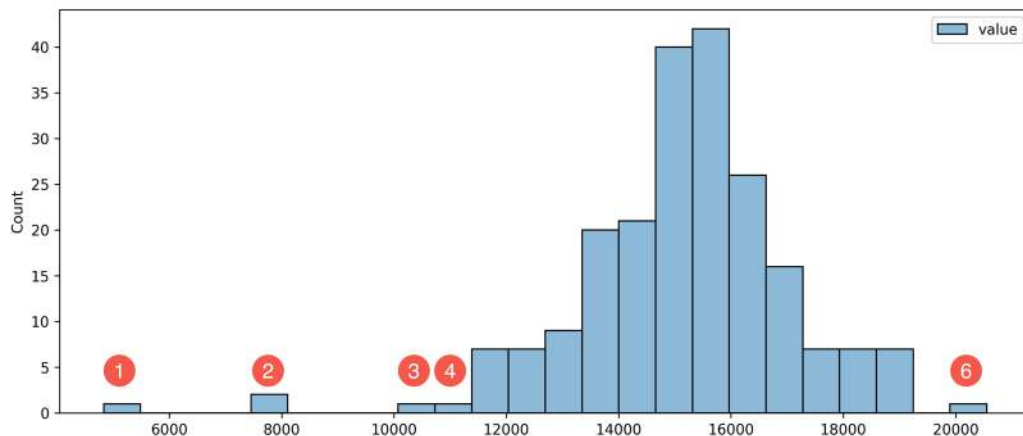


Figure 7.3: Histogram showing the distribution of daily mean passenger rides

In *Figure 7.3*, the observations labeled as 1, 2, 3, 4, and 5 seem to represent extreme passenger values. Recall that these numbers represent the average daily passengers after resampling. The question you should ask is whether these observations are outliers. The center of the histogram is close to 15,000 daily average passengers. This should make you question whether the extreme value close to 20,000 (*label 5*) is that extreme. Similarly, with the observations labeled 3 and 4 (since they are close to the tail of the distribution), are they actually extreme values? How about labels 1 and 2 with average passenger rides at 3,000 and 8,000 daily average passengers, respectively? These do seem more extreme compared to the rest and may potentially be actual outliers. Again, determining what is an outlier and what is not requires domain knowledge and further analysis. There is no specific rule, and you will see throughout this chapter that some of the generally accepted rules are arbitrary and subjective. You should not jump to conclusions immediately.

You can achieve a similar plot using `displot()`, which has a `kind` parameter. The `kind` parameter can take one of three values: `hist` for the histogram plot, `kde` for the kernel density estimate plot, and `ecdf` for the empirical **cumulative distribution function** (CDF) plot.

5. You will use `displot(kind='hist')` to plot a similar histogram to the one in *Figure 7.3*:

```
sns.displot(tx, kind='hist', height=5, aspect=2);
```

6. A box plot provides more information than a histogram and can be a better choice for spotting outliers. In a box plot, observations that are outside the whiskers or boundaries are considered potential outliers. The whiskers represent the visual boundary for the upper and lower fences as proposed by mathematician John Tukey in 1977.

The following code shows how to create a box plot using Seaborn:

```
sns.boxplot(tx['value'], orient='h')
```

The box plot in *Figure 7.4* shows potential outliers outside the whiskers:

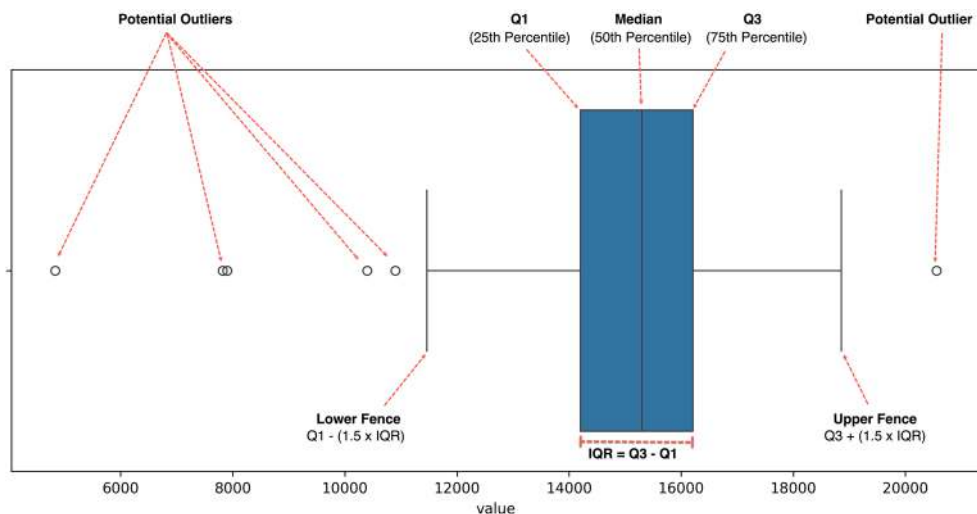


Figure 7.4: A box plot showing potential outliers that are outside the boundaries (whiskers)

The width of the box ($Q1$ to $Q3$) is called the interquartile range, or IQR, calculated as the difference between the 75th and 25th percentiles ($Q3 - Q1$). The lower fence is calculated as $Q1 - (1.5 \times IQR)$, and the upper fence as $Q3 + (1.5 \times IQR)$. Any observation less than the lower boundary or greater than the upper boundary is considered a potential outlier. More on that in the *Detecting outliers using the Tukey method* recipe. The `whis` parameter in the `boxplot()` function is set to 1.5 by default (1.5 times the IQR), which controls the width or distance between the upper and lower fences. Larger values mean fewer observations will be deemed as outliers, and smaller values will make non-outlier points seem outside of the boundaries (more outliers). The preceding code will produce the same plot as in *Figure 7.4*:

```
sns.boxplot(tx['value'], orient='h', whis=1.5)
```

You can adjust the `whis` parameter to change the width of the whiskers.

- There are two more variations for box plots in Seaborn (`boxenplot()` and `violinplot()`). They provide similar insight to the box plot but are presented differently. The **boxen plot**, which in literature is referred to as a **letter-value plot**, can be considered as an enhancement to regular box plots to address some of their shortcomings, as described in the paper *Heike Hofmann, Hadley Wickham & Karen Kafadar (2017) Letter-Value Plots: Boxplots for Large Data, Journal of Computational and Graphical Statistics, 26:3, 469-477*. More specifically, boxen plots are better suited when working with larger datasets (a higher number of observations for displaying data distribution and more suitable for differentiating outlier points for larger datasets). The Seaborn implementation of `boxenplot` is based on that paper.

The following code shows how to create a boxen plot using Seaborn:

```
sns.boxenplot(tx['value'], orient='h');
```

This should produce a plot that looks like the box plot in *Figure 7.4*, but with boxes extending beyond the quartiles ($Q1$, $Q2$, and $Q3$). To get the exact values for $Q1$, $Q2$, and $Q3$, you can use the `describe()` method:

```
print(tx.describe())
>>
              value
count      215.000000
mean      15137.569380
std       1937.391020
min       4834.541667
25%      14205.197917
50%      15299.937500
75%      16209.427083
max       20553.500000
```

The 25th percentile is at the 14,205 daily average passengers mark, and the 75th percentile is at the 16,209 daily average passengers mark.

In *Figure 7.5*, you are getting additional insight into the distribution of passengers by visualizing more quantiles than a traditional box plot:

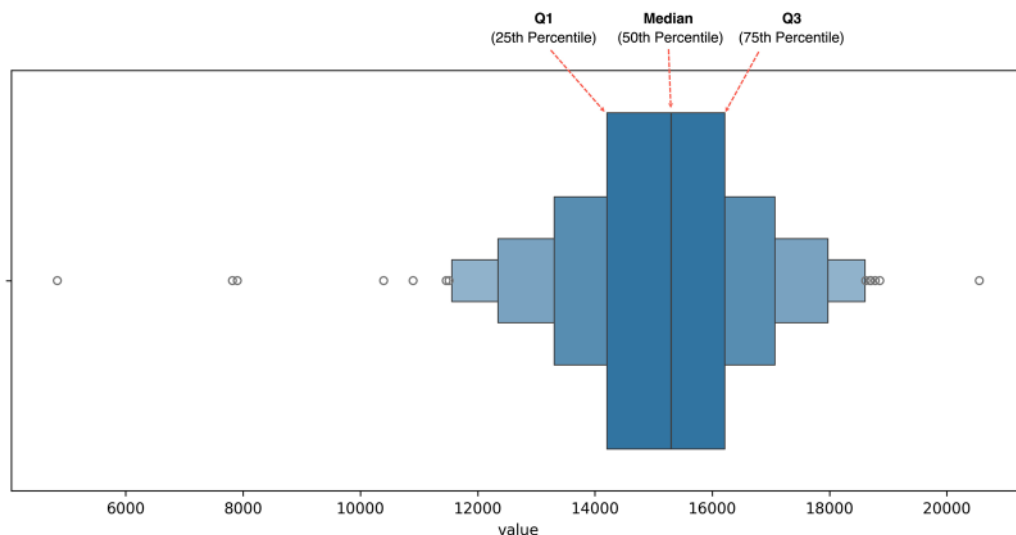


Figure 7.5: A boxen (letter-value) plot for the average daily taxi passengers in NYC

In other words, it extends the box plot to show additional quantile distributions to give more insight into the tails of the data. The boxes, in theory, could keep going to accommodate all the data points, but in order to show outliers, there needs to be a stopping point, referred to as the depth. In Seaborn, this parameter is called `k_depth`, which can take a numeric value, or you can specify different methods such as `tukey`, `proportion`, `trustworthy`, or `full`.

For example, `k_depth=1` will show just the central box (representing the range from Q1 to Q3, similar to a regular box plot). As a reference, *Figure 7.5* shows four boxes determined using the Tukey method, which is the default value (`k_depth="tukey"`). The Tukey method uses the formula $\log_2(n) - 3$ to determine the number of levels, where n is the sample size. In this case, our data resulted in four boxes. Using `k_depth=4` would produce the same plot.

These methods are explained in the referenced paper by *Heike Hofmann, Hadley Wickham & Karen Kafadar (2017)*. To explore the different methods, you can try the following code:

```
# exploring different methods
for k in ["tukey", "proportion", "trustworthy", "full"]:
    sns.boxenplot(tx['value'], k_depth=k, orient='h')
    plt.title(f'k_depth={k}')
    plt.show()
```

This should produce four plots; notice the different numbers of boxes that were determined by each method. Recall that you can also specify `k_depth` numerically.

8. Another variation is the **violin plot**, which you can create using the `violinplot` function:

```
sns.violinplot(tx['value'], inner='quartile', orient='h');
```

This should produce a plot that is a hybrid between a box plot and a **kernel density estimation (KDE)**. A kernel is a function that estimates the probability density function; the larger peaks (wider area), for example, show where the majority of the points are concentrated. This means that there is a higher probability that a data point will be in that region as opposed to the much thinner regions showing much lower probability.

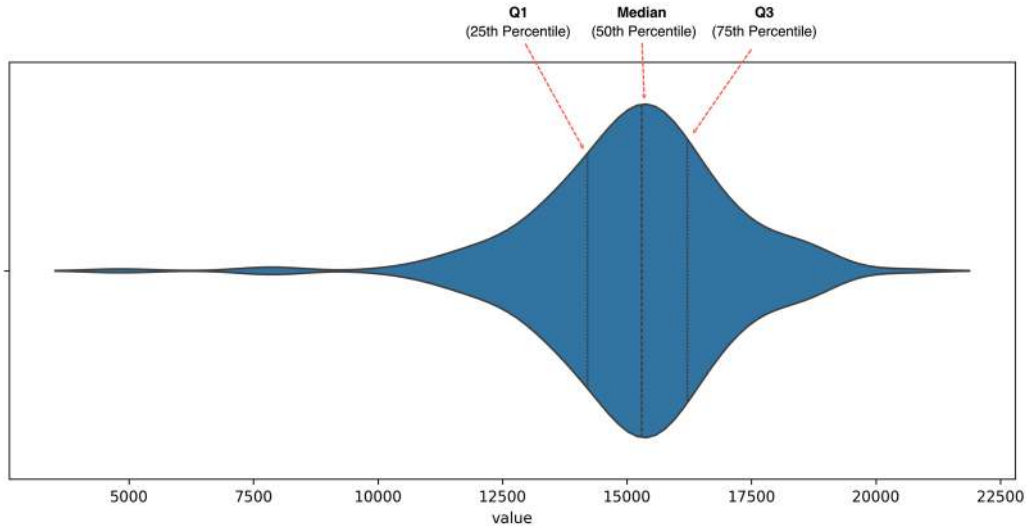


Figure 7.6: A violin plot for the average daily taxi passengers in NYC

Notice that *Figure 7.6* shows the distribution for the entire dataset. Another observation is the number of peaks; in this case, we have one peak, which makes it a unimodal distribution. If there is more than one peak, we call it a multimodal distribution, which should trigger a further investigation into the data. A KDE plot will provide similar insights as a histogram but with a smoother curve.

How it works...

In the recipe, we were introduced to several plots that help visualize the distribution of the data and show outliers. **Histograms**, such as the one in *Figure 7.3*, are excellent for showing the overall distribution by grouping data into bins, but they don't provide a clear threshold for what constitutes an outlier.

Box plots, such as the one in *Figure 7.4*, are much better for outlier detection because they establish statistical boundaries (the whiskers) based on the IQR. Points beyond these boundaries are explicitly highlighted as potential outliers. This makes box plots a powerful tool for quickly identifying extreme values in your data.

The **box plot**, shown in *Figure 7.5*, extends this concept by showing additional quantile information, making it more suited for larger datasets where a traditional box plot might oversimplify the distribution where outliers typically reside.

The **violin plot**, shown in *Figure 7.6*, combines the advantages of both approaches—showing the full distribution like a histogram while maintaining the central box from the box plot. This gives you both the overall shape and the statistical summary in one visualization, though outliers are not explicitly marked as in box plots.

While visualizations are a great starting point for identifying outliers, they're just one piece of the puzzle. In the following recipes, we'll dive into more quantitative methods, such as the Tukey method and z-scores, to further validate our outlier detection process.

There's more...

In Seaborn, both `histplot()` and `displot()` support adding the KDE plot with the `kde` parameter. This allows you to overlay a smoothed estimate of the distribution on top of the histogram, providing a more detailed view of the data's shape. The following code snippets should produce a similar plot:

```
sns.histplot(tx['value'], kde=True)
sns.displot(tx['value'], kind='hist', height=5, aspect=2, kde=True)
```

This should produce a plot that combines a **histogram** and a **KDE** plot, as shown in the following figure:

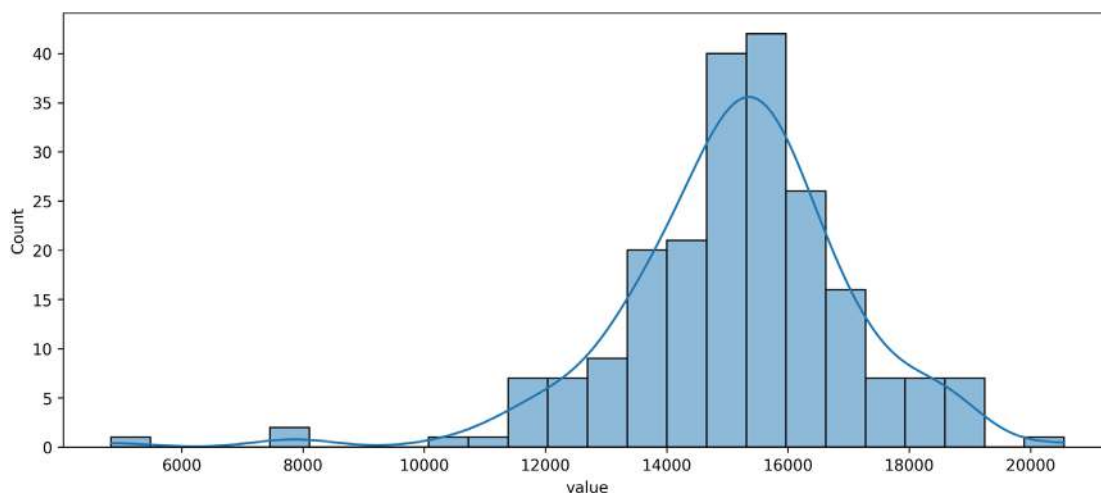


Figure 7.7: A histogram with a KDE plot

Another useful visualization for spotting outliers is the **lag plot**. A lag plot is essentially a scatter plot, but instead of plotting two variables to observe correlation, as an example, we plot the same variable against its lagged version (relationships between a variable and its lagged version). This means it is a scatter plot using the same variable, but the y axis represents passenger count at the current time (t) and the x axis shows the passenger count at a prior period ($t-1$), which we call the **lag**. The lag parameter determines how many periods to go back; for example, a lag of 1 means one period back, while a lag of 2 means two periods back. In our resampled data (downsampled to daily), a lag of 1 represents the prior day.

The pandas library provides the `lag_plot` function, which you can use as shown in the following example:

```
from pandas.plotting import lag_plot
lag_plot(tx['value']);
```

This should produce the following lag plot:

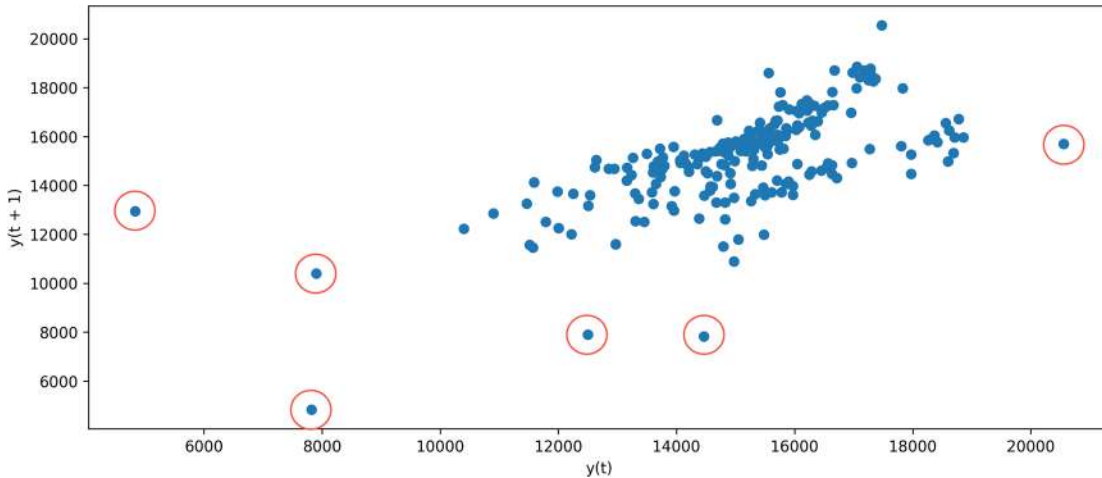


Figure 7.8: A lag plot of average daily taxi passengers in NYC

In Figure 7.8, the circled data points highlight interesting points that can be potential outliers—observations that deviate significantly from the expected relationship. Some seem more extreme than others. Additionally, you can observe a linear relationship between the passenger counts and the lagged version (prior day), indicating the existence of an **autocorrelation**. Recall from basic statistics that correlation shows the relationship between two independent variables, so you can think of autocorrelation as a correlation of a variable at a time (t) and its prior version (past self) at a time ($t-1$). More on this in Chapter 8 and Chapter 9.

The labels for the x axis and the y axis in Figure 7.8 can be a bit confusing, with the y axis being labeled as $y(t+1)$. Essentially, it is saying the same thing we described earlier: the x axis represents prior values (the predictor) to its future self at $t+1$, which is what the y axis represents. To make it clearer, you can recreate the exact visualization produced by `lag_plot` using Seaborn manually, as shown in the following code:

```
y = tx[1:].values.reshape(-1) # Values from t+1 to the end
x = tx[:-1].values.reshape(-1) # Values from the start to t-1
sns.scatterplot(x=x, y=y);
```

This should produce a similar plot to that in Figure 7.8.

Notice in the code that the y values start from $t+1$ (we skipped the value at index 0) up to the last observation, and the x values start from index 0 up to index -1 (we skip the last observation). This makes the values on the y axis ahead by one period. This manual construction using Seaborn demonstrates the underlying principle of the lag plot and provides more control over the visualization.

In the next recipe, we will dive further into the **IQR** and **Tukey fences**, which we briefly discussed when talking about box plots.

See also

You can learn more about the plots we used and the different options available from the Seaborn documentation. To learn more about the following, visit the associated URLs:

- For box plots (boxplot), you can visit <https://seaborn.pydata.org/generated/seaborn.boxplot.html>
- For boxen plots (boxenplot), you can visit <https://seaborn.pydata.org/generated/seaborn.boxenplot.html>
- For violin plots (violinplot), you can visit <https://seaborn.pydata.org/generated/seaborn.violinplot.html#seaborn.violinplot>
- For histograms (histplot), you can visit <https://seaborn.pydata.org/generated/seaborn.histplot.html>
- For distribution plots (displot), you can visit <https://seaborn.pydata.org/generated/seaborn.displot.html#seaborn.displot>

Detecting outliers using the Tukey method

The Tukey method is considered a non-parametric method since it doesn't assume any particular distribution of your data. This makes it particularly useful for time-series analysis where data might not follow a normal distribution. The method simply identifies points that are unusually distant from a central portion of the data, regardless of the underlying distribution.

This recipe builds on the previous recipe, *Detecting outliers using visualizations*. In *Figure 7.4*, the box plot of the downsampled data displayed the quartiles with whiskers extending to the upper and lower fences, calculated using the Tukey method.

The box plot in *Figure 7.4* visually represents the key components used in the Tukey method. The blue box spans from Q1 (25th percentile) to Q3 (75th percentile), with the median (50th percentile) shown as a vertical line inside the box. The IQR is the width of this box, and the lower and upper fences extend $1.5 \times \text{IQR}$ below Q1 and above Q3, respectively. Any points beyond these fences are identified as potential outliers.

Visualizations are great to give you a high-level perspective on the data you are dealing with, such as the overall distribution and potential outliers, but often subjective. Ultimately, you want to identify these outliers programmatically so you can flag these data points for further investigation and analysis. This recipe will teach you how to calculate the IQR and define points that fall outside the lower and upper Tukey fences.

How to do it...

Most statistical methods for outlier detection involve spotting extreme values beyond a certain threshold—whether that’s the mean, standard deviation, a percentile, or another metric. We’ll start by exploring how to obtain basic descriptive statistics, focusing on quantiles, which are key to the Tukey method.

Before we begin, let’s quickly recap our dataset. We’re working with the NYC taxi dataset from the NAB, which captures taxi passenger counts at 30-minute intervals. In the first recipe, we downsampled this data to a daily frequency using the mean, creating the `tx` DataFrame we’ll use here. This daily averaging helps make outliers more apparent, as extreme values in the 30-minute intervals influence the daily mean.

1. Both DataFrames and Series have the `describe` method, which outputs summary statistics that include **quartiles**: the first quartile (Q1, 25th percentile), the second quartile (Q2, 50th percentile or median), and the third quartile (Q3, 75th percentile). You can customize the percentiles by providing a list of values to the `percentiles` parameter. The `percentiles` parameter accepts values between 0 and 1, where 0 represents the minimum and 1 represents the maximum. The following code shows how you can get values for additional percentiles:

```
tx = nyc_taxi.resample('D').mean()
percentiles = [0, 0.05, .10, .25, .5, .75, .90, .95, 1]
tx.describe(percentiles= percentiles)
```

This should produce the following DataFrame:

	value
count	215.000000
mean	15137.569380
std	1937.391020
min	4834.541667
Minimum	0% 4834.541667
	5% 11998.181250
	10% 13043.854167
1st Quartile or Lower Quartile	25% 14205.197917
2nd Quartile or Median	50% 15299.937500
3rd Quartile or Upper Quartile	75% 16209.427083
	90% 17279.300000
	95% 18321.616667
Maximum	100% 20553.500000
max	20553.500000

Figure 7.9: Descriptive statistics with custom percentiles for the daily taxi passenger data



Quantiles versus quartiles versus percentiles

These terms can be a bit confusing, but essentially, both percentiles and quartiles are types of quantiles. Sometimes you will see people use *percentiles* more loosely and interchangeably with *quantiles*.

Quantiles are values that divide your data distribution into equal parts. **Quartiles** divide the distribution into four segments (hence the name), marked as Q1 (25th percentile), Q2 (50th percentile or median), and Q3 (75th percentile). **Percentiles** divide the distribution into 100 segments. Percentiles can take any range from 0 to 100, but pandas uses from 0 to 1, while NumPy uses from 0 to 100.

The names indicate the type of partitioning (number of quantiles) applied to the distribution. For example, with 4 quantiles, we call it quartiles; with 2 quantiles, we call it the median; with 10 quantiles, we call it deciles; and with 100 quantiles, we call it percentiles.

2. The NumPy library also offers the percentile and quantile functions, which return the value(s) for the specified percentiles/quantiles. Note the use of integers (0–100) for percentile and floats (0–1) for quantile. The following code explains how this can be used:

```
percentiles = [0, 5, 10, 25, 50, 75, 90, 95, 100]
np.percentile(tx, percentiles)
>>
array([ 4834.54166667, 11998.18125    , 13043.85416667, 14205.19791667,
        15299.9375    , 16209.42708333, 17279.3        , 18321.61666667,
        20553.5        ])

quantiles = [0, 0.05, 0.10, 0.25, 0.50, 0.75, 0.90, 0.95, 1]
np.quantile(tx, quantiles)
>>
array([ 4834.54166667, 11998.18125    , 13043.85416667, 14205.19791667,
        15299.9375    , 16209.42708333, 17279.3        , 18321.61666667,
        20553.5        ])
```

3. Notice that most extreme values, or potential outliers, fall below the lower fence, calculated as $Q1 - (1.5 \times IQR)$, or above the upper fence, calculated as $Q3 + (1.5 \times IQR)$. The IQR is calculated as the difference between Q3 and Q1 ($IQR = Q3 - Q1$), which determines the width of the box in the box plot. These upper and lower fences are known as **Tukey's fences**, and more specifically, they are referred to as inner boundaries. The outer boundaries also have lower, $Q1 - (3.0 \times IQR)$, and upper, $Q3 + (3.0 \times IQR)$, fences. We will focus on the inner boundaries and describe anything outside of those as potential outliers.

You will create a function, `iqr_outliers`, that calculates the IQR, upper (inner) fence, and lower (inner) fence, and then filters the data to return the outliers. These outliers are any data points that are below the lower fence or above the upper fence:

```
def iqr_outliers(data, k):
    """
    Detect outliers using Tukey's method with
    customizable fence multiplier.
    """
    q1, q3 = np.percentile(data, [25, 75])
    IQR = q3 - q1
    lower_fence = q1 - (k * IQR)
    upper_fence = q3 + (k * IQR)
    return data[(data.value > upper_fence) | (data.value < lower_fence)]
```

4. Test the function by passing the `tx` DataFrame:

```
outliers = iqr_outliers(tx, 1.5)
print(outliers)
>>
           value
timestamp
2014-11-01  20553.500000
2014-11-27  10899.666667
2014-12-25   7902.125000
2014-12-26  10397.958333
2015-01-26   7818.979167
2015-01-27   4834.541667
```

These dates (points) are the same ones identified in *Figure 7.4* as outliers based on Tukey's fences.

5. Use the `plot_outliers` function defined earlier in the *Technical requirements* section:

```
plot_outliers(outliers, tx,
              "Outliers using IQR with Tukey's Fences",
              labels=True)
```

This should produce a plot similar to that in *Figure 7.2*, except the X markers are based on the outliers identified using the Tukey method:

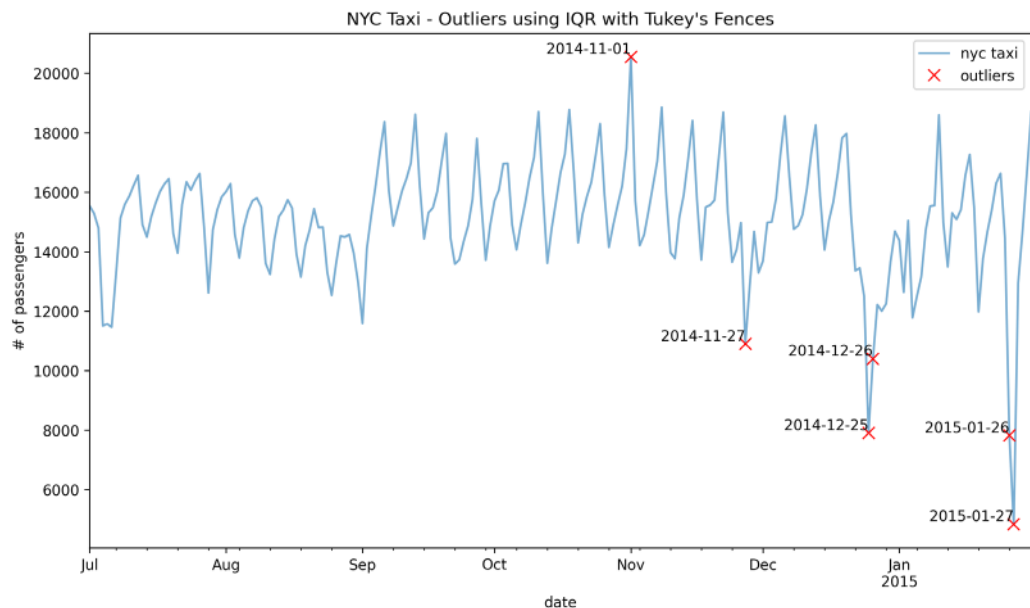


Figure 7.10: Daily average taxi passengers and outliers identified using the Tukey method

Compare Figure 7.10 and Figure 7.2 and you will see that this simple method did a great job at identifying four of the five known outliers. In addition, Tukey's method identified two additional outliers on 2014-12-26 and 2015-01-26.

How it works...

Using the IQR and Tukey's fences is a straightforward, non-parametric statistical method. Most box plot implementations, such as those in Seaborn or Matplotlib, use the $1.5 \times \text{IQR}$ rule to define the upper and lower fences. This method doesn't assume a specific distribution, making it versatile for various datasets. This distribution-agnostic property is particularly valuable for time-series data such as our taxi passenger counts, which may exhibit seasonality, trends, and other patterns that create naturally skewed distributions.

By adjusting the k parameter in the `iqr_outliers` function, you can control the sensitivity of the outlier detection, which you will explore in the next section.

There's more...

The use of $1.5 \times \text{IQR}$ is common when it comes to defining outliers; the choice is still somewhat arbitrary, even though it has become a statistical convention. You can change the multiplier value for experimentation. For example, in Seaborn, you can change the default 1.5 value by updating the `whis` parameter in the `boxplot` function. The choice of 1.5 makes the most sense when the data follows a Gaussian distribution (normal), but this is not always the case. Generally, the larger the value, the fewer outliers you will capture as you expand your boundaries (fences). Similarly, the smaller the value, the more non-outliers will be defined as outliers, as you are shrinking the boundaries (fences).

You can experiment with different values of the k multiplier in the `iqr_outliers` function to adjust the sensitivity of your outlier detection:

```
for k in [1.3, 1.5, 2.0, 2.5, 3.0]:
    print(f'using k={k}')
    print(iqr_outliers(tx, k))
    print('-'*25)
```

```
>>
```

```
using k=1.3
```

	value
timestamp	
2014-07-04	11511.770833
2014-07-05	11572.291667
2014-07-06	11464.270833
2014-09-01	11589.875000
2014-11-01	20553.500000
2014-11-08	18857.333333
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

```
-----
using k=1.5
```

	value
timestamp	
2014-11-01	20553.500000
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

```
-----
using k=2.0
```

	value
timestamp	
2014-11-01	20553.500000
2014-12-25	7902.125000
2015-01-26	7818.979167
2015-01-27	4834.541667

```
-----
using k=2.5
```

```

                                value
timestamp
2014-12-25  7902.125000
2015-01-26  7818.979167
2015-01-27  4834.541667
-----
using k=3.0
                                value
timestamp
2014-12-25  7902.125000
2015-01-26  7818.979167
2015-01-27  4834.541667
-----

```

As shown in the results, decreasing the k value from 1.5 to 1.3 increases the number of detected outliers from 6 to 11, making the detector more sensitive. Conversely, increasing k to 2.0 or higher makes the detector more conservative, identifying only the most extreme values. For time-series forecasting applications, you might want to use a more conservative approach (higher k) when you expect legitimate seasonal spikes, and a more sensitive approach (lower k) when dealing with data where any deviation could be significant. The best value will depend on your data and how sensitive you need the outlier detection to be.

See also

- To learn more about Tukey's fences for outlier detection, you can refer to this Wikipedia page: https://en.wikipedia.org/wiki/Outlier#Tukey's_fences.

Detecting outliers using a z-score

The **z-score** is a common transformation for standardizing data, making it easier to compare different datasets—it is easier to compare two data points from two different datasets relative to their distributions. This standardization centers the data around a 0 mean, with units representing standard deviations away from the mean. For example, in our dataset, the unit is measured in daily taxi passengers (in thousands) after downsampling from the original 30-minute interval data.

After applying the z-score transformation, you are no longer dealing with the number of passengers; rather, the units represent standard deviations, indicating how far an observation is from the mean. Here is the formula for the z-score:

$$z = \frac{x - \mu}{\sigma}$$

Here, x is a data point (an observation), μ is the mean of the dataset, and σ is the standard deviation.

The z-score is a *lossless transformation*, meaning you will not lose information such as its distribution (shape of the data) or the relationship between the observations. One key property of the z-score is that it's an *invertible transformation*—you can recover the original data if you know the mean and standard deviation. This means you preserve all the original information while standardizing the scale. All that is changing is the units of measurement as they are being scaled (standardized).

Once the data is transformed using the z-score, you can pick a **threshold**. Any data point above or below that threshold (in standard deviation) is considered an outlier. For example, your threshold can be ± 3 standard deviations away from the mean. Any points lower than -3 or higher than +3 standard deviations can be considered an outlier. In other words, the further a point is from the mean, the higher the probability of it being an outlier.

The z-score has one major shortcoming due to it being a parametric statistical method based on assumptions. It assumes a Gaussian (normal) distribution. If the data is not normally distributed, a modified z-score may be more appropriate, which is discussed in the next recipe, *Detecting outliers using a modified z-score*.

How to do it...

You will start by creating the `zscore` function, which takes in a dataset and a threshold value. The function will return both the standardized data and the identified outliers. These outliers are any points above the positive threshold or below the negative threshold:

1. Create the `zscore()` function to standardize the data and filter out the extreme values based on a threshold. Recall that the threshold is based on the standard deviation:

```
def zscore(df, threshold=3):
    """
    Detect outliers using z-score method with customizable threshold.
    """
    data = df.copy()
    data['zscore'] = (data - data.mean())/data.std()
    outliers = data[
        (data['zscore'] <= -threshold) | (data['zscore'] >= threshold)]

    return outliers, data
```

The choice of threshold depends on your statistical confidence requirements and data characteristics. For normally distributed data, the following apply:

- A threshold of 1.96 corresponds to approximately 95% confidence (only ~5% of data should fall outside if normally distributed)
- A threshold of 2.58 corresponds to approximately 99% confidence
- A threshold of 3 is often used as a conservative rule of thumb

- For our taxi data, we'll use a threshold of 2.5, which is a moderate value that balances sensitivity:

```
tx = nyc_taxi.resample('D').mean()
threshold = 2.5
outliers, transformed = zscore(tx, threshold)
```

- To see the effect of the z-score transformation, you can plot a histogram. The transformed DataFrame contains two columns, the original data labeled `value` and the standardized data labeled `zscore`:

```
transformed.hist()
```

This should produce two histograms for the two columns:

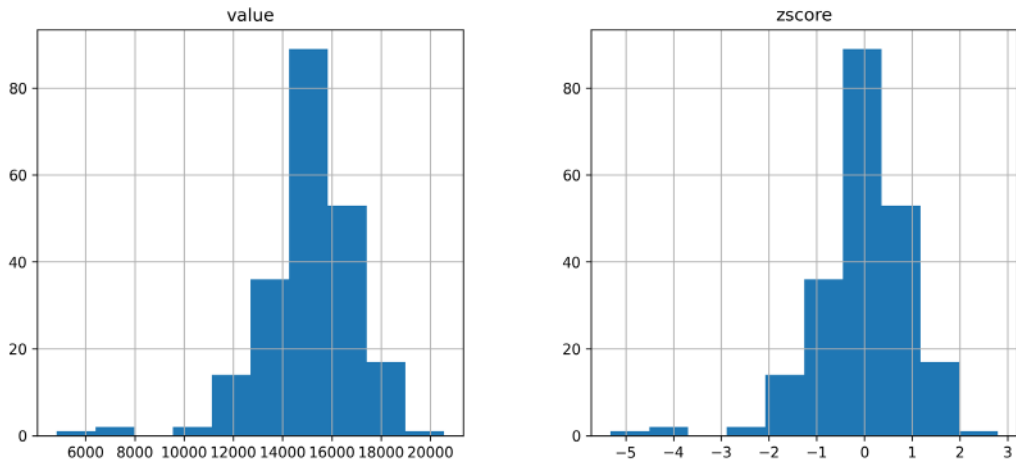


Figure 7.11: Histogram to compare the distribution of the original and standardized data

In Figure 7.11, the left histogram shows the original passenger counts, while the right histogram shows the standardized z-scores. Notice how both distributions maintain the same shape with a slight right skew, but the z-score distribution is centered at 0 with most values falling between -2 and 2. This standardization makes it easier to apply consistent thresholds across different datasets.

The z-score is called a *lossless transformation* because it does not change the shape of the data; the only difference is in the scale (units).

- You ran the `zscore` function using a threshold of 2.5, meaning any data point that is 2.5 standard deviations away from the mean in either direction. For example, any data point that is above the +2.5 standard deviations or below the -2.5 standard deviations will be considered an outlier. Print out the results captured in the `outliers` object:

```
print(outliers['value'])
>>
timestamp
```

```

2014-11-01    20553.500000
2014-12-25     7902.125000
2015-01-26     7818.979167
2015-01-27     4834.541667
Name: value, dtype: float64

```

This simple method managed to capture three out of the five known outliers.

5. Use the `plot_outliers` function defined earlier in the *Technical requirements* section:

```

plot_outliers(outliers[['value']], tx,
              'Outliers using Z-score', labels=True)

```

This should produce a plot similar to that in *Figure 7.2*, except that the X markers are based on the outliers identified using the z-score method.

You will need to play around to determine the best threshold value. The larger the threshold, the fewer outliers you will capture, and the smaller the threshold, the more non-outliers will be labeled as outliers.

6. Finally, let's use the `plot_zscore` function which takes the standardized data to plot the data with the threshold lines. This way, you can visually see how the threshold is isolating extreme values. Run the function using a threshold of 2.5:

```

plot_zscore(transformed['zscore'], d=2.5)

```

This should produce a scatter plot with two horizontal lines:

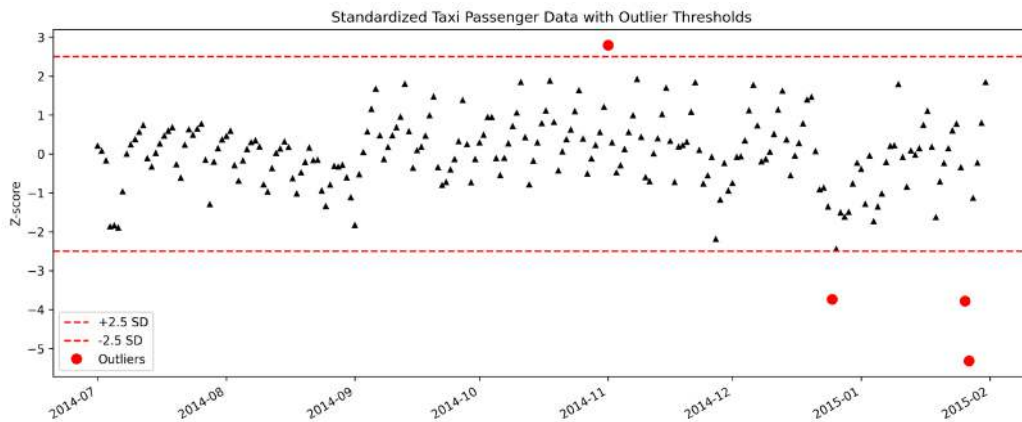


Figure 7.12: Plot of the standardized data and outliers based on the threshold lines

The four circles represent the outliers that were returned by the `zscore` function. Run the function using different threshold values to gain a deeper understanding of this simple technique.

In *Figure 7.12*, the horizontal red dashed lines represent our threshold of ± 2.5 standard deviations. Any points beyond these lines are identified as outliers. This visualization helps us understand why certain points were flagged—they exceed our defined threshold in terms of standard deviations from the mean.

Comparing these results with the Tukey method from the previous recipe, we can see that the z-score method detected fewer outliers (4 versus 6). This is because the z-score method assumes a normal distribution, while the Tukey method is non-parametric. The z-score missed 2014-11-27 (Thanksgiving) and 2014-12-26 (the day after Christmas), which were identified by the Tukey method, highlighting the importance of understanding your data's distribution before selecting an outlier detection approach.

How it works...

The z-score method is straightforward and interpretable. The z-scores represent the number of standard deviations that an observation is away from the mean, which is the center of the distribution. Since we are subtracting the mean from all observations, we are essentially mean-centering the data (centering the data around 0). Dividing by the standard deviation standardizes the data, making it easier to compare different datasets.

Figure 7.12 effectively illustrates how this method works. Once the data is standardized, it becomes easy to apply the standard deviation threshold. If the data were not standardized, it would be more challenging to determine an appropriate threshold based on raw daily passenger counts.

There's more...

In outlier detection, the z-score is a parametric method and assumes the data comes from a Gaussian (normal) distribution. Before applying this method, it's valuable to verify whether your data actually follows a normal distribution.

The `statsmodels` library provides several tests for normality. We'll use the **Kolmogorov-Smirnov** test with Lilliefors correction (`kstest_normal`), which is specifically designed for testing normality when the mean and variance are estimated from the data. This test measures how well your data's distribution matches a normal distribution.

The test works with a null hypothesis that the data follows a normal distribution. It returns a test statistic and a p-value; if the p-value is less than 0.05, you can reject the null hypothesis (concluding the data is not normally distributed). Otherwise, you would fail to reject the null hypothesis (suggesting the data is normally distributed).

To make this easier to run, let's create a `test_normal` function that leverages the `kstest_normal` function from `statsmodels`:

```
from statsmodels.stats.diagnostic import kstest_normal

def test_normal(df):
    """
    Test if data follows a normal distribution using Kolmogorov-Smirnov test.
    """
    t_test, p_value = kstest_normal(df)
    if p_value < 0.05:
        print("Reject null hypothesis. Data is not normal")
    else:
        print("Fail to reject null hypothesis. Data is normal")
```

Run the test using the `test_normal` function:

```
test_normal(tx)
>>
Reject null hypothesis. Data is not normal
```

Since the z-score method assumes normality, this test helps us determine whether it's an appropriate technique for our data. As we can see, our taxi data fails the normality test, which explains why the z-score method missed some of the outliers that the Tukey method identified.

In *Chapter 8*, you will learn about additional normality tests in the *Applying power transformations* recipe. But do be cautious; normality tests such as this one can fail in the presence of outliers. If your data fails a normality test, then use some of the plotting methods discussed in the *Detecting outliers using visualizations* recipe to examine any outliers that may be causing the test to fail.

In the following recipe, you will explore a very similar method to the z-score that is more robust to outliers and is more suitable for non-normal data.

See also

- To read more about z-scores and standardization, you can refer to this Wikipedia page: https://en.wikipedia.org/wiki/Standard_score.

Detecting outliers using a modified z-score

In the previous recipe, you explored the z-score method, which is a simple and intuitive method for identifying outliers. However, it comes with a major drawback: it assumes the data follows a normal distribution. Real-world data often deviates from normality, making the regular z-score less effective.

Fortunately, to address this, there is a modified version of the z-score designed to work with non-normal data. The main difference between the regular z-score and the **modified z-score** is that we replace the mean and standard deviation with the **median** and **median absolute deviation (MAD)**:

$$\text{Modified } Z = \frac{0.6745 (x_i - \tilde{x})}{MAD}$$

Here, $\tilde{x}$ (*tilde x*) is the median of the dataset, and **MAD** is the **median absolute deviation** of the dataset:

$$MAD = \text{median}(\text{abs}(x_i - \tilde{x}))$$

The constant 0.6745 is a normalization factor that corresponds to the 75th percentile (Q3) in a Gaussian distribution. It helps to approximate the standard deviation, allowing the modified z-score to be interpreted in standard deviation units, similar to how we interpret the regular z-score.

You can obtain this normalization factor using SciPy's **percent point function (PPF)**, also known as the **inverse** of the CDF. Simply give the PPF function a percentile, for example, 75%, and it will return the quantile corresponding to the lower tail probability:

```
import scipy.stats as stats
stats.norm.ppf(0.75)
>>
0.6744897501960817
```

This is the normalization factor used in the formula. The **modified z-score** is sometimes referred to as the **robust z-score**.

The modified z-score is particularly well suited for our NYC taxi dataset, where extreme events such as the North American blizzard and the NYC Marathon create outliers that don't fit a normal distribution.

How to do it...

Overall, the approach works similarly to the standard z-score method. You will start by creating the `modified_zscore` function, which takes in a dataset and a threshold value, and returns both the standardized data and the identified outliers. These outliers are any points above the positive threshold or below the negative threshold:

1. Create the `modified_zscore()` function to standardize the data and filter out extreme values based on a threshold. The threshold represents the number of standard deviations (in modified z-score units) away from the median that a value must exceed to be considered an outlier:

```
import scipy.stats as stats
def modified_zscore(df, threshold=3):
    """
    Detect outliers using a modified z-score method with
    customizable threshold.
    """
    data = df.copy()

    # Calculate median and MAD
    median = data['value'].median()
```

```

MAD = np.median(np.abs(data['value'] - median))

# Calculate modified z-score
s = stats.norm.ppf(0.75)
data['m_zscore'] = s * (data['value'] - median) / MAD
# Identify outliers
outliers = data[
    (data['m_zscore'] > threshold) |
    (data['m_zscore'] < -threshold)
]

return outliers, data

```

- Now, use the `modified_zscore` function with the downsampled data with a threshold of 3, and store the returned objects:

```

tx = nyc_taxi.resample('D').mean()
threshold = 3
outliers, transformed = modified_zscore(tx, threshold)

```

- To see the effect of the modified z-score transformation, let's plot a histogram. The transformed DataFrame contains two columns—the original data labeled `value` and the standardized data labeled `m_zscore`:

```
transformed.hist()
```

This produces two histograms for the two columns, as shown in *Figure 7.13*:

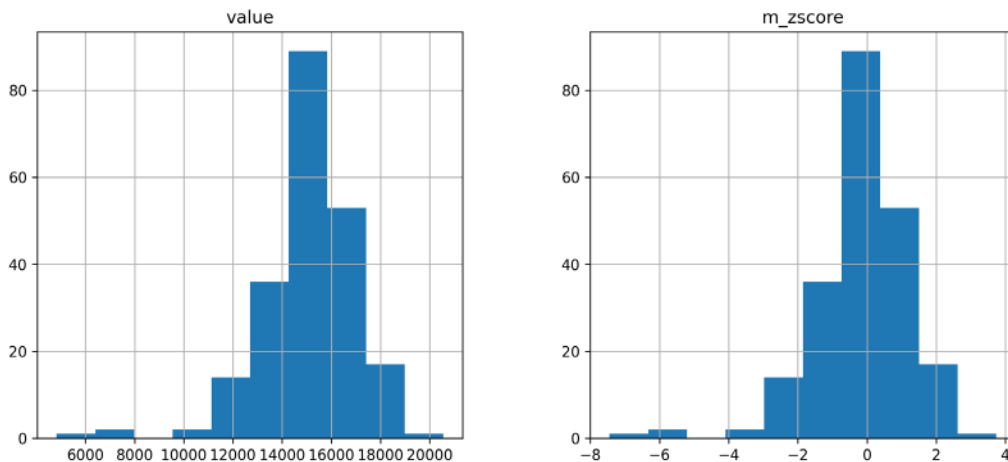


Figure 7.13: Histograms to compare the distribution of the original and modified z-score standardized data

Notice how the original data is right-skewed, reflecting the non-normal nature of taxi passenger counts due to seasonal patterns and extreme events. The modified z-score transformation doesn't change the shape of the distribution—it simply scales the data.

Compare the results from *Figure 7.13* with the histograms from the standard z-score in *Figure 7.11*. Both approaches—z-score and modified z-score—retain the shape of the data distribution. The difference lies in the scaling factor and central tendency measure used (mean and standard deviation versus median and MAD).

4. You ran the `modified_zscore` function using a threshold of 3, meaning any data point three standard deviations away from the median in either direction (above +3 standard deviations or below -3 standard deviations) will be considered an outlier. Print out the results captured in the `outliers` object:

```
print(outliers['value'])
>>
timestamp
2014-11-01    20553.500000
2014-11-27    10899.666667
2014-12-25     7902.125000
2014-12-26    10397.958333
2015-01-26     7818.979167
2015-01-27     4834.541667
Name: value, dtype: float64
```

Interestingly, the modified z-score identified six outliers, including four out of the five known outliers in our ground-truth list (`nyc_dates`) and also identified December 26, 2014 (the day after Christmas), and January 26, 2015 (the first day of the North American blizzard). Compare this to the standard z-score used in the previous recipe, in which it identified only four outliers, including three out of the five known outliers.

5. Use the `plot_outliers` function defined earlier in the *Technical requirements* section:

```
plot_outliers(outliers[['value']], tx,
              'Outliers using Modified Z-score', labels=True)
```

This should produce a plot similar to that in *Figure 7.2*, except the X markers are based on the outliers identified using the modified z-score method.

You will need to experiment to determine the optimal threshold value. A larger threshold will identify fewer outliers, while a smaller threshold may incorrectly label non-outliers as outliers.

6. Use the `plot_zscore` function that was defined in the previous recipe, *Detecting outliers using a z-score* with a threshold of 3:

```
plot_zscore(transformed['m_zscore'], d=3)
```

This should produce a scatter plot with two horizontal lines, as shown in *Figure 7.14*:

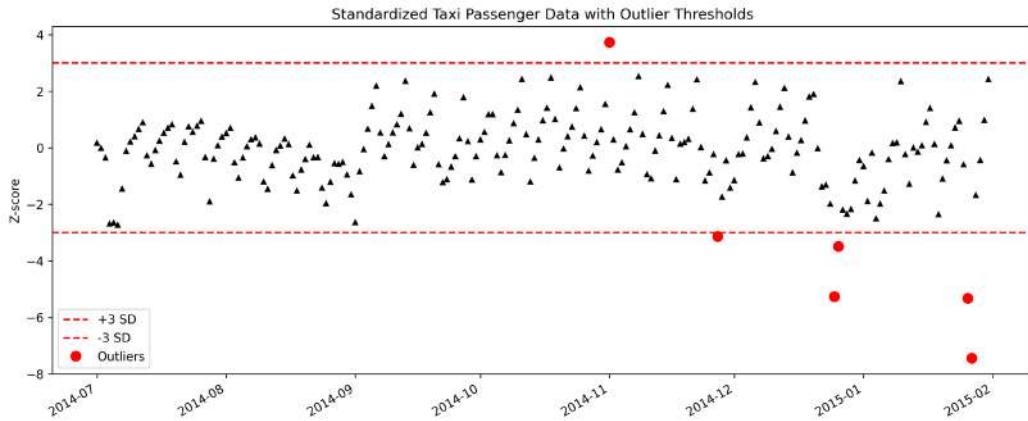


Figure 7.14: Plot of the standardized data and outliers based on the threshold lines

The six circles represent the outliers that were returned by the `modified_score` function. Try running the function using different threshold values to gain more deeper intuition into this technique.

Notice in *Figure 7.14* how one data point is right at the threshold line. Would you consider this an outlier? Generally, when it comes to outlier detection, you should always apply due diligence to inspect the results rather than relying solely on automated methods.

How it works...

The modified z-score method improves upon the standard z-score by using robust statistics—the **median** and **MAD**—which are less influenced by extreme values. This makes it especially suitable for the following:

- **Non-normally distributed data:** Unlike the standard z-score, which assumes a normal distribution, the modified z-score does not rely on this assumption. This makes it suitable for data that may be skewed or have heavy tails.
- **Data with existing outliers:** The mean and standard deviation, used in the standard z-score, can be heavily influenced by outliers. By using the median and MAD, the modified z-score mitigates this issue, providing a more reliable measure of central tendency and dispersion.
- **Need for a robust outlier detection:** In practical applications where data quality may vary, the modified z-score offers a more resilient approach to identifying outliers, ensuring that the detection process is not unduly affected by anomalous data points.

When comparing the results from the modified z-score with the standard z-score method, we observe the following:

- The modified z-score detected six outliers, while the standard z-score detected only four
- The modified z-score successfully identified more of the known anomalous dates in our dataset
- The robust approach was less affected by the skewed distribution of our taxi passenger data

This demonstrates why the modified z-score is often preferred for real-world time-series data, which frequently exhibits non-normal distributions and seasonal patterns.

There's more...

In the previous recipe, *Detecting outliers using a z-score*, we utilized the `kstest_normal` function from `statsmodels` to test for normality. Another helpful tool for assessing normality and identifying potential outliers is the **QQ plot**. A QQ plot compares the quantiles of your data against the quantiles of a theoretical normal distribution, providing a visual assessment of how well your data aligns with normality.

You can create a QQ plot using SciPy or `statsmodels`. Both libraries will produce the same plot, but they offer slightly different interfaces. The following code will show how you can plot using either method.

This shows how you can plot using **SciPy**:

```
import scipy
import matplotlib.pyplot as plt

scipy.stats.probplot(tx['value'], plot=plt);
```

This shows how you can plot using **statsmodels**:

```
from statsmodels.graphics.gofplots import qqplot
qqplot(tx['value'], line='s');
```

Both methods will produce a plot similar to *Figure 7.15*, which compares the taxi passenger data against a hypothetical normal distribution.

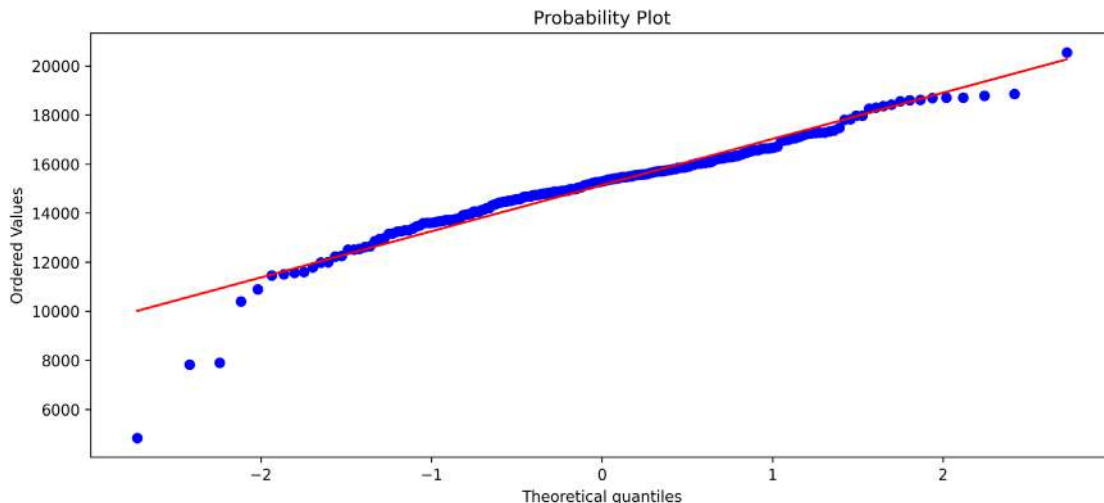


Figure 7.15: QQ plot comparing the taxi passenger data against a hypothetical normal distribution

The solid line in the QQ plot represents a reference line for what normally distributed data would look like. If the data you are comparing is normally distributed, all the points will lie on that straight line. In *Figure 7.15*, we can observe that the distribution of the downsampled taxi passenger data is almost normal but not perfect. Specifically, there are noticeable deviations in both tails of the distribution, with more pronounced deviation in the lower tail (around theoretical quantiles below -2). This aligns with what we observed in *Figure 7.13* and *Figure 7.11*, where we can see small frequencies of very low passenger counts in the left tail of the distribution, showing that many of the outliers are concentrated in the lower tail (less than -2 standard deviation).

See also

- To learn more about MAD, you can refer to the Wikipedia page here: https://en.wikipedia.org/wiki/Median_absolute_deviation.

Detecting outliers using the Hampel filter

In this recipe, we'll explore the Hampel filter, a robust non-parametric method for detecting outliers in time-series data. The z-score and Tukey methods, which we've discussed in previous recipes, work well for static distributions as they treat the entire dataset as a single population. The **Hampel filter**, however, can be thought of as a "*rolling window modified z-score*" that examines local behavior through a sliding window approach, making it effective for time series with changing baselines, seasonal patterns, or trends.

The Hampel filter, like the modified z-score, uses the median and MAD to identify anomalies, making it resistant to extreme values and ideal for non-normal distributions. However, instead of calculating these statistics globally across the entire dataset, the Hampel filter calculates them within a **sliding window**.

A point is considered an outlier if its deviation from the local median exceeds `n_sigma` times the scaled MAD ($k \times MAD$). The scaling factor ($k = 1.4826$ by default) makes the MAD consistent with the standard deviation for normally distributed data.

This local focus makes the Hampel filter especially valuable for real-world time-series data such as our NYC taxi passenger counts, where normal passenger levels in July might look very different from those in December due to seasonality. Additionally, the filter is less affected by trends that would distort global statistics.

We'll implement the Hampel filter using `sktime`'s implementation and visualize the results using the `plot_outliers` plotting function from the *Technical requirements* section.

Getting ready

In this recipe, you will work with the `sktime` library, described as "*a unified framework for machine learning with time series*." Behind the scenes, `sktime` is a wrapper to other popular machine learning and time-series libraries, including `scikit-learn`.



Creating a new Python environment when installing new libraries such as `sktime` is always a good idea. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup* recipe in *Chapter 0 on GitHub*. The chapter covers two methods: using `conda` and `venv`.

To install `sktime` using `pip`, you can run the following:

```
pip install "sktime[all_extras]"
```

This command installs all optional dependencies, which can be resource-intensive. If you encounter issues, consider installing only the necessary components (e.g., `sktime[forecasting]`).

How to do it...

We'll start with the Hampel filter, which extends the modified z-score concept into a rolling window implementation. Unlike the previous methods, the Hampel filter considers the local behavior of the time series rather than treating the entire dataset as a single distribution. This makes it especially effective for data with changing baselines, seasonal patterns, or trends—all common characteristics of real-world time-series data such as our NYC taxi passenger counts:

1. Start by importing the `HampelFilter` class from `sktime`:

```
from sktime.transformations.series.outlier_detection import HampelFilter
```

2. To make `HampelFilter` easy to use and consistent with the functions in previous recipes, we'll create a `hampel_outlier_detection` function that accepts time-series data, the window length, and the threshold as inputs. The function will return both the outliers and the transformed data for visualization:

```
def hampel_outlier_detection(df, window_length=10, n_sigma=3):
    """
    Detect outliers using sktime's HampelFilter implementation.
    """
    # Create a copy of the input data
    data = df.copy()

    # Initialize the HampelFilter
    hampel = HampelFilter(
        window_length=window_length,
        n_sigma=n_sigma,
        k=1.4826, # Default scale factor for Gaussian distribution
        return_bool=False # Return NaN for outliers
    )

    # Apply the filter -
    # this will return a series with NaN values at outlier positions
```



```
data['filtered'] = Hampel.fit_transform(data['value'])

# Identify outliers (where transformed values are NaN)
outliers = data[data['filtered'].isna()]

return outliers, data
```

3. Apply the `hampel_outlier_detection` function to our downsampled daily data. You will use a window length of 21 days to capture approximately monthly patterns, and a threshold of `n_sigma=2.5`:

```
tx = nyc_taxi.resample('D').mean()
window_length = 21 # Adjust based on your data's seasonality
n_sigma = 2.5 # Slightly more lenient than the typical 3 sigma

outliers, transformed = hampel_outlier_detection(
    tx, window_length, n_sigma)
```

4. The outliers DataFrame contains the detected outliers:

```
print(outliers['value'])
>>
timestamp
2014-07-04    11511.770833
2014-07-05    11572.291667
2014-07-06    11464.270833
2014-07-28    12615.854167
2014-11-01    20553.500000
2014-12-06    18561.625000
2014-12-25     7902.125000
2015-01-10    18597.166667
2015-01-26     7818.979167
2015-01-27     4834.541667
Name: value, dtype: float64
```

The Hampel filter identified 10 potential outliers. Importantly, it successfully detected three out of the five known anomalous dates (2014-11-01, 2014-12-25, and 2015-01-27). Additionally, it flagged 2015-01-26, which is the day before the blizzard, showing the start of the impact.

The dates in early July (2014-07-04, 2014-07-05, and 2014-07-06) likely correspond to Independence Day weekend, which would naturally affect taxi usage patterns. These extra detections could be subtle anomalies—or false positives. That’s where your domain knowledge comes in to separate the signal from the noise.

5. Visualize the results using the `plot_outliers` function:

```
plot_outliers(outliers[['value']], tx,  
              'Outliers using Hampel Filter')
```

This should produce a plot similar to that in *Figure 7.2*, except the X markers are based on the outliers identified using the Hampel filter method:

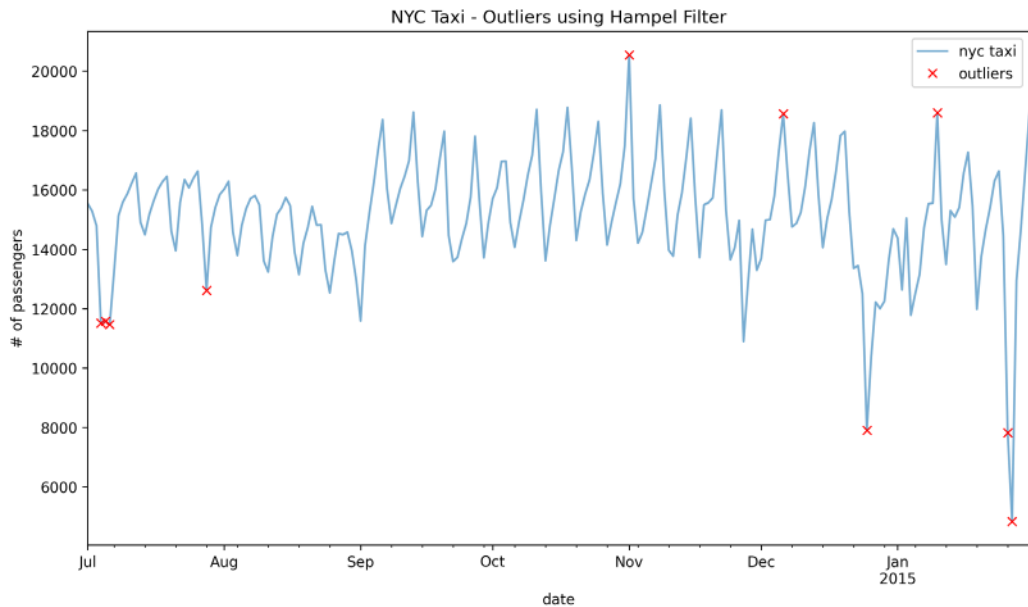


Figure 7.16: Daily average taxi passengers and outliers identified using the Hampel filter method

You can further explore and tune the parameters (`window_length` and `n_sigma`). The number of outliers (10) suggests the current settings might be too lenient. Consider a larger window (e.g., 30) to smooth out noise and incorporate more context or a higher threshold (e.g., 3.0) for more conservative outlier detection.

How it works...

The Hampel filter operates as a moving window outlier detector, applying robust statistics to each local segment of your time series. It is great for detecting **local outliers**, which are data points that may appear normal in a global context but are anomalous compared to their immediate temporal neighbors.

Here's a breakdown of the process for each data point:

1. **Windowing:** A window of observations, centered on the current data point, is selected. In our implementation, we used a `window_length` value of 21 days to capture approximately monthly patterns in our daily taxi data.
2. **Median calculation:** The median of the values within this window is calculated. The median is a robust measure of central tendency, less affected by extreme values than the mean.

3. **MAD calculation:** The MAD is computed for the window. This measures the typical distance of data points from the median and serves as a robust alternative to standard deviation.
4. **Scaling the MAD:** The MAD is scaled by a factor of 1.4826. This constant makes the MAD a consistent estimator of the standard deviation if the data were normally distributed. This scaling allows us to interpret `n_sigma` in a way that's similar to how we use it with z-scores. This is comparable to the modified z-score's use of 0.6745 (related to the 75th percentile of a Gaussian distribution), which we discussed in the previous recipe as a way to handle non-normal data.
5. **Thresholding:** Thresholds are established at `n_sigma` times the scaled MAD, above and below the window's median. Points outside these thresholds are candidates for outliers.
6. **Outlier identification:** Any data point falling outside these threshold bands is flagged as an outlier.

The filter successfully detected our known anomalies (2014-11-01 for NYC Marathon preparation, 2014-12-25 for Christmas, and 2015-01-27 for the blizzard) and also identified other potential outliers. These additional detections—such as the Independence Day period—demonstrate how the method can uncover patterns we might not have initially considered. Depending on your domain knowledge, these could either be valuable new insights or false positives requiring further investigation.

The effectiveness of the Hampel filter depends on two key parameters:

- `window_length`: Controls the size of the sliding window. Larger windows (such as our 21-day window) detect outliers against broader patterns, making them less sensitive to short-term fluctuations but better at capturing seasonal anomalies. Smaller windows are more responsive to abrupt and recent changes. For hourly data with daily patterns, a `window_length` value of 24 might be appropriate; for monthly data, a `window_length` value of 12 could capture yearly seasonal patterns.
- `n_sigma`: Determines the outlier **threshold**, indicating how many scaled MADs a point must deviate from to be considered an outlier. Our choice of 2.5 balances **sensitivity** and **specificity**. Lower values (around 2.0) would flag more outliers, increasing sensitivity but potentially introducing false positives, while higher values (3.5 or above) would be more selective, reducing false positives but potentially missing subtle anomalies.

Experiment with different combinations of these parameters to find the optimal balance for your specific time-series data. A good approach is to start with domain knowledge—if you know the approximate periodicity of your data, set `window_length` to capture at least one full cycle—and then adjust `n_sigma` to control sensitivity. For our NYC taxi data, the 21-day window was selected to incorporate approximately monthly patterns, while the `n_sigma` value of 2.5 was chosen as a balanced threshold that captured known anomalies without excessive false positives.

Unlike global methods such as the standard z-score, which can miss local anomalies in trending or seasonal data, the Hampel filter adapts to the changing characteristics of your time series.



Sensitivity versus specificity

In outlier detection, there's always a trade-off between sensitivity and specificity:

- **Sensitivity** refers to the ability to detect true outliers from all actual outliers in the data—higher sensitivity means capturing more of the actual anomalies in your data
- **Specificity** refers to the ability to correctly identify normal points as non-outliers—higher specificity means fewer false alarms, meaning fewer normal points being incorrectly flagged as outliers

Adjusting `n_sigma` allows you to balance these competing goals based on your application's needs. In critical systems where missing an anomaly could be costly, you might prioritize sensitivity. In systems where false alarms are disruptive, you might prioritize specificity.

There's more...

In the `HampelFilter` implementation in `sktime`, outliers are replaced with `NaN` values by default, effectively filtering them out of the data without replacing them with alternative values. You can visualize the filtered data:

```
transformed['filtered'].plot();
```

This will give us the following output:

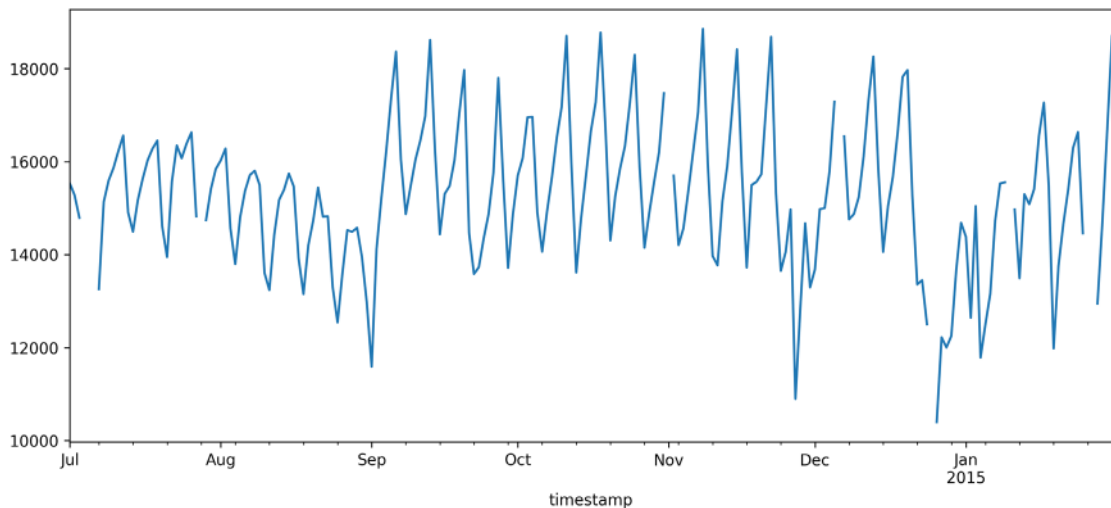


Figure 7.17: Daily average taxi passengers filtered (outliers removed) using the Hampel filter

To correct for the outlier removal, you can use `sktime`'s `Imputer`

using `sktime.transformations.series.impute` import `Imputer`:

```
from sktime.transformations.series.impute import Imputer
# Other options: "mean", "median", "nearest", etc.
imputer = Imputer(method="linear")

# Fill in the missing values (former outliers)
y_corrected = imputer.fit_transform(transformed['filtered'])
y_corrected.plot()
plt.title("Daily Average Taxi Passengers After Linear Imputation");
```

The `Imputer` class offers several imputation methods, which you can explore here: https://www.sktime.net/en/stable/api_reference/auto_generated/sktime.transformations.series.impute.Imputer.html.

You will explore more of `sktime`'s capabilities in *Chapter 11*. For now, you will explore two outlier detection classes in `sktime`: **STRAY** and **Matrix Profile**.

Detecting outliers with STRAY

The **Search and TRace AnomaLY** (STRAY) algorithm is another statistical method for anomaly and outlier detection. It's an extension of the **HDoutliers** algorithm, which is efficient at detecting clusters of outliers in multi-dimensional data. STRAY extends **HDoutliers** by using **extreme value theory** (EVT) for calculating anomalous thresholds. The algorithm is designed for detecting anomalies in data streams that exhibit **concept drift**. STRAY gets its name from its approach: it searches for anomalies by examining the trace (a measure of spread) of the data.

Concept drift

Concept drift refers to the phenomenon where the statistical properties of the target variable change over time in unforeseen ways. In time-series data, this means the underlying patterns, relationships, or distributions evolve gradually or suddenly.

Examples include the following:

- Consumer behavior changing during economic shifts
- Traffic patterns evolving as new infrastructure is built
- Sensor readings drifting as equipment ages

The STRAY implementation involves several key components:

- **Nearest neighbor computation:** The algorithm utilizes `sklearn`'s `NearestNeighbors` to calculate distances between data points. Unlike distribution-based approaches, STRAY leverages a distance-based approach for outlier detection.
- **Threshold determination:** STRAY utilizes EVT to calculate thresholds for anomaly detection, allowing it to identify outliers that deviate significantly from expected behavior.

- **Hyperparameters:** The algorithm accepts several configuration parameters, including the following:
 - **alpha:** Outlier cutoff threshold (default is 0.01)
 - **k:** Number of neighbors considered (default is 10)
 - **knn_algorithm:** The method for computing nearest neighbors (default is 'brute')
 - **p:** Proportion of outlier candidates (default is 0.5)
 - **size_threshold:** Sample size for empirical threshold calculation (default is 50)
 - **outlier_tail:** The direction of the outlier tail (default is 'max')

Here is how we can implement STRAY on our downsampled average NYC taxi passenger counts:

```
from sktime.detection.stray import STRAY

model = STRAY(k=7, alpha=0.05)
model.fit(tx['value'])
output = model.transform(tx['value'])

# True if anomalous, false otherwise
outliers = tx[output]
print(outliers)

>>
           value
timestamp
2014-11-01  20553.500000
2014-12-25   7902.125000
2014-12-26  10397.958333
2015-01-26   7818.979167
2015-01-27   4834.541667
```

The STRAY algorithm detected five potential outliers, of which three are known outliers. Interestingly, it also identified December 26, 2014 (the day after Christmas), and January 26, 2015 (the day before the blizzard), suggesting that these events had extended effects on taxi usage patterns.

You can visualize these points using the `plot_outliers` function:

```
plot_outliers(outliers, tx, 'Outliers using STRAY', labels=True)
```

This should produce a plot similar to that in *Figure 7.2*, except the X markers are based on the outliers identified using the STRAY method:

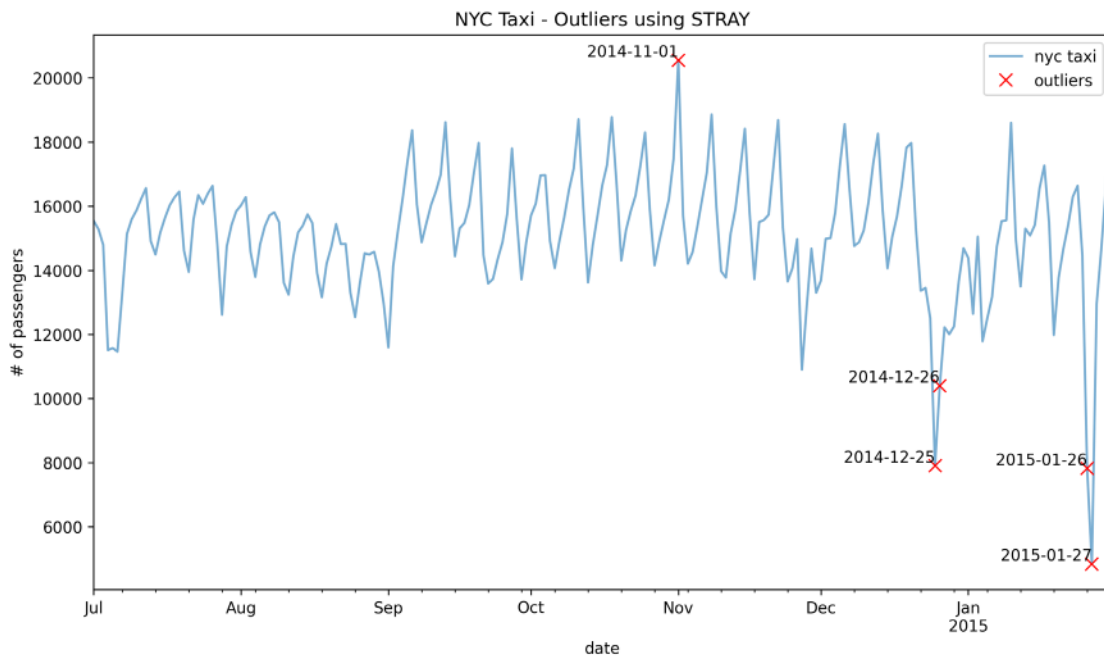


Figure 7.18: Daily average taxi passengers and outliers identified using the STRAY method

Detecting outliers with Matrix Profile

Matrix Profile is a powerful technique for time-series analysis that finds patterns and anomalies by comparing every possible **subsequence** of your data against all other subsequences. A subsequence in Matrix Profile is a continuous segment of fixed length extracted from your time-series data. Think of it as a “window” that slides across your time series, capturing snapshots of your data’s behavior over time.

Matrix Profile is fundamentally different from the other outlier detection methods you have been introduced to so far in this chapter. Rather than looking at individual points (point outliers) and their statistical properties, Matrix Profile does the following:

- Analyzes subsequence similarity patterns within your time series
- Identifies subsequences (windows of data) that are most dissimilar from all other subsequences in your dataset
- Focuses on finding **pattern anomalies** rather than **point anomalies**

Matrix Profile computes the **Euclidean distance** between each subsequence and its nearest non-trivial match elsewhere in the time series, creating a “profile” of these minimum distances. This profile essentially measures how similar or dissimilar each segment of your data is to any other segment in the dataset.



Euclidean distance

Euclidean distance measures the straight-line distance between two points in space. In Matrix Profile, it quantifies how different two subsequences are by calculating the square root of the sum of squared differences between corresponding data points. The higher the Euclidean distance, the more dissimilar the patterns are, helping identify unusual sequences that don't match other parts of your time series.

For two subsequences, A and B , each containing m points, the Euclidean distance is calculated as follows:

$$d(A, B) = \sqrt{\sum_{i=0}^{m-1} (A_i - B_i)^2}$$

It's fundamentally a pattern anomaly detector rather than a point anomaly detector.

Other methods we've used so far work differently:

- **Z-score/modified z-score:** Identify points that deviate significantly from the mean, capturing extreme values (very high or low taxi usage)
- **Hampel filter:** Uses a moving window median to identify local outliers
- **Tukey fences (IQR):** Flags values outside $1.5 \times$ the IQR
- **STRAY:** Looks for points that deviate from nearest neighbors, considering concept drift

For example, if our subsequence length is set to 30 ($m = 30$), we're essentially looking for 30-day patterns that don't resemble any other 30-day pattern in our dataset.

The choice of subsequence length is crucial because it defines what constitutes a "pattern" in our data. For our average daily taxi data, we have the following:

- **Length 1-2:** Detect unusual point transitions (close to point anomalies)
- **Length 7:** Detect unusual weekly patterns—such as weeks where the typical weekday/weekend pattern was disrupted (such as holiday weeks where Monday behaves more like a Sunday)
- **Length 30:** Detect unusual monthly patterns—identifying months with unusual seasonal transitions or extended periods of abnormal taxi demand due to events or weather patterns
- **Length 365:** Detect unusual annual patterns

This flexibility makes Matrix Profile powerful but also requires careful tuning based on what kind of anomalies you're looking for. If you're primarily interested in finding days with unusual taxi usage regardless of pattern context, a very small subsequence length would indeed be more appropriate.

Here is an example using our taxi data where we search for unusual 30-day patterns:

```
from sktime.transformations.panel.matrix_profile import MatrixProfile
# Set the subsequence length (window size)
subsequence_length = 30

# Create the MatrixProfile transformer
mp_transformer = MatrixProfile(m=subsequence_length)
mp_result = mp_transformer.fit_transform(tx)

# The points with highest matrix profile values are the most anomalous
# You can set a threshold or take the top N highest values
threshold = np.percentile(mp_result, 95) # Top 5% as anomalies
anomalies = mp_result > threshold

anomalies_series = anomalies.iloc[0]
anomaly_indices = np.where(anomalies_series)[0]

anomaly_timestamps = tx.index[anomaly_indices]

outliers = tx.loc[anomaly_timestamps]
print(outliers)
>>
```

	value
timestamp	
2014-08-09	15499.708333
2014-08-10	13607.979167
2014-08-11	13239.645833
2014-08-12	14426.500000
2014-08-13	15178.916667
2014-08-14	15394.395833
2014-11-27	10899.666667
2014-12-24	12502.000000
2014-12-25	7902.125000
2014-12-29	12250.479167

You can visualize these points using the `plot_outliers` function:

```
plot_outliers(outliers, tx, 'Outliers using Matrix Profile')
```

This should produce a plot similar to that in *Figure 7.2*, except the X markers are based on the outliers identified using the Matrix Profile method:

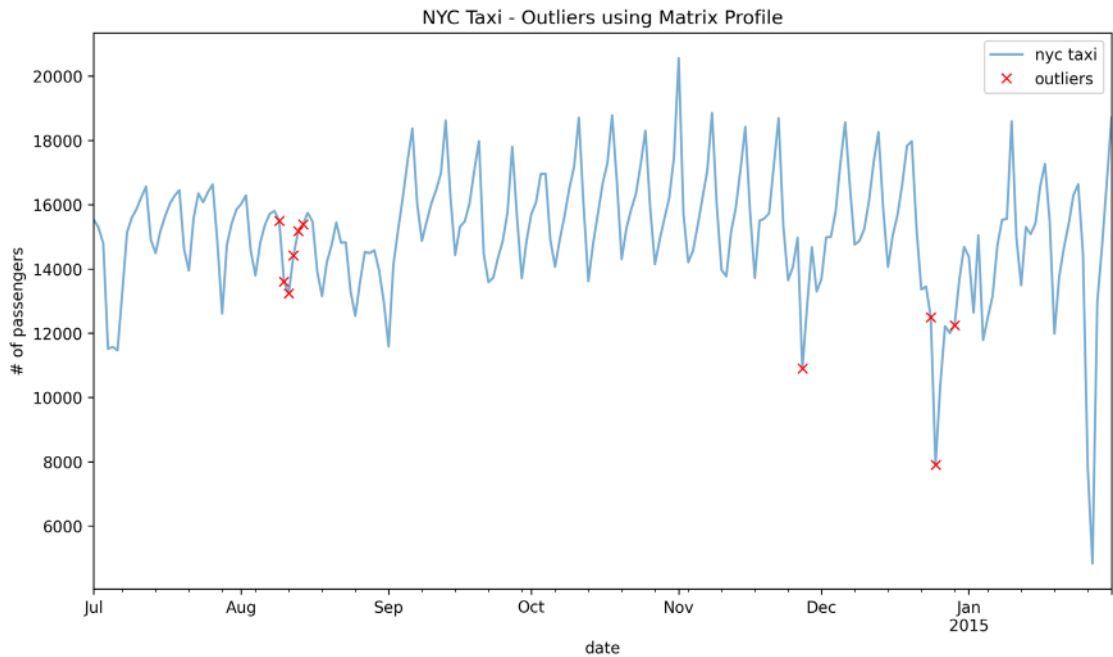


Figure 7.19: Matrix Profile with a subsequence length of 30 days

Overall, Matrix Profile successfully identified two of our known outliers: Thanksgiving (Nov 27) and Christmas (Dec 25). Interestingly, Matrix Profile identified a series of days in August 2014 (8/9–8/14) as outliers, which none of the other methods detected. This indicates that this period had a pattern of passenger counts that was highly unusual compared to other 30-day patterns in your dataset. This demonstrates how Matrix Profile can uncover pattern anomalies that might be missed by point-based methods.

In summary, when looking for individual data points that are statistical outliers, methods such as z-score, modified z-score, Tukey’s method, or the Hampel filter are often more appropriate. However, if you’re interested in finding anomalous periods or patterns—sequences of data that behave differently from the rest of your time series—Matrix Profile provides a powerful and flexible approach.

See also

- To learn more about **Matrix Profile**, you can read the documentation from STUMPY here: https://stumpy.readthedocs.io/en/latest/Tutorial_The_Matrix_Profile.html
- To learn more about **STRAY** implementation in sktime, you can read the documentation here: https://www.sktime.net/en/v0.23.1/api_reference/auto_generated/sktime.annotation.stray.STRAY.html
- To learn more about the **Hampel filter** implementation in sktime, you can read the documentation here: https://www.sktime.net/en/latest/api_reference/auto_generated/sktime.transformations.series.outlier_detection.HampelFilter.html

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



8

Exploratory Data Analysis and Diagnosis

In previous chapters, we covered essential data preparation techniques. In *Chapter 1* and *Chapter 2*, we extracted data from various sources, including files and databases. Then in *Chapter 5*, we explored handling time-related transformations and feature engineering. *Chapter 6* focused on handling missing data. These chapters provided the foundation to help you prepare, clean, and adjust your time series datasets.

With data collection and preparation in place, we can now delve into **Exploratory Data Analysis (EDA)** to better understand the underlying time series processes behind the data. Before diving into modeling or advanced analysis, a critical first step is to thoroughly inspect the data at hand. This involves examining specific time series characteristics such as stationarity, trends, seasonality, and autocorrelation, among others. Understanding these characteristics, combined with domain knowledge about the process itself, is vital for accurate analysis and meaningful interpretation.

This chapter builds on your previous learning to prepare you for creating and evaluating forecasting models, starting from *Chapter 9* and beyond.

In this chapter, you will learn how to visualize time series data, decompose a time series into its components (trend, seasonality, and the residual random process), test for key assumptions that your models may rely on (such as stationarity, normality, and homoskedasticity), and explore techniques to transform the data to satisfy some of these assumptions.

While the techniques introduced in this chapter provide valuable insights into the structure and characteristics of time series data, it is important to note that not all of them are necessary for every time series analysis. The relevance of these methods depends on the specific requirements of your problem and the assumptions of the models you intend to use. For instance, some methods, such as ARIMA, require stationarity, whereas others, such as exponential smoothing, do not. Similarly, seasonality analysis is only relevant if your data exhibits periodic patterns.

The recipes that you will encounter in this chapter are as follows:

- Decomposing time series data
- Detecting time series stationarity
- Applying power transformations
- Testing for autocorrelation

By the end of this chapter, you will have a robust toolkit to explore and diagnose time series data, ensuring that you are well-prepared for modeling and forecasting in later chapters.



Don't miss the GitHub-exclusive recipes— *Chapter 8-1: Plotting time series data using pandas*, and *Chapter 8-2: Plotting time series data with interactive visualizations using hvPlot*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Technical requirements

You can download the Jupyter notebooks and datasets needed from the GitHub repository to follow along:

- **Jupyter notebooks:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch8/Chapter_8.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch8>

This chapter extensively uses pandas 2.3.3 (released September 29, 2025), which is the latest version as of this writing. You can use a more recent version if available.

Throughout this chapter, you will work with three datasets: **Closing Price Stock Data**, **CO2**, and **Air Passengers**. The CO2 and Air Passengers datasets are provided by the statsmodels library. The Air Passengers dataset contains monthly airline passenger numbers from 1949 to 1960. The CO2 dataset contains weekly atmospheric carbon dioxide levels on Mauna Loa from 1958 to 2001. The Closing Price Stock Data dataset includes Microsoft, Apple, and IBM stock prices from November 2019 to November 2021.

To keep the code concise, the dataset loading and comparison utilities for this chapter are encapsulated in a helper module, `utils.py`.

You can import and load all three time series as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
from utils import load_ch8_datasets, plot_comparison

airp_df, closing_price, co2_df = load_ch9_datasets()
```

Now you have three pandas DataFrames ready for use in the decomposition recipes: `airp_df` (airline passengers), `closing_price` (daily closing prices), and `co2_df` (weekly CO2 concentrations).

Decomposing time series data

When conducting time series analysis, one key objective often involves **forecasting**, where you build a model capable of making future predictions. Before starting the modeling process, it is crucial to analyze the components of the time series to better understand its underlying patterns. This step provides critical insights and significantly informs decisions throughout the modeling process.

A time series typically comprises **three** main components: **trend**, **seasonality**, and the **residual** (also called **noise** or an **irregular component**). Some time series may also exhibit **cyclical** components, which are patterns that occur over longer and irregular intervals, unlike the fixed seasonal intervals. For statistical models that require the time series to be stationary, estimating and subsequently removing the trend and seasonality components from the time series might be necessary. Techniques and libraries for time series decomposition generally provide visual representations and identification of trends, seasonality, and the residual random process.

- **Trend:** The long-term direction of the time series, which can be upward, downward, or horizontal. For instance, a sales time series might exhibit an upward trend, reflecting growing demand over time.
- **Seasonality:** Regular patterns that repeat at fixed intervals (e.g., monthly, quarterly, or annually). For example, sales data might experience an annual spike during the holiday season.
- **Residual** (or noise): The irregular or random portion of the time series, which represents the portion of the time series that remains once the trend and seasonality have been extracted. This component accounts for unexplained variability in the data.
- **Cyclical components:** In some time series, these represent longer-term patterns that occur over irregular intervals, often seen in economic or financial data (e.g., business cycles).

Time series **decomposition** is the process of separating a time series into these components and estimating the trend and seasonality as distinct models. Depending on the nature of the interaction between the components, the decomposition can be classified as either **additive** or **multiplicative**:

- **Additive:** When the original time series can be reconstructed by adding the components together (combine linearly):

$$y_t = T_t + S_t + R_t$$

An additive decomposition model is reasonable when the seasonal variations do not change over time (remain constant).

- **Multiplicative:** Assumes the time series can be reconstructed by multiplying the components (interact proportionally):

$$y_t = T_t \times S_t \times R_t$$

A *multiplicative* model is suitable when seasonal variations fluctuate and scale proportionally with the level of the trend over time.

Decomposing a time series is particularly useful when visualizing patterns, understanding how the components interact, and diagnosing stationarity. However, it is not always a mandatory step before modeling, and its relevance depends on the modeling approach and data characteristics:

- Some models, such as ARIMA or SARIMA, inherently handle trend and seasonality without requiring explicit decomposition (see *Chapter 9*).
- Modern machine learning approaches, such as neural networks, can model all components together without pre-processing (see *Chapter 11* and *Chapter 12*).

That said, decomposition remains a valuable tool for visualizing and understanding the structure of time series, diagnosing stationarity, and validating assumptions about your data. Additionally, time series components can be grouped as either **predictable** or **unpredictable**:

- **Predictable** components are consistent, repeating patterns that can be captured and modeled (trend and seasonality)
- On the other hand, **unpredictable** components show irregularity, often called **noise** (or **residual**), and represent the random or irregular variability remaining after extraction of the predictable parts

In this recipe, you will explore different techniques for **decomposing** your time series available in the `statsmodels` library, including the following:

- **Seasonal decomposition** (`seasonal_decompose`): A classical method for decomposing time series into trend, seasonality, and residual components using moving averages. This method is simple, straightforward, and a good starting point for basic time series analysis.
- **Seasonal and trend decomposition with LOESS (STL)**: A more robust method that uses locally estimated scatterplot smoothing (LOESS) for a more flexible trend and seasonal estimation. This method is also robust to outliers and more suitable for time series with non-linear trends or irregular seasonality.
- **Hodrick-Prescott filter** (`hp_filter`): A smoothing technique primarily used for separating long-term trend and cyclical components in time series data. This technique is widely used in macroeconomics and in analyzing economic time series.

Getting ready

You can download the Jupyter notebooks and datasets needed from the GitHub repository. Additionally, you will be using `co2_df` and `airp_df`, created in the *Technical requirements* section of this chapter.

You will be using the `statsmodels` library.

To install it with `pip`, you can use this:

```
pip install statsmodels
```

As of this writing, the latest version of `statsmodels` is 0.14.5. You can use a more recent version if available.

How to do it...

You will explore two methods available in the `statsmodels` library: `seasonal_decompose` and `STL`. In the following *There's more...* section, you will explore `hp_filter`.

Using seasonal_decompose

The `seasonal_decompose` function relies on **moving averages** to decompose a time series into trend, seasonality, and residual components. You will use the CO2 dataset and the Air Passenger dataset (refer to the *Technical requirements* section) to observe additive and multiplicative behavior.

1. Import the libraries needed and set `rcParams` to ensure the generated visuals are large enough. By default, plots produced by `statsmodels` are small, so we can increase the figure size globally using `rcParams["figure.figsize"]`:

```
from statsmodels.tsa.seasonal import seasonal_decompose, STL
plt.rcParams["figure.figsize"] = [10, 5]
```

This ensures all plots in the notebook are consistently sized, with a width of 10 inches and a height of 5 inches (*W x H*).

You can apply a style such as the grayscale theme:

```
plt.style.use('grayscale')
```

2. Before decomposing, visualize the **CO2 dataset** to determine whether the model should be **additive** or **multiplicative**:

```
co2_df.plot(title='Mauna Loa Weekly Atmospheric CO2 Data');
```

This should display a line chart showing weekly carbon dioxide levels measured in **parts per million (ppm)** from 1960 to 2000. When using the `.plot()` method, the default chart type is a line chart with the `kind="line"` parameter. For more information about pandas' plotting capabilities, refer to the *Plotting time series data using pandas* recipe in the *Bonus Recipes (Chapter 8-1)* section on GitHub.

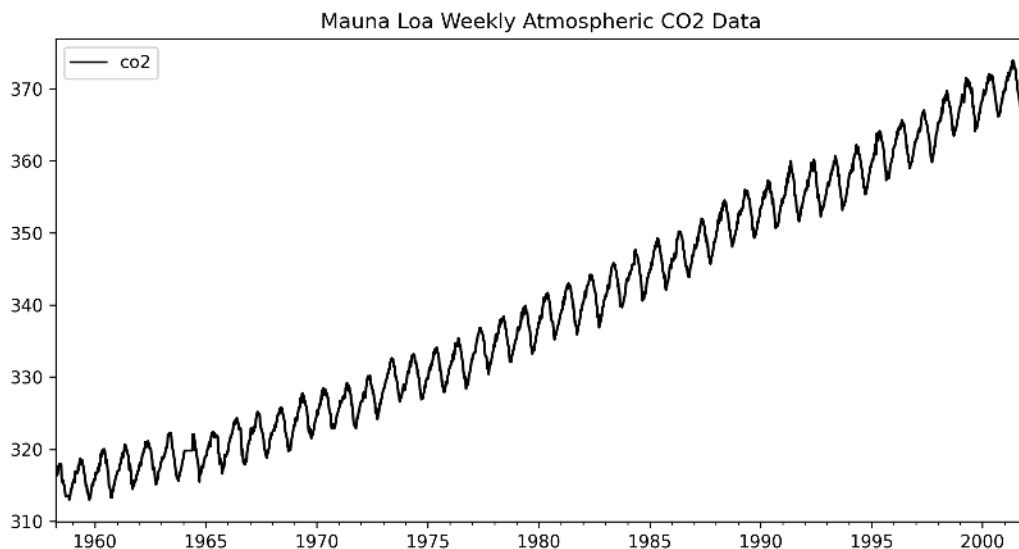


Figure 8.1: The CO2 dataset showing an upward trend and constant seasonal variation

The plot for `co2_df` shows an upward linear trend with a **constant** seasonal variation (repeated seasonal pattern at a constant rate). This suggests that the CO2 dataset is an **additive model**.

Similarly, explore the Air Passenger dataset to observe whether the seasonality shows *multiplicative* or *additive* behavior:

```
airp_df['passengers'].plot(
    title='Monthly Airline Passengers Numbers 1949-1960');
```

This should produce a line chart showing the number of passengers per month from 1949 to 1960:

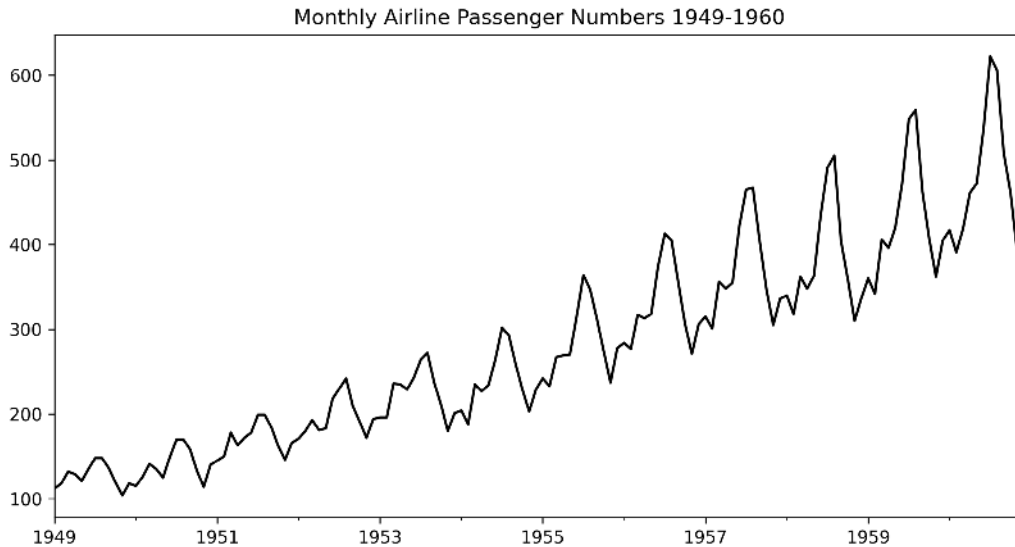


Figure 8.2: The Air Passengers dataset showing trend and increasing seasonal variation

The plot for `airp_df` shows an upward trend with **increasing** seasonal variations (seasonality fluctuations seem to be increasing), suggesting a **multiplicative model**.

3. Use `seasonal_decompose` on the two datasets. Use an **additive** decomposition for the CO2 dataset and **multiplicative** decomposition for the Air Passenger dataset:

```
co2_decomposed = seasonal_decompose(co2_df,model='additive')
air_decomposed = seasonal_decompose(airp_df,model='multiplicative')
```

Both `co2_decomposed` and `air_decomposed` give you access to the decomposed components using `.trend`, `.seasonal`, and `.resid`, or you can plot all three components by using the `.plot()` method:

```
air_decomposed.plot();
```

The following plot is the result:

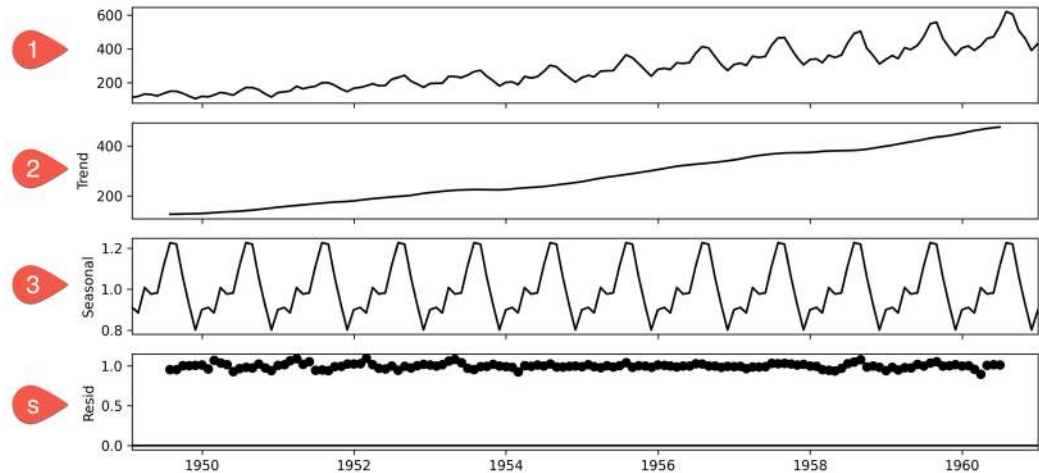


Figure 8.3: Air Passengers dataset multiplicative decomposition into trend, seasonality, and residual

Let's understand the four parts presented in the plot (numbered in Figure 8.3):

1. This is the original observed data (monthly air passenger dataset) that we are decomposing.
2. The **trend** component highlights the long-term direction of the time series. For the Air Passenger dataset, the trend shows a steady upward movement over time. Generally, the trend indicates whether there is positive (increasing or upward), negative (decreasing or downward), or constant (no trend or horizontal) long-term movement.
3. The **seasonal** component shows the seasonality effect in terms of repeating patterns of highs and lows over fixed intervals.
4. Finally, the **residual** (also called **noise**) component shows the random variation in the data after removing trend and seasonality. In this case, a multiplicative model was used.

Similarly, you can plot the decomposition of the CO2 dataset:

```
co2_decomposed.plot();
```

This should produce the following plots:

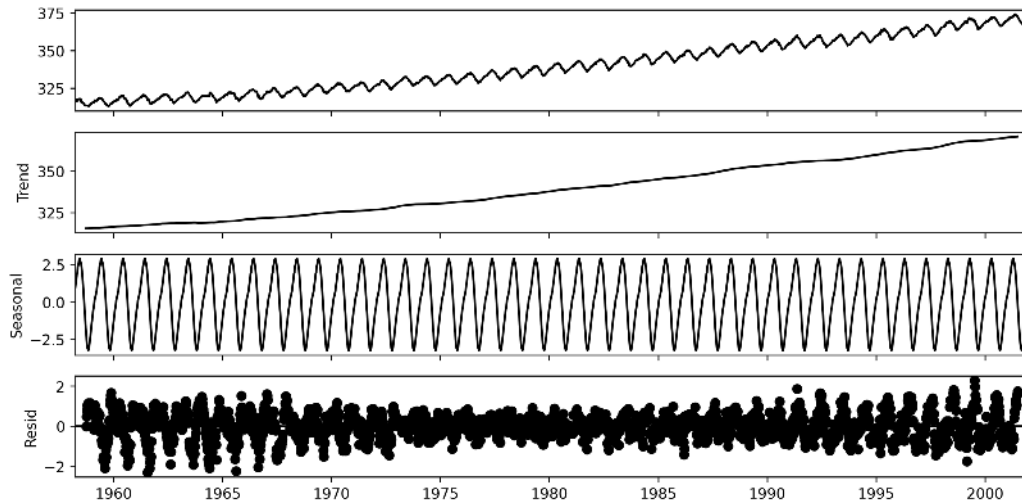


Figure 8.4: CO2 additive decomposition into trend, seasonality, and residual

The CO2 dataset shows a clear upward trend alongside a repeating seasonal pattern. Unlike the Air Passenger dataset, where the seasonal variations grow in magnitude over time (characteristic of a **multiplicative** model), the seasonal fluctuations in the CO2 dataset remain steady (constant) throughout the observed period (characteristic of an **additive** model).

4. For a multiplicative model, the original time series is reconstructed by multiplying its components: trend, seasonality, and residuals. In contrast, for an additive model, the components are added together. To demonstrate, use `air_decomposed`, an instance of the `DecomposeResult` class. This class provides access to the `trend`, `seasonal`, and `resid` attributes, along with the `.plot()` method for visualization.

The following code reconstructs the time series for the Air Passengers dataset using a **multiplicative** model:

```
(air_decomposed.trend *
 air_decomposed.seasonal *
 air_decomposed.resid).plot();
```

Similarly, for the CO2 dataset, which uses an **additive** model, you can reconstruct the time series by summing its components:

```
(co2_decomposed.trend +
 co2_decomposed.seasonal +
 co2_decomposed.resid).plot();
```

Using STL

Another decomposition option within `statsmodels` is **STL**. STL stands for *Seasonal and Trend decomposition using LOESS*, which is a more advanced decomposition technique. In `statsmodels`, the STL class requires additional parameters than the `seasonal_decompose` function.

- **endog**: The input time series to decompose (one-dimensional NumPy array or a pandas Series). Both of our DataFrames (`co2_df` and `airp_df`) meet this criterion as they consist of one column with a `DatetimeIndex`.
- **Period** (optional): Specifies the length of the seasonal cycle. For example, `period=12` for annual seasonality in monthly data, or `period=52` for annual seasonality in weekly data. If the pandas Series has a defined frequency, then `period` is automatically **inferred**.
- **seasonal** (optional): The length of the seasonal smoother window. It must be an odd integer and should be greater than or equal to 7 (the default is 7). Larger values result in a smoother seasonal component, while a smaller value makes it more responsive to local variations.
- **trend** (optional): The length of the trend smoother window. It must be an odd integer value, typically larger than the period and seasonal values. If not provided, it defaults to a value calculated based on the original STL implementation as $1.5 \times \text{period} / (1 - 1.5/\text{seasonal})$.
- **robust** (optional): A Boolean flag (True or False) indicating whether to use a robust version of LOESS (weighted version) that reduces sensitivity to outliers. Setting `robust=True` helps remove the impact of outliers on seasonal and trend components when calculated. The default value is `False`.

You can check the **frequency** of the two datasets to confirm the period inference:

- This indicates a **weekly** frequency for the CO2 dataset:

```
print(co2_df.index.freq)
>>
<Week: weekday=5>
```

- This indicates a **monthly** frequency for the Air Passenger dataset.

```
print(airp_df.index.freq)
>>
<MonthEnd>
```

Now, we have all the information needed to use **STL**:

1. You will use STL to decompose the **CO2 Dataset** DataFrame. The STL class automatically infers period since the frequency is already defined in `DatetimeIndex`:

```
co2_stl = STL(
    co2_df,
    seasonal=13,
    robust=True
).fit()
```

Alternatively, you can explicitly specify `period=52` (weekly seasonality):

```
co2_stl = STL(
    co2_df,
    seasonal=13,
    period=52,
    robust=True
).fit()
```

2. To plot the trend, seasonal, and residual components for the CO2 dataset, use the `.plot()` method:

```
co2_stl.plot();
```

This should produce subplots resembling those produced using `seasonal_decompose`:

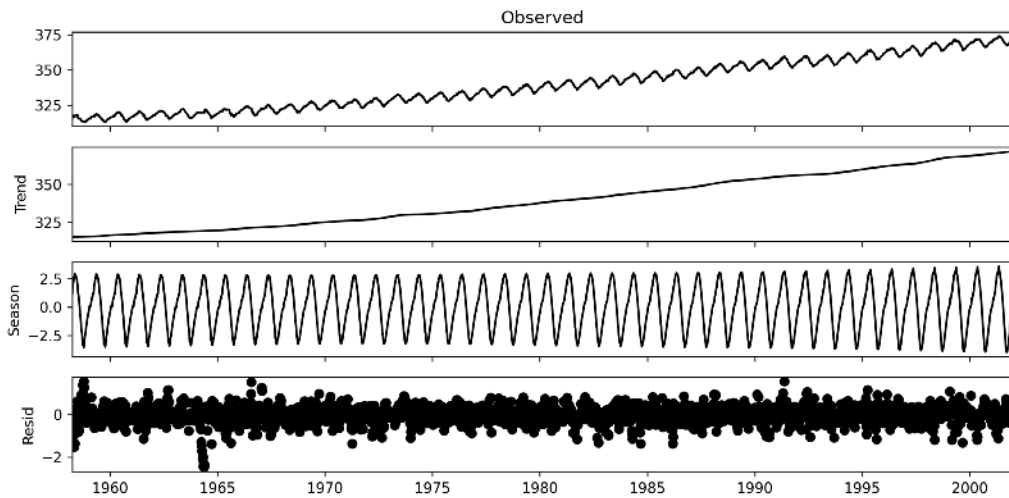


Figure 8.5: Decomposing the CO2 dataset using STL

Compare the output in *Figure 8.4* to that in *Figure 8.5*. The seasonal component in *Figure 8.5* shows a constant pattern over time.

Notice that when you used STL, you provided `seasonal=13`, which represents the number of consecutive observations used to smooth the seasonal component. Our CO2 dataset is weekly, so `seasonal=13` means that 13 weeks are used for smoothing. The `seasonal` parameter must be an odd integer greater than or equal to 7. Start with the smallest odd value and increase it incrementally to achieve smoother seasonality if needed (for example, an odd number between 7 and 53).

Note that STL works only with additive decompositions. If the time series displays a multiplicative structure (increasing seasonal variation over time, as seen in the Air Passengers dataset), you must first transform it to an additive structure. This can be accomplished using **log transformation** (taking the logarithm).

Apply log transformation to the `airp_df` DataFrame using `np.log()` from the NumPy library:

```
import numpy as np
airp_stl = STL(
    np.log(airp_df),
    period=12,
    seasonal=13,
    robust=True
).fit()

airp_stl.plot();
```

This should produce the following plot:

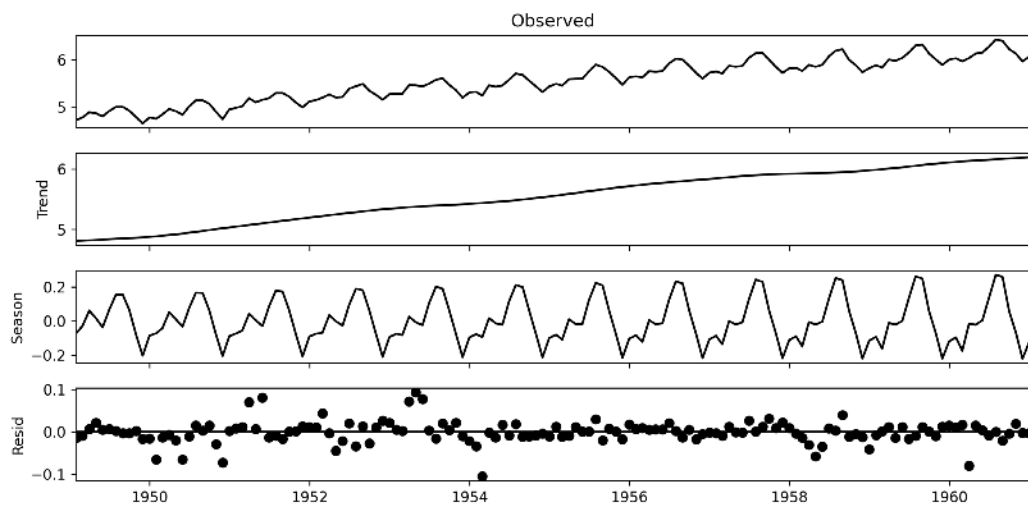


Figure 8.6: Decomposing the Air Passenger dataset using STL

The `period=12` was inferred automatically because the Air Passenger DataFrame has a monthly frequency. For the seasonal smoother, a typical range would be any odd number between 7 and 13.

The seasonal component in Figure 8.6 exhibits an increasing magnitude over time, consistent with our earlier observation of multiplicative models. In STL, seasonality is allowed to vary smoothly over time, making it adaptive for complex patterns.

- Both `seasonal_decompose` and `STL` produce a `DecomposeResult` object, which gives access to the residuals through `.resid`. You can compare the residuals for the CO2 dataset:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 4))
co2_stl.resid.plot(ax=ax1,
                    title='CO2 Residual Plot from Seasonal Decomposition')
co2_decomposed.resid.plot(ax=ax2, title='CO2 Residual Plot from STL');
```

This should produce the following figure with two plots comparing the `seasonal_decompose` and `STL` approaches.

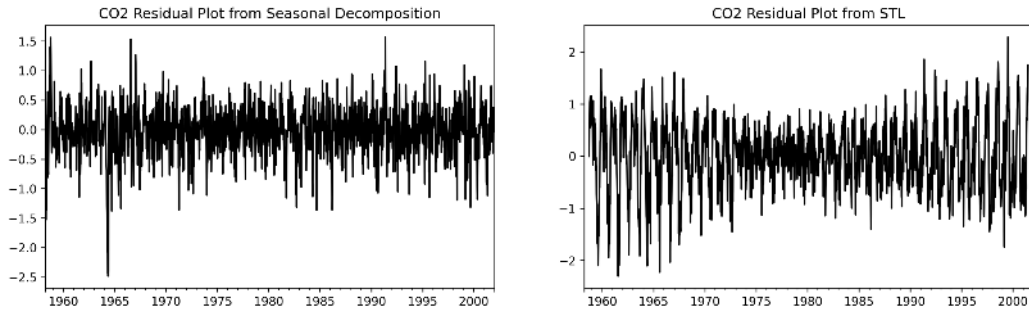


Figure 8.7: Comparing residual plots from seasonal decomposition and STL for the CO2 dataset

- Similarly, compare residuals for the Air Passenger dataset using `seasonal_decompose` and `STL`. Since `STL` was applied to the log-transformed data, you need to **back-transform** (reverse the transformation) using the exponential function `np.exp()`. This ensures a fair comparison between the residuals obtained from `seasonal_decompose`, which used the raw data.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 4))
airp_stl.resid.plot(
    ax=ax1,
    title='Air Passenger Residual Plot from Seasonal Decomposition')
# back transform using np.exp
np.exp(air_decomposed.resid).plot(
    ax=ax2, title='Air Passenger Residual Plot from STL');
```

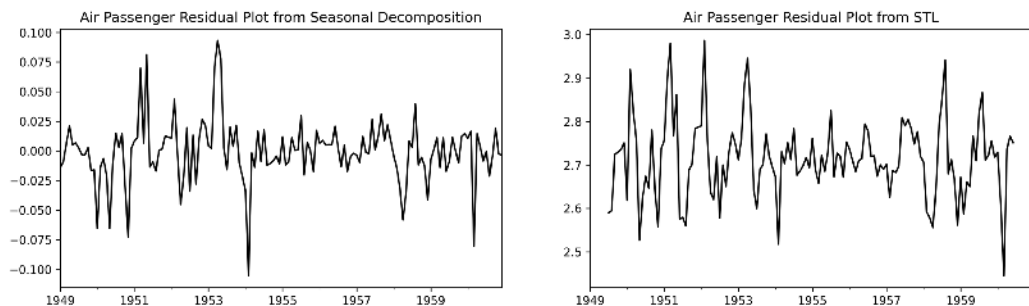


Figure 8.8: Comparing residual plots from seasonal decomposition and STL for the Air Passenger dataset

You will notice that the residuals (in *Figure 8.7* and *Figure 8.8*) differ between `seasonal_decompose` and `STL`. This highlights that the two decomposition methods are capturing time series components (trend, seasonality, and residuals) using distinct smoothing techniques. While `seasonal_decompose` relies on simple moving averages, `STL` uses **LOESS**, which makes it more adaptable to irregular seasonality and robust to outliers.

How it works...

In this recipe, you explored two different approaches for time series decomposition: `seasonal_decompose` and `STL`. Both methods decompose a time series into trend, seasonal, and residual components, but they differ in their underlying techniques and flexibility.

`STL` utilizes **LOESS smoothing (Locally Estimated Scatterplot Smoothing)** to decompose a time series into trend, seasonal, and residual components. `STL` is generally more robust than the `seasonal_decompose` method, especially when dealing with non-linear trends or complex seasonal patterns. However, `STL` supports only **additive decompositions**, while `seasonal_decompose` allows for both additive and multiplicative models. To use `STL` with data that exhibits multiplicative seasonality (where seasonal fluctuations increase proportionally with the trend), you can first apply a transformation, such as a **log transform**, to convert it into an additive form.

Both `seasonal_decompose` and `STL` approaches are valuable for extracting seasonality from time series data, which enables a clearer observation of the underlying trend.

The following outlined steps are a high-level, conceptual overview of the underlying algorithm for `seasonal_decompose` and `STL`:

For `seasonal_decompose`:

- Smooth the time series data to estimate the trend component using a convolution filter.
- Once the trend is estimated, it is then removed from the time series (de-trended).
- The remaining de-trended time series is then averaged for each period or seasonal group. The seasonal averaging takes all data values in the time series corresponding to each season and averages them (for example, we take every January from each year and average that).
- Once the seasonal component is estimated, it is then removed, and you are left with the residuals.

For `STL`:

- Similar to `seasonal_decompose`, the time series is smoothed to estimate the trend, but in `STL`, this is done using LOESS instead of a simple convolution filter.
- Once the trend is estimated, it is then removed from the time series (de-trended).
- For the seasonal component, `STL` applies LOESS smoothing to the de-trended data, separately for each seasonal period, to estimate the seasonal component.
- Once the seasonal component is estimated, it is then removed, and you are left with the residuals.

The key difference is that `STL` uses the more advanced LOESS technique for both trend and seasonal estimation, whereas `seasonal_decompose` uses simpler moving averages.

There's more...

The **Hodrick-Prescott filter** is a smoothing technique that can be used to separate short-term fluctuations and other deviations from long-term trends. It is implemented as `hp_filter` in the `statsmodels` library.

Unlike `seasonal_decompose` and `STL`, which decompose a time series into three components (trend, seasonal, and residual), `hp_filter` produces only two components:

- **Trend component:** Represents the **long-term** movement
- **Cyclical component:** Captures **all deviations** from the trend, including short-term fluctuations, noise, business cycles, and potentially seasonality

In simple terms, `hp_filter` isolates the long-term trend as one component. Everything that remains after extracting the long-term trend is grouped into the cyclical component (a composite of various short-term variability).

Start by importing the `hpfiler` function from the `statsmodels` library:

```
from statsmodels.tsa.filters.hp_filter import hpfiler
```

Decompose the CO2 dataset into trend and cyclical components as shown:

```
co2_cyclic, co2_trend = hpfiler(co2_df)
```

The `hpfiler` function returns two pandas Series: the first for the cyclical component and the second for the trend component.

Visualize the components side by side (`co2_cyclic` and `co2_trend`) to gain a better understanding of what the Hodrick-Prescott filter extracted:

```
fig, ax = plt.subplots(1, 2, figsize=(16, 4))
co2_cyclic.plot(ax=ax[0], title='CO2 Cyclical Component')
co2_trend.plot(ax=ax[1], title='CO2 Long-term Trend Component');
```

This should produce two subplots on the same row (side by side), as shown:

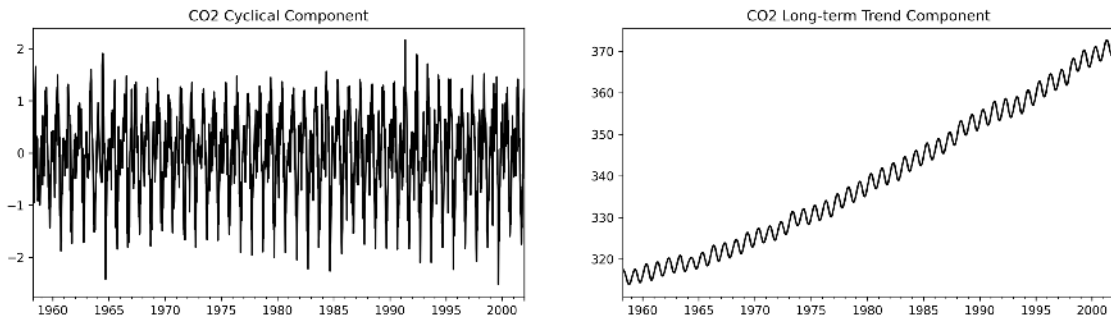


Figure 8.9: Cyclical and trend components using the Hodrick-Prescott filter

Because the `hpfiler` components are **additive**, you can reconstruct the original CO2 series by summing the trend and cyclical components (for example, using `(co2_cyclic+co2_trend).plot()`), which produces a plot that closely matches the original CO2 series in *Figure 8.1*.

The same process can be applied to the Air Passengers dataset:

```
airp_cyclic, airp_trend = hpfiler(airp_df)
fig, ax = plt.subplots(1,2, figsize=(16, 4))
airp_cyclic.plot(ax=ax[0], title='Air Passengers Cyclical Component')
airp_trend.plot(ax=ax[1], title='Air Passengers Long-term Trend Component');
```

This produces the following output:

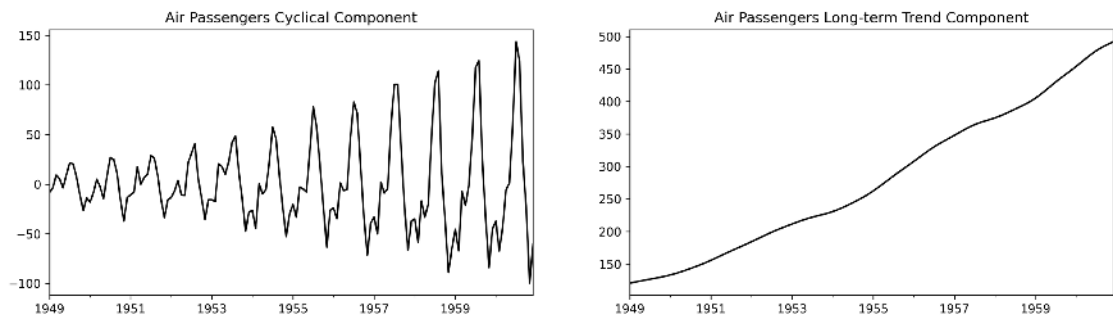


Figure 8.10: Cyclical and trend components for the Air Passenger dataset using the Hodrick-Prescott filter

See also

- To learn more about `hpfiler()`, visit the official documentation page here: https://www.statsmodels.org/stable/generated/statsmodels.tsa.filters.hp_filter.hpfilter.html
- To learn more about `seasonal_decompose()`, visit the official documentation page here: https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html
- To learn more about `STL()`, visit the official documentation page here: <https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.STL.html#statsmodels.tsa.seasonal.STL>

Detecting time series stationarity

Many traditional time series forecasting techniques assume a **stationary** time series process. Thus, it is crucial to determine whether the time series you are working with (the observed time series or the realization you have) originates from a **stationary** or **non-stationary** process.

A **stationary** time series suggests that specific statistical properties, such as the mean, variance, and autocovariance, do not change over time and remain steady, making the processes easier to model and predict. In other words, this stability makes it easier to model and analyze the data because the relationships between data points remain consistent. Conversely, a **non-stationary** process means these statistical properties change over time, making it more challenging to model due to its dynamic nature and variations over time. If the time series is non-stationary, it may need to be transformed into a stationary series before applying forecasting techniques that assume stationarity.

There are different ways to define stationarity, ranging from strict conditions (may not be observable in real-world data), known as **strong stationarity**, to those more modest in their criteria that can be observed (or transformed into) in real-world data, called **weak stationarity**.

In this recipe, and for practical reasons, the term **stationarity** implies *weak* stationarity, defined as a time series with a constant mean called *mu* (μ), a constant variance called *sigma squared* (σ^2), and a consistent covariance (or autocorrelation) between identical distanced periods (lags). Having a stable mean and variance simplifies modeling since you are not solving for them as functions of time.

Generally, a time series with trend or seasonality can be considered non-stationary. Simple visual inspections, such as line plots, can often help identify trends or seasonal patterns and assess stationarity. In such cases, a simple line plot would suffice.

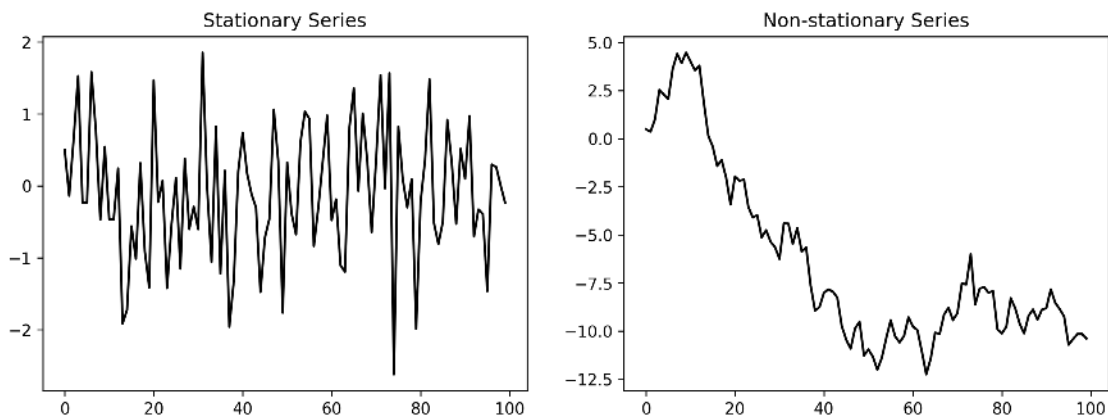


Figure 8.11: Illustration of a stationary and a non-stationary time series

However, for numerical assessment, statistical tests are more reliable for detecting stationarity. In this recipe, you will learn how to use statistical tests to evaluate stationarity and explore techniques for transforming a non-stationary series into stationary.

In `statsmodels`, two widely used statistical tests for stationary are the **Augmented Dickey-Fuller (ADF)** test and the **Kwiatkowski-Phillips-Schmidt-Shin (KPSS)** test:

- **ADF test**
 - Null hypothesis (H_0): The series has a unit root and thus is **non-stationary**
 - Alternative hypothesis (H_A): The series is **stationary**

- **KPSS test:**
 - Null hypothesis (H_0): The series is **stationary**
 - Alternative hypothesis (H_A): The series has a unit root and thus is **non-stationary**

Both ADF and KPSS test for **unit roots** in a univariate time series process, but have different hypotheses. Unit roots are just one cause for a time series to be non-stationary, but generally, the presence of unit roots indicates non-stationarity.

Both ADF and KPSS employ linear regression as their foundation and are types of statistical hypothesis tests. Therefore, you will need to interpret the test results to determine whether you can reject or fail to reject the **null hypothesis**. Generally, you can use the returned **p-values** to guide your decision whether to reject or fail to reject the null hypothesis.



Remember, the interpretation for ADF and KPSS test results differs due to their **opposite null hypotheses**. If the **p-value** is less than the significance level (usually < 0.05), you can **reject the null hypothesis**. For ADF, it would suggest the time series does not have a unit root and is likely **stationary**, but for KPSS, it would suggest a **non-stationary series**. If the p-value is greater than or equal to 0.05, then you **fail to reject the null hypothesis**. For ADF, it would suggest the series is non-stationary, but for KPSS, it would suggest a stationary series.

Getting ready

Refer to the *Technical requirements* section on how to download the Jupyter notebooks and datasets needed.

In this recipe, you will be using the CO2 dataset, which was previously loaded as a pandas DataFrame in the *Technical requirements* section of this chapter.

How to do it...

In addition to visually analyzing a time series plot to identify stationarity, statistical tests offer a more concrete and numerical approach. **Unit root tests** such as the **ADF** and **KPSS** tests provide a formal approach for detecting stationarity.

In *Figure 8.1*, the CO2 dataset exhibits an upward trend and a recurring seasonal pattern (annual). Generally, when trend or seasonality exists (in this case, both), we can consider the time series to be non-stationary. However, it's not always this easy to identify stationarity or lack of it visually, and therefore, you will rely on statistical tests.

You will use both the `adfuller` and `kpss` tests from the `statsmodels` library and interpret their results to determine stationarity. Remember, these tests have opposite null hypotheses: the ADF test checks for stationarity by testing for the presence of a unit root (H_0 : non-stationary), while the KPSS test assumes stationarity (H_0 : stationary):

1. Start by importing both the `adfuller` and `kpss` functions from `statsmodels`:

```
from statsmodels.tsa.stattools import adfuller, kpss
```

2. To simplify the interpretation of the test results, create three helper functions:
 - `adf_results`: Runs the ADF test and processes its output
 - `kpss_results`: Runs the KPSS test and processes its output
 - `format_test_results`: Formats the test outputs into a clean pandas Series for easy comparison

Here is the code for implementing these helper functions:

```
def format_test_results(output, critical_values, decision, test):
    """Format the test results into a pandas Series."""
    output_dict = {
        'Test Statistic': output[0],
        'p-value': output[1],
        'Number of lags': output[2],
        'Decision': decision
    }
    for key, value in critical_values.items():
        output_dict[f"Critical Value ({key})"] = value
    return pd.Series(output_dict, name=test)

def adf_results(data):
    """Perform ADF test and format results."""
    output = adfuller(data)
    decision = "Stationary" if output[1] < 0.05 else "Non-Stationary"
    return format_test_results(output, output[4], decision, test='ADF')

def kpss_results(data):
    """Perform KPSS test and format results."""
    output = kpss(data)
    decision = "Stationary" if output[1] >= 0.05 else "Non-Stationary"
    return format_test_results(output, output[3], decision, test='KPSS')
```

3. Apply both statistical tests (**KPSS** and **ADF**) on the CO2 dataset using the defined functions:

```
adf_output = adf_results(co2_df)
kpss_output = kpss_results(co2_df)
```

4. Combine the outputs of both tests into a single pandas DataFrame for easier comparison using `pd.concat`:

```
pd.concat([adf_output, kpss_output], axis=1)
```

This should produce a DataFrame similar to the following:

	ADF	KPSS
Test Statistic	0.046051	7.649196
p-value	0.962179	0.01
Number of lags	27	29
Decision	Non-Stationary	Non-Stationary
Critical Value (1%)	-3.433252	0.739
Critical Value (5%)	-2.862822	0.463
Critical Value (10%)	-2.567452	0.347
Critical Value (2.5%)	NaN	0.574

Figure 8.12: Result output from the ADF and KPSS unit root tests

Interpret the results:

- **ADF test:** The p-value is 0.96, which is greater than 0.05. This means you fail to reject the null hypothesis, and therefore, the time series is **non-stationary**.
- **KPSS test:** The p-value is 0.01, which is less than 0.05. This means you reject the null hypothesis, and therefore, the time series is **non-stationary**.

Both tests confirm that the CO2 dataset is non-stationary.

Next, you will explore eight techniques for transforming a non-stationary time series into a stationary one. These techniques include differencing, decomposition, and other transformations.

Often, stationarity can be achieved by removing trend (de-trending) and seasonality effects. For each transformation, you will run the stationarity tests (ADF and KPSS) and visually compare the results between the different techniques.

To simplify interpretation and comparison of multiple transformations, this chapter uses a helper function called `plot_comparison` defined in `utils.py` (see Technical Requirements). This function:

- Loops through a list of transformed series (eight methods)
- Tests for stationarity using the `kpss_results` and `adf_results` functions created earlier
- Plots the transformed series in a 4x2 grid, including a rolling mean (52-week window) for visual evaluation

The function takes a `plot_type` parameter, so you can explore a line chart or a histogram. Start by creating the `plot_comparison` function.

Let's apply the different techniques for making the time series stationary on the CO2 dataset. Then, combine the methods into a Python list before passing the list to the `plot_comparison` function:

1. **First-order differencing:** Also known as **de-trending**, which is calculated by subtracting an observation at time t from the previous observation at time $t-1$ ($y_t - y_{t-1}$). In pandas, this can be done using the `.diff()` function, which defaults to `period=1`. Note that the differenced data will contain one less data point (row) than the original data, hence the use of the `.dropna()` method:

```
first_ord_diff = co2_df.diff().dropna()
```

2. **Second-order differencing:** This is useful when first-order differencing does not fully remove non-stationarity, such as in cases with strong seasonality of a complex trend. This is essentially applying first-order differencing twice – seasonal differencing followed by de-trending:

```
second_ord_diff = co2_df.diff(52).diff().dropna()
```

3. **Seasonal differencing** involves subtracting observations that are separated by a fixed seasonal period. For weekly data with yearly seasonality, this means differencing observations that are 52 weeks apart. This technique removes repeating seasonal patterns from the time series:

```
deseasonalize = co2_df.diff(52).dropna()
```

4. **Subtracting a moving average** from the time series for de-trending. You will calculate a rolling mean (`rolling_mean`), with a 52-week window and then subtract from `co2_df`:

```
rolling_mean = co2_df.rolling(window=52).mean()
rolling_mean = co2_df - rolling_mean
```

5. **Logarithmic (log) transformation** using `np.log()` is a common technique to stabilize the variance in a time series, and sometimes enough to make the time series stationary. However, log transformations alone may not make the series stationary if trends or seasonality remain. Simply, all it does is replace each observation with its log value:

```
log_transform = np.log(co2_df)
```

6. **Decomposition using** `seasonal_decompose` to extract residuals after removing trend and seasonality components. From *Figure 8.1*, we determined that it is additive, which is the default parameter in `seasonal_decompose`, so you do not need to make any changes here:

```
decomp = seasonal_decompose(co2_df)
seasonal_decomp = decomp.resid
```

7. **Decomposition using STL** as another decomposition method:

```
co2_stl = STL(co2_df, seasonal=13, robust=True).fit()
stl_decomp = co2_stl.resid
```

8. Using the Hodrick-Prescott filter to remove the long-term trend component with `hp_filter`:

```
hp_cyclic, hp_trend = hpfilter(co2_df)
```

9. Combine all transformed series into a Python list and pass it to the `plot_comparison` function:

```
methods = [first_ord_diff, second_ord_diff,
           deseasonalize, rolling_mean,
           log_transform, seasonal_decomp,
           stl_decomp, hp_cyclic]

plot = plot_comparison(methods, kpss_results,
                      adf_results, plot_type='line')
```

This will generate an 8-panel grid (4 rows x 2 columns), each showing the transformed series and its rolling average:

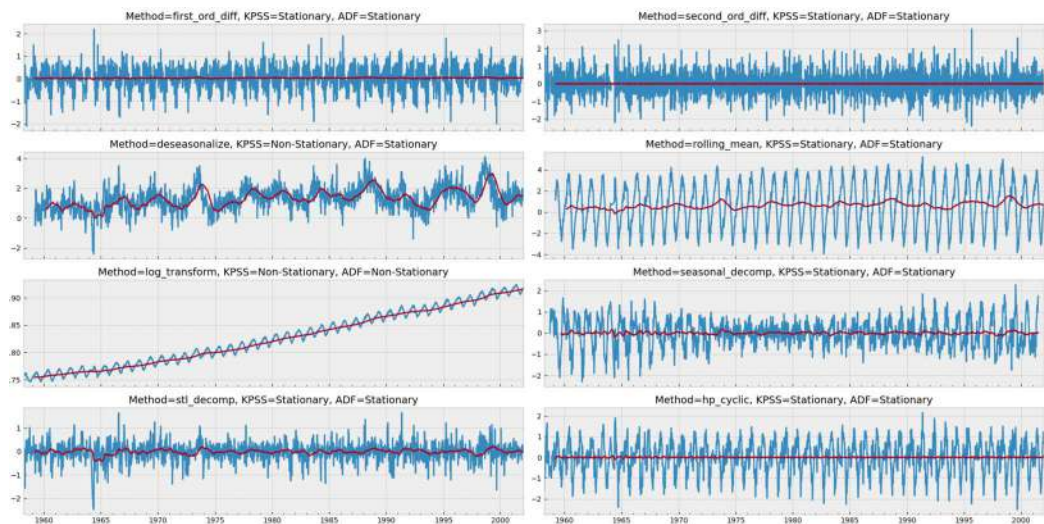


Figure 8.13: Plotting the different methods to make the CO2 time series stationary

The center line in each subplot represents the rolling mean (calculated using a 52-week window) of the transformed time series. For a stationary time series, this rolling mean should remain relatively constant over time, resembling a straight horizontal line. Observing this line helps assess whether the transformations successfully achieved stationarity by stabilizing the mean and removing trend and seasonality.

Here are some key observations from *Figure 8.13*:

- **First-order differencing** effectively removed the trend, making the time series stationary. This is evident from the ADF and KPSS test results, as well as the stable rolling mean (center line), which shows no fluctuations over time.
- **Second-order differencing** also achieved stationarity; however, it may over-difference the series if the first-order transformation already achieved stationarity. Over-differenced data can lead to less accurate models, as supported by various studies.

- The **log transformation** helps stabilize the variance but does not address the trend or seasonality. As a result, the transformed time series remains non-stationary, as indicated by the fluctuating rolling mean and test results.
- Both the `STL` and `seasonal_decompose` methods successfully removed trend and seasonality, leaving residuals that appear stationary. This is supported by the ADF and KPSS outputs, and by the stable rolling mean (center line), which shows no significant deviations over time.
- The Hodrick-Prescott filter separated the long-term trend component, leaving the remaining cyclical component that seems to be stationary. This is consistent with the ADF and KPSS test results, and with the stable rolling mean (center line).

How it works...

Stationarity is a fundamental concept in time series analysis and forecasting across various domains. A time series is considered weakly stationary if its mean, variance, and autocovariance remain constant over time. Many classical time series methods, such as ARMA models, rely on this assumption. You can learn more about statistical methods for time series forecasting in *Chapter 9*.

When the time series is stationary, it implies a stable mean. While values may deviate above or below this mean, a stationary series tends to revert to this average over time. This property, known as **mean reversion**, forms the foundation for various trading strategies, including the mean reversion trading strategy.

The `statsmodels` library offers several stationarity tests, such as the `adfuller` and `kpss` functions. Both are considered **unit root tests** and are used to determine whether differencing or other transformation strategies are needed to make the time series stationary. Remember, ADF and KPSS tests are based on different null hypotheses and thus the p-value that you use to reject (or fail to reject) the null hypothesis will be interpreted differently between the two.

In *Figure 8.12*, there is additional information returned by the tests. This includes the following:

- The **Test Statistic** ADF value of 0.046 is greater than all critical value thresholds at any significant level (e.g., 1% at -3.433, 5% at -2.863, or 10% at -2.567). This suggests that we **fail to reject the null hypothesis** of a unit root, indicating that the time series is non-stationary. On the other hand, for KPSS, a **Test Statistic** value of 7.469 exceeds all critical value thresholds at standard significant levels (e.g., 1% at 0.7939, 2.5% at 0.574, 5% at 0.463, and 10% at 0.347). This leads us to **reject the null hypothesis** of stationarity and suggest the time series is non-stationary.
- The **p-value** result is associated with the test statistic. Generally, you can reject the null hypothesis if the p-value is less than 0.05 (5%). When using ADF, KPSS, or other stationarity tests, make sure to understand the null hypothesis to accurately interpret the results. For example, in ADF, a p-value < 0.05 indicates stationarity, while a p-value ≥ 0.05 suggests non-stationarity. While in KPSS, a p-value < 0.05 suggests non-stationarity, while a p-value ≥ 0.05 indicates stationarity.
- **Number of lags** represents the number of lags used in the autoregressive process in the test (ADF and KPSS). ADF used 27 lags, while KPSS used 29 lags. Since our CO2 data is weekly, one lag represents one week.

- For `adfuller`, the optimal lag length is determined by minimizing **information criteria** through the `autolag` parameter. The default is `autolag="aic"` which minimizes the **Akaike Information Criterion**. Other acceptable values for `autolag` include `"bic"` for minimizing the **Bayesian Information Criterion** and `"t-stat"`, which selects lags based on significance testing.

To achieve stationarity, you explored different techniques to transform a non-stationary time series into a stationary one. These include de-trending and differencing methods, such as first-order differencing and decomposition to remove the effect of the trend. De-trending stabilizes the mean of the time series, and in some cases, may be all that is required to make it stationary. By de-trending your data, you are essentially removing long-term trends (upward or downward movements) that could overshadow underlying patterns. This allows you to focus on those less obvious patterns and build models that better capture them. An example of this was the first-order differencing approach.

However, in the presence of seasonal patterns, it is also necessary to remove the seasonal effect. This can be achieved through seasonal differencing or seasonal decomposition. Seasonal differencing involves differencing observations separated by the seasonal period. Often, this is combined with first-order differencing, resulting in second-order differencing, also referred to as twice-differencing or differencing twice. This approach is applied when first-order differencing alone is insufficient to make the time series stationary.

The key is to use the minimum amount of differencing needed to achieve stationarity while avoiding over-differencing, as this can lead to less accurate models. Typically, you will rarely need to go beyond differencing twice.

There's more...

In the introduction section of this recipe, we mentioned that both ADF and KPSS utilize **Ordinary Least Squares (OLS)** regression. More specifically, the ADF test uses OLS to estimate the coefficients of the regression model. To access the OLS results for ADF, you can use the `regresults=True` or `store=True` parameters (setting `regresults=True` automatically sets `store=True`).

```
adf_result = adfuller(first_ord_diff, regresults=True)
```

The `adfuller()` function returns a tuple containing the test results. The regression summary will be appended as the last item in the tuple.

There should be four items in the tuple:

1. `adf_result[0]`: The test statistics (**t-statistic**)
2. `adf_result[1]`: The **p-value**
3. `adf_result[2]`: The **critical values** for 1%, 5%, and 10% intervals
4. `adf_result[3]` or `adf_result[-1]`: A `ResultStore` object that contains detailed OLS regression results

To view the OLS regression summary, you can access the `ResultStore` object and call its `.resols.summary()` method:

```
adf_result[-1].resols.summary()
```

The `.summary()` method provides detailed regression results, including coefficients, standard errors, t-statistics, and p-values for all terms in the model.

See also

- To learn more about **stationarity** and **detrending**, visit the official statsmodels page here: https://www.statsmodels.org/dev/examples/notebooks/generated/stationarity_detrending_adf_kpss.html

Applying power transformations

Time series data often contains critical information embedded within its structure, requiring careful analysis to determine the best modeling approach. For instance, you have explored time series decomposition, analyzed the effects of trend and seasonality, and tested for stationarity. In the previous recipe, *Detecting time series stationarity*, you learned how to transform non-stationary data into stationary data through techniques such as detrending, which helps stabilize the mean over time.

When working with time series models, you often need to validate additional assumptions against the observed dataset or residuals. Two critical assumptions to test for are **homoskedasticity** (or homoscedasticity) and **normality**.

- **Homoskedasticity:** This means the variance remains stable (constant) over time. More specifically, it is the variance of the residuals. When the variance changes over time instead of staying constant, we call it **heteroskedasticity** (or heteroscedasticity).
- **Normality:** This tests whether the observations follow a normal (Gaussian) distribution. During the model diagnostics stage, you'll particularly want to check the normality of the residuals.

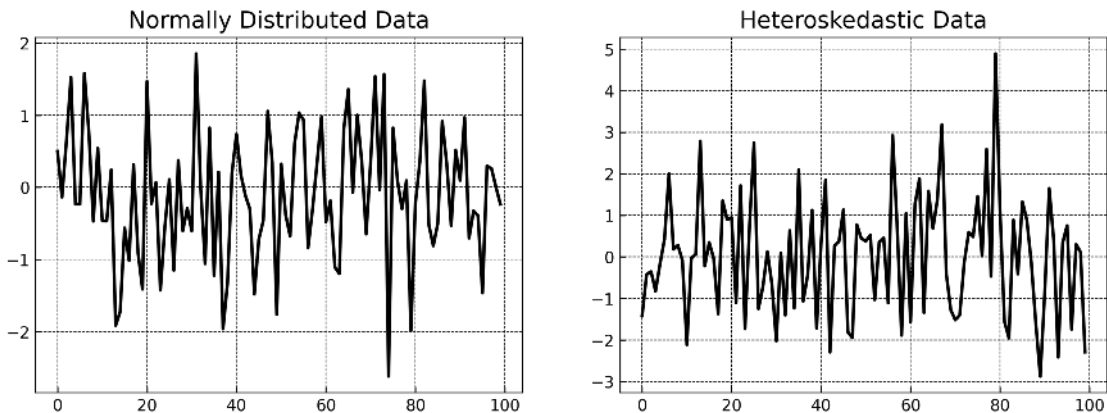


Figure 8.14: Illustration of normally distributed data and heteroskedastic data

It is critical to be aware of the assumptions made by specific models or techniques so you can determine which test to use and against which dataset. If you do not do this, you may end up with a flawed model or an outcome that may be overly optimistic or overly pessimistic.

Power transformations can help address issues such as non-normality and heteroskedasticity in time series data. In this recipe, you will learn about a commonly used power transformation, **Box-Cox transformation**, used to stabilize the variance (homoskedasticity) and achieve a more normal distribution.

Box-Cox transformation takes the following form:

$$T(x) = \begin{cases} \frac{(x^\lambda - 1)}{\lambda} & \text{if } \lambda \neq 0 \\ \ln(x) & \text{if } \lambda = 0 \end{cases}$$

Figure 8.15: Box-Cox transformation

Box-Cox transformation is controlled by a parameter called **lambda** (λ), which determines the type of transformation applied. For example, when lambda is equal to zero ($\lambda = 0$), the transformation becomes logarithmic; otherwise ($\lambda \neq 0$), a power transformation is applied.

The SciPy library provides the `boxcox` function to apply Box-Cox transformations. You'll notice that the parameter name in the function is `lmbda`, not `lambda`. This naming choice is done to avoid conflicting with Python's built-in `lambda` keyword (used for defining anonymous functions).

This `lmbda` parameter in the function specifies the value of λ . For example, if `lmbda=0`, the function will apply the natural log transformation. If the `lmbda` parameter is set to `None`, the function will automatically find the optimal lambda (λ) value that maximizes the log-likelihood for your data.



Note

Box-Cox transformation requires all data values to be positive. If zeros or negative values are present, a constant must be added before applying the transformation.

Getting ready

Refer to the *Technical requirements* section on how to download the necessary Jupyter notebooks and datasets. This recipe uses the **Air Passengers** dataset, which was previously loaded as a pandas DataFrame in the *Technical requirements* section of this chapter.

You will be using the **SciPy** and **statsmodels** libraries.

For pip installation, use the following command:

```
> pip install scipy
```

For conda installation, use the following command:

```
> conda install scipy
```

Import the following libraries, which will be used throughout the recipe:

```
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

To make the plots more readable, set a fixed figure size of 14 inches (width) by 6 inches (height), using the following command:

```
plt.rcParams["figure.figsize"] = (14,6)
```

This will globally change the default size of all subsequent figures.

How to do it...

In this recipe, you will build upon what you learned about stationarity testing in the *Detecting time series stationarity* recipe, extending your diagnostic toolkit to examine two additional crucial assumptions: **normality** and **homoskedasticity**.



While stationarity forms the foundation for many statistical time series models, examining residuals for normality and **homoskedasticity** provides vital insights into your model's validity and reliability.

Visual inspection often provides your first clue about normality and **homoskedasticity**. For example, a **histogram** or a **Q-Q plot** can reveal departures from normality, while scatter plots of residuals against fitted values help identify heteroskedasticity patterns. This recipe aims to teach you how to implement these diagnostic tests programmatically in Python.

For testing homoskedasticity, you will learn how to implement the **White test** and the **Breusch-Pagan Lagrange test**.

For normality diagnostics, you will work with three powerful tests: the **Shapiro-Wilk**, **D'Agostino-Pearson**, and **Kolmogorov-Smirnov** statistical tests. Overall, Shapiro-Wilk tends to perform best and handles a broader set of cases.

Testing normality

Testing for normality in time series data is crucial because many statistical methods and forecasting models assume that the underlying data or residuals follow a normal distribution. Understanding whether your data meets this assumption helps you choose appropriate modelling techniques and validates your model's statistical inferences.

The `statsmodels` library and the `SciPy` library have overlapping implementations. For example, the Kolmogorov-Smirnov test is implemented as `kstest` in `SciPy` (for testing against any distribution) and `kstest_normal` in `statsmodels` (specifically for testing normality). In `SciPy`, the D'Agostino-Pearson test is implemented as `normaltest` and the Shapiro-Wilk test as `shapiro`:

1. Start by importing the normality tests provided by the `SciPy` and `statsmodels` libraries:

```
from scipy.stats import shapiro, normaltest
from statsmodels.stats.diagnostic import kstest_normal, normal_ad
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

2. Normality diagnostics are statistical tests that evaluate whether data follows a normal distribution based on a null hypothesis. All the tests implemented here evaluate whether the data is normally distributed, though the specific formulation of the null hypothesis may vary slightly between tests (e.g., based on skewness, kurtosis, or overall fit). Despite these differences, they all allow us to use a consistent evaluation criterion: to determine whether the time series is normally distributed, comparing the test's p-value against a significance level (typically 0.05). If the p-value falls below this threshold, you reject the null hypothesis and conclude that your time series data is not normally distributed. To simplify the interpretation of test results, let's create a function, `is_normal()`, that returns either "Normal" or "Not Normal" based on the p-value:

```
def is_normal(test, p_level=0.05):
    """
    Determines if data is normally distributed based on test statistics.

    Parameters:
        test: Tuple of (statistic, p-value) from normality test
        p_level: Significance level (default: 0.05)

    Returns:
        str: 'Normal' if p-value > p_level, 'Not Normal' otherwise
    """
    stat, pval = test
    return 'Normal' if pval > 0.05 else 'Not Normal'
```

Run each test to check the results:

```
print("Shapiro-Wilk Test:", is_normal(shapiro(airp_df)))
print("D'Agostino-Pearson Test:", is_normal(normaltest(airp_df)))
print("Anderson-Darling Test:", is_normal(normal_ad(airp_df)))
print("Kolmogorov-Smirnov Test:", is_normal(kstest_normal(airp_df)))

>>
Shapiro-Wilk Test: Not Normal
D'Agostino-Pearson Test: Not Normal
Anderson-Darling Test: Not Normal
Kolmogorov-Smirnov Test: Not Normal
```

As expected, the output from the tests confirms that the time series data does not come from a normal distribution. You do not need to run that many tests. The shapiro test, for example, is a very common and popular test that you can rely on. Generally, as with any statistical test, you need to read the documentation regarding the implementation to gain an understanding of the test. More specifically, you will need to understand the null hypothesis behind the test to determine whether you can reject or fail to reject the null hypothesis.

3. A key use case for normality tests is being part of model evaluation and diagnostics. Typically, you would evaluate the **residuals** (defined as the difference between actual and predicted values) if they follow a normal distribution. In *Chapter 9*, you will explore building forecasting models using autoregressive and moving average models. You will run the following SARIMAX model to demonstrate how you can use a normality test against the residuals of a model:

```
model = SARIMAX(airp_df,
                trend="c",
                order=(2,1,2), # non-seasonal part (p,d,q)
                seasonal_order=(0,1,1,12), # seasonal part (P,D,Q,s)
                enforce_stationarity=False
                ).fit()
```

You can obtain the residuals from the model using `model.resid`. These are the raw residuals, representing the difference between observed values and predicted values, and they assume a constant variance (**homoskedasticity**). However, a better option is to use **standardized residuals**. These are calculated by dividing the residuals by their predicted standard deviation, accounting for time-varying variance in the residuals (**heteroskedasticity**). Think of standardized residuals as raw residuals that have been “adjusted” to account for changing variance patterns in your time series. Thus, the standardized residuals provide a more accurate representation of the residuals when variance changes over time.

4. Capture the standardized residuals from the SARIMAX model:

```
standardized_residuais = (
    model.filter_results.standardized_forecasts_error[0]
)
```

5. To evaluate the residuals for normality, you can start by running the Shapiro-Wilk test. For consistency, you will also run the same tests used earlier:

```
print("Shapiro-Wilk Test:",
      is_normal(shapiro(standardized_residuais)))
print("D'Agostino-Pearson Test:",
      is_normal(normaltest(standardized_residuais)))
print("Anderson-Darling Test:",
      is_normal(normal_ad(standardized_residuais)))
print("Kolmogorov-Smirnov Test:",
      is_normal(kstest_normal(standardized_residuais)))
>>
Shapiro-Wilk Test: Not Normal
D'Agostino-Pearson Test: Not Normal
Anderson-Darling Test: Not Normal
Kolmogorov-Smirnov Test: Not Normal
```

The output indicates the residuals are not normally distributed. This fact, residuals not being normally distributed, is not enough to determine the model's validity or potential improvements. But taken into context with the other tests, it should help you determine how good your model is. This is a topic you will explore further in the next chapter.

Testing homoskedasticity

You will be testing for the stability of the variance against the model's residuals. This will be the same SARIMAX model from the previous normality test:

1. Let's start by importing the method needed for this recipe:

```
from statsmodels.stats.api import (het_breuschpagan, het_white)
```

2. You will perform a homoskedasticity test on the model's residuals. As stated earlier regarding statistical tests, it is vital to understand the hypothesis behind these tests. The null hypothesis states that *the data is homoskedastic* for the two tests. For example, you would reject the null hypothesis if the p-value is less than 0.05, making the time series heteroskedastic.

Let's create a small function, calling it `het_test`, that takes in a model and the test function and returns either *Heteroskedastic* or *Homoskedastic* based on the p-value to determine whether the null hypothesis is accepted or rejected:

```
def het_test(model, test=het_breuschpagan, p_level=0.05):
    """
    Determines if residuals exhibit heteroskedasticity
    based on test statistics.

    Returns:
        str: 'Homoskedastic' if p-value > p_level,
            'Heteroskedastic' otherwise
    """
    lm, lm_pvalue, fvalue, f_pvalue = test(
        model.filter_results.standardized_forecasts_error[0],
        add_constant(model.fittedvalues)
    )

    return "Heteroskedastic" if f_pvalue < p_level else "Homoskedastic"
```


3. Start with the **Breusch-Pagan Lagrange Multiplier** test to diagnose the residuals for heteroskedasticity. In `statsmodels`, you will use the `het_breuschpagan` function, which accepts a *resid* parameter for the model's residual, and `exog_het`, where you provide the **explanatory** variables that are hypothesized to influence the variance of the residuals. You will use the fitted values of the SARIMAX model as the explanatory variables after adding a constant term auxiliary (additional) regression.

```
print("Breusch-Pagan Test:", het_test(model, test=het_breuschpagan))
>>
Breusch-Pagan Test: Heteroskedastic
```

This result indicates that the residuals are heteroskedastic (the variance is not stable).

4. A very similar test is White's Lagrange Multiplier test. In `statsmodels`, you will use the `het_white` function, which has the same two parameters that you used in `het_breuschpagan`:

```
print("White Test:", het_test(model, test=het_white))
>>
White Test: Heteroskedastic
```

Both **Breusch-Pagan** and **White's** tests indicate that the residuals of the SARIMAX model exhibit unstable variance (heteroskedasticity). These tests use an auxiliary regression where the squared residuals from the primary model are regressed on explanatory variables to detect systematic relationships. In these tests, explanatory variables often include a constant term and regressors or fitted values from the original model. This auxiliary regression (also called secondary or diagnostic regression) examines whether there is a systematic relationship between residual variance and these variables (e.g., fitted values). If such a relationship exists, it suggests that forecasting error's variance changes with prediction size or other factors, indicating heteroskedasticity.

Keep in mind that both normality and homoskedasticity are some of the tests you may need to conduct on the residuals as you diagnose your model. Another essential test is testing for **autocorrelation**, which is discussed in the *Testing for autocorrelation* recipe.

Applying a Box-Cox transform

Box-Cox transformation can be a useful tool, and it's good to be familiar with it. Box-Cox transforms attempt to make a non-normally distributed dataset more normal and often help stabilize variance (more homoskedastic). However, it doesn't guarantee perfect normality, especially with complex time series data. To gain a better understanding of the effect of Box-Cox transformation, you will use the Air Passengers dataset, which contains both trend and seasonality:

1. Start by importing the `boxcox` function from the SciPy library:

```
from scipy.stats import boxcox
```

- Recall, in the formula in *Figure 8.15*, there is a lambda parameter, used to determine which transformation to apply (*logarithm* or *power transform*). Use the `boxcox` function with the default parameter value for `lambda`, which is `None`. Just provide the dataset to satisfy the required `x` parameter:

```
# Box-Cox transformation requires positive values
if (airp_df['passengers'] <= 0).any():
    raise ValueError("Box-Cox transformation requires
                      all positive values")
xt, lambda = boxcox(airp_df['passengers'])
print(lambda)
>>
0.14802254856840585

# Convert xt into a pandas Series
xts = pd.Series(xt, index=airp_df.index)
```

By not providing a value to `lambda` and keeping it at `None`, the function will find the optimal lambda (λ) value. From the introduction of this recipe, you'll remember that lambda is spelled `lambda` in the `boxcox` implementation. The function returns two values, captured by `xt` containing the transformed data and `lambda` containing the optimal lambda value found.

The lambda value produced by `boxcox()` determines the specific transformation applied to your data. When lambda is 0, it corresponds to a log transformation, while other values represent different power transformations. The function uses maximum likelihood to search for the optimal lambda that makes your data as close to normal as possible while stabilizing the variance. This automatic optimization is why we often let `boxcox()` determine lambda rather than specifying it ourselves. When you do provide your own lambda value, the `boxcox` function will return only the transformed array without the lambda value, since it's not performing the optimization step.

A histogram can visually show the impact of the transformation:

```
fig, ax = plt.subplots(1, 2, figsize=(16,5))
airp_df.hist(ax=ax[0])
ax[0].set_title('Original Time Series')
xts.hist(ax=ax[1])
ax[1].set_title('Box-Cox Transformed');
```

This should produce the following two plots:

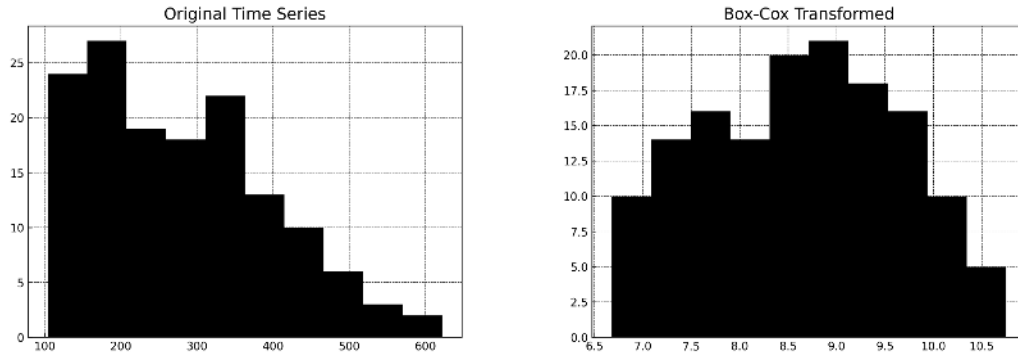


Figure 8.16: Box-Cox transformation and effect on the distribution

The second histogram shows that the data was transformed, and the overall distribution changed (it looks a bit closer to a normal distribution).

Plot both datasets to compare before and after the transformation:

```
fig, ax = plt.subplots(1, 2, figsize=(16,5))
airp_df.plot(ax=ax[0])
ax[0].set_title('Original Time Series')
ax[0].grid(True)
xts.plot(ax=ax[1])
ax[1].set_title('Box-Cox Transformed')
ax[1].grid(True);
```

This should produce the following two plots:

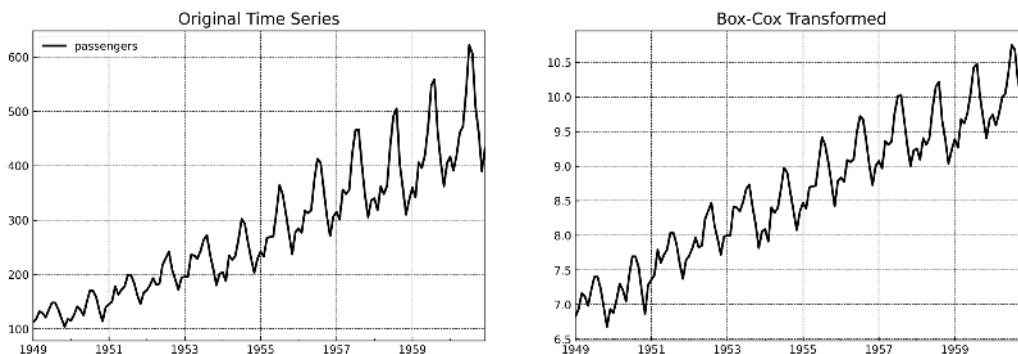


Figure 8.17: Box-Cox transformation and overall effect on time series data

Notice how the seasonal effect on the transformed dataset looks more stable than before.

- Run a SARIMAX model (the same parameters as before) on the transformed dataset and then examine the residuals:

```
model_bx = SARIMAX(xts,
                    trend="c",
                    order=(2,1,2), # non-seasonal part (p,d,q)
                    seasonal_order=(0,1,1,12), # seasonal part (P,D,Q,s)
                    enforce_stationarity=False
                    ).fit()
```

- Compare the residuals to evaluate the overall effect before and after the transformation:

```
fig, ax = plt.subplots(1, 2, figsize=(16,5))
ax[0].plot(model.filter_results.standardized_forecasts_error[0])
ax[1].plot(model_bx.filter_results.standardized_forecasts_error[0])

ax[0].set_title("Standardized Forecast Errors")
ax[1].set_title("Standardized Forecast Errors after Box-Cox
Transformation");
```

This should produce the following two plots:

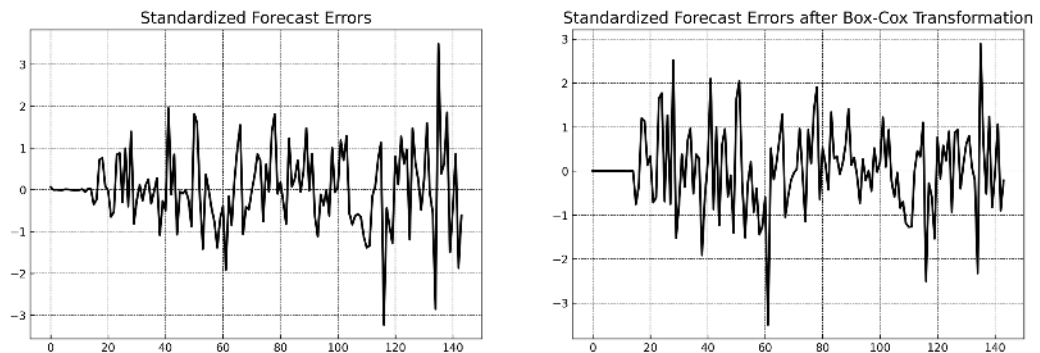


Figure 8.18: Box-Cox transformation and effect on residuals

- Re-run the normality tests on the residuals:

```
s_resid = model_bx.filter_results.standardized_forecasts_error[0]

print("Shapiro-Wilk Test:", is_normal(shapiro(s_resid)))
print("D'Agostino-Pearson Test:", is_normal(normaltest(s_resid)))
print("Anderson-Darling Test:", is_normal(normal_ad(s_resid)))
print("Kolmogorov-Smirnov Test:", is_normal(kstest_normal(s_resid)))
>>
Shapiro-Wilk Test: Not Normal
D'Agostino-Pearson Test: Not Normal
Anderson-Darling Test: Not Normal
Kolmogorov-Smirnov Test: Normal
```

Getting mixed results from normality tests is quite common in time series analysis. Each test approaches normality assessment differently – **Shapiro-Wilk** excels at detecting subtle departures from normality, **D’Agostino-Pearson** focuses on skewness and kurtosis, **Anderson-Darling** emphasizes tail behavior, while **Kolmogorov-Smirnov** primarily examines the center of the distribution. These differences in sensitivity and focus areas explain why we see varying results.

In this case, while the **Kolmogorov-Smirnov** test suggests **normality**, the majority of tests indicate **non-normal** residuals. However, in time series analysis, strictly normal residuals aren’t always necessary for a model to be useful. What matters more is understanding the nature and extent of any departures from normality. For instance, if the non-normality stems from a few outlier periods in your data, you might focus on investigating those specific time points rather than overhauling your entire model. Similarly, if your residuals show only minor deviations from normality but maintain homoskedasticity and lack significant autocorrelation, your model might still provide reliable point forecasts.

The key is to interpret these normality results alongside other diagnostic measures, which we’ll explore in the next recipe (*Testing for autocorrelation*). This comprehensive approach helps you make informed decisions about model adequacy and potential improvements.

6. Re-run the homoskedasticity tests on the residuals:

```
print("White Test:", het_test(model_bx,
                             test=het_white))
print("Breusch-Pagan Test:", het_test(model_bx,
                                     test=het_breuschpagan))

>>
White Test: Homoskedastic
Breusch-Pagan Test: Homoskedastic
```

The homoskedastic results from White/Breusch-Pagan demonstrate an improvement in the stability of variance, validating the effectiveness of the Box-Cox transformation for variance stabilization. However, statistical tests provide only one perspective on our model’s performance.

A comprehensive model evaluation requires both statistical tests and visual diagnostics. The `plot_diagnostics()` method provides four essential plots that help us visually assess our model’s assumptions and performance:

```
model_bx.plot_diagnostics();
```

This generates four diagnostic plots (*Figure 8.19*):

- **Standardized residuals:** Shows the residuals over time, helping detect patterns or changes in variance
- **Histogram plus estimated density:** Compares residual distribution to normal distribution
- **Normal Q-Q:** Plots theoretical quantiles against sample quantiles to assess normality
- **Correlogram:** Displays autocorrelation in residuals at different lags

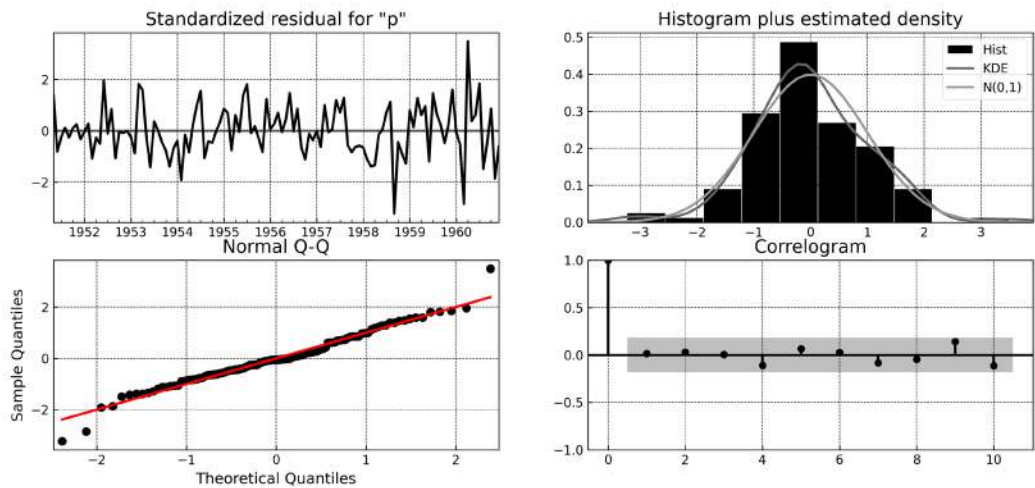


Figure 8.19: SARIMAX diagnostic plots on the Box-Cox transformed Air Passenger dataset

These diagnostic plots complement our statistical tests and provide visual insights into model performance. The standardized residuals plot reveals relatively stable variance over time, while the histogram and Q-Q plot suggest a near-normal distribution with some tail deviations. The correlogram (Autocorrelation Function or ACF plot) shows that most **autocorrelations** fall within confidence bands, indicating the model captures most temporal dependencies. You will examine ACF plots in more detail in the *Plotting ACF and PACF* recipe in *Chapter 9*.

These visuals confirm that the Box-Cox transformation has improved both normality and variance stability, though some subtle seasonal patterns may remain.

Remember that while statistical tests give us quantitative measures, these visual diagnostics often reveal nuanced patterns that numbers alone might miss, helping us make more informed decisions about potential model refinements.

How it works...

Box-Cox transformation is a versatile technique that helps achieve both normality and homoskedasticity in your data. What makes Box-Cox particularly powerful is its ability to encompass various power transformations through a single parameter called lambda (λ). By adjusting lambda, you can apply different transformations ranging from inverse square to square transformations.

The lambda (λ) parameter determines the specific transformation applied:

- $\lambda = -2$: Inverse square transformation
- $\lambda = -1$: Inverse transformation
- $\lambda = -0.5$: Inverse square root transformation
- $\lambda = 0$: Natural logarithm transformation
- $\lambda = 0.5$: Square root transformation
- $\lambda = 1$: No transformation
- $\lambda = 2$: Square transformation



The boxcox function can only be applied to positive data values.

In this recipe, you explored various statistical tests to assess both the normality and homoskedasticity of model residuals. To test for normality, you used several tests that complement each other:

- **Shapiro–Wilk test** (`shapiro`)
 - Particularly good at detecting small deviations from normality.
 - Often recommended for smaller sample sizes.
- **D’Agostino–Pearson test** (`normaltest`)
 - Assesses both skewness and kurtosis.
 - Suited for moderate to large samples.
- **Anderson–Darling test** (`normal_ad`)
 - Places more emphasis on the tail behavior of the distribution.
 - More sensitive to outliers than some other tests.
- **Kolmogorov–Smirnov test** (`kstest_normal`)
 - Compares the sample distribution to a theoretical distribution (e.g., normal).
 - Less sensitive for strict normality checks but useful for general distribution comparisons.
- **Jarque–Bera test** (`test_normality(method='jarquebera')`)
 - Evaluates whether skewness and kurtosis match those of a normal distribution
 - Particularly suited for residual diagnostics in econometric models
 - Directly accessible via the SARIMAX model:

```
jb_test = model_bx.test_normality(method='jarquebera')
print("Jarque-Bera Test:",
      "Normal" if jb_test[0][1] > 0.05 else "Not Normal" )
>>
Jarque-Bera Test: Not Normal
```

- Alternatively, you can access the function directly:

```
from statsmodels.stats.stattools import jarque_bera
```

For assessing homoskedasticity, you employed two primary tests:

- **Breusch–Pagan test** (`het_breuschpagan`)
 - Tests whether residual variance depends on predictor variables.
 - Good at detecting linear relationships in variance.
- **White test** (`het_white`)
 - Can detect both linear and non-linear variance relationships.
 - More general than Breusch–Pagan.
- **Goldfeld-Quandt test** (`test_heteroskedasticity(method='breakvar', alternative='two-sided')`)
 - Tests for heteroskedasticity by splitting the data into two subsets and comparing their variances
 - Particularly useful for detecting variance changes over time or across subsets of data
 - Directly accessible via the SARIMAX model:

```
gq_test = model_bx.test_heteroskedasticity(method='breakvar')
print("Goldfeld-Quandt Test:",
      "Homoskedastic" if gq_test[0][1]>0.05 else "Heteroskedastic" )
>>
Goldfeld-Quandt Test: Homoskedastic
```

- Alternatively, you can access the function directly:

```
from statsmodels.stats.api import het_goldfeldquandt
```

There's more...

Building on our exploration of Box-Cox transformation, SciPy provides additional specialized functions to complement `boxcox`, including: `boxcox_normmax` for optimal lambda estimation and `boxcox_normplot` for visual lambda evaluation. These tools help you make informed decisions about transformation parameters.

When you use the `boxcox` function with `lmbda=None`, it automatically finds the optimal lambda value by maximizing the log-likelihood function (MLE):

```
xt, lmbda = boxcox(airp_df['passengers'])
print(f"Optimal lambda from boxcox: {lmbda:.5f}")
>>
Optimal lambda from boxcox: 0.14802
```


The `boxcox_normmax` offers two estimation methods to find the optimal lambda value. You can choose between maximizing the Pearson correlation coefficient ('pearsonr') or the log-likelihood ('mle'). Using `method='all'` lets you compare both approaches:

```
lmbdas = boxcox_normmax(airp_df['passengers'], method='all')
print(f'Using Person Correlation (personr): {lmbdas[0]:.5f}')
print(f'Using Maximum-Likelihood (mle): {lmbdas[1]:.5f}')
>>
Using Person Correlation (personr): 0.21083
Using Maximum-Likelihood (mle): 0.14802
```

To visualize how different lambda values affect your transformation, `boxcox_normplot` creates a plot showing the **Probability Plot Correlation Coefficient** across a range of lambda values:

```
fig, ax = plt.subplots()
boxcox_normplot(airp_df['passengers'], la=-5, lb=5, plot=ax);
_, optimal_lambda = boxcox(airp_df['passengers'])
ax.axvline(optimal_lambda, color='red', linestyle='--',
            label=f'Optimal  $\lambda$  = {optimal_lambda:.4f}')
plt.legend()
plt.show()
```

This should produce the following plot where the x-axis represents the different lambda values used and the corresponding probability plot correlation coefficient on the y-axis:

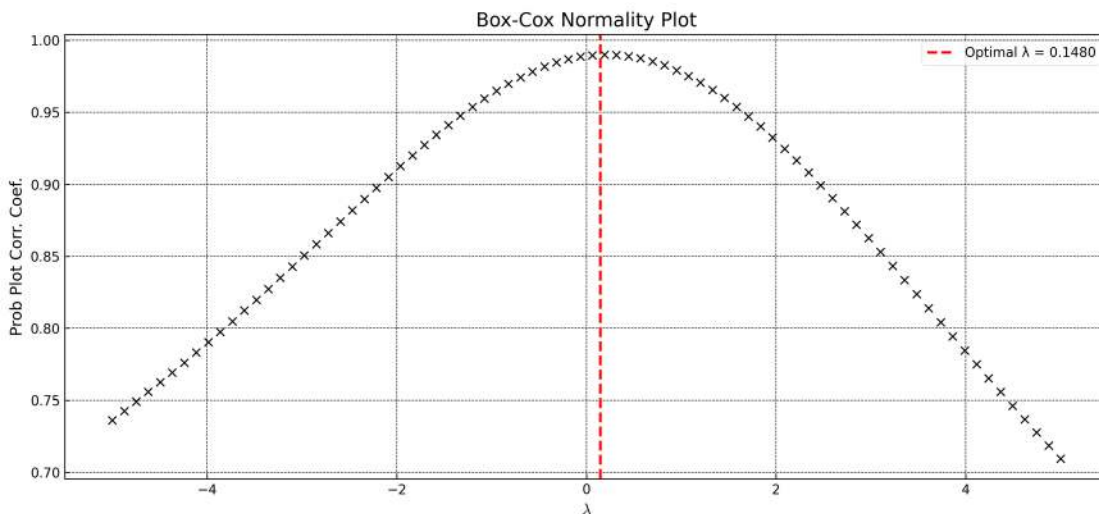


Figure 8.20: Output of `boxcox_normplot` evaluating different lambda values from -5 to 5

If you transform the data before modeling, you will need to inverse-transform your forecasts to get them back to their original scale. This can be done using the `inv_boxcox` function from SciPy. The function takes the transformed data and the lambda value used:

```
airp_restored = pd.Series(inv_boxcox(xt, lambda), index=airp_df.index)
```

If your data contains negative values, you have two main approaches:

- Add a constant to shift the distribution to become so that all data values are positive

```
# Add absolute minimum plus small buffer to ensure positivity
shift_constant = abs(min(airp_df['passengers'])) + 1
shifted_data = airp_df['passengers'] + shift_constant
transformed_data, lambda_val = boxcox(shifted_data)
```

- Use **Yeo-Johnson** transformation, an extension of Box-Cox transformation that handles zero, negative, or positive values.

Similar to Box-Cox, SciPy offers three functions for **Yeo-Johnson**: `yeojohnson`, `yeojohnson_normmax`, and `yeojohnson_normplot`. Unlike `boxcox_normmax`, `yeojohnson_normmax` only supports maximum likelihood estimation, so it won't be necessary to use it if you are using `yeojohnson` to get the optimal lambda value:

```
from scipy import stats
xt, lambda = yejohnson(airp_df['passengers'])
print(f"Optimal lambda from yejohnson: {lambda:.5f}")
>>
Optimal lambda from yejohnson: 0.14526
```

You can visualize Yeo-Johnson lambda optimization just like Box-Cox using `yeojohnson_normplot`:

```
fig, ax = plt.subplots()
yeojohnson_normplot(airp_df['passengers'], la=-5, lb=5, plot=ax);
_, optimal_lambda = yejohnson(airp_df['passengers'])
ax.axvline(optimal_lambda, color='red', linestyle='--',
            label=f'Optimal  $\lambda$  = {optimal_lambda:.4f}')
plt.legend()
plt.show()
```

Another alternative is the `PowerTransformer` class from `scikit-learn`, which provides a unified interface for both transformations: 'yeo-johnson' or 'box-cox'. If you do not have `scikit-learn` installed, you can do the following:

Install it using `pip`:

```
pip install scikit-learn
```

Once installed, you can use `PowerTransformer` as shown here:

```
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer(method='yeo-johnson')
y_hat = pt.fit_transform(airp_df)
print(pt.lambdas_)
>>
[0.14526327]
```

To perform inverse-transformation to restore the original values, you can use the `inverse_transform` method:

```
airp_restored = pt.inverse_transform(y_hat)
```

See also

- To learn more about the **boxcox** function, visit the official SciPy documentation here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html>

Testing for autocorrelation

Autocorrelation measures the relationship between a time series and a lagged (delayed) version of itself. It is similar to the concept of **Pearson correlation**, except that instead of comparing two different variables, autocorrelation examines how observations of the same time series correlate when separated by a specific lag. In essence, it captures the degree of dependence between a time series and its past values.

For example, consider daily temperature readings. Using autocorrelation, you might discover that today's temperature has a strong relationship (highly correlated) with yesterday's temperature (lag 1) but exhibits a weaker relationship with the temperature from a week ago (lag 7).

In this recipe, you will use the **Ljung-Box test** to check for autocorrelations up to a specified maximum lag and determine whether they deviate significantly from zero. *The null hypothesis for the Ljung-Box test states that the previous lags are not correlated with the current period.* In other words, it assumes there's no autocorrelation present in the data.

The `statsmodels` library implements this test through the `acorr_ljungbox` function, which requires you to specify a maximum lag value. The test then runs for all lags up to the specified maximum lag.

When applying this test to model residuals, it becomes a crucial model diagnostics tool. As you learned in the previous recipe, *Applying power transformations*, there are certain assumptions that you need to validate against the model's residuals, such as **normality** and **homoskedasticity**. Another key assumption is that residuals should not show significant autocorrelation, meaning they should be independent of each other. This indicates that the model has successfully captured all the relevant patterns in the data.

If the residuals exhibit significant autocorrelation, this suggests that the model has missed critical information and requires further refinement.

Getting ready

You can download the Jupyter notebooks and datasets needed from the GitHub repository. You will be using `acorr_ljungbox` from the `statsmodels` library.

How to do it...

You will use the Air Passengers dataset stored in the `airp_df` DataFrame:

1. Load `acorr_ljungbox` from the `statsmodels` library:

```
from statsmodels.stats.diagnostic import acorr_ljungbox
```

2. To see how the Ljung-Box test works, let's apply it to the raw Air Passenger dataset:

```
ljung_test = acorr_ljungbox(airp_df,
                             lags=12,
                             return_df=True)
ljung_test['eval'] = ljung_test['lb_pvalue'] < 0.05
print(ljung_test)
>>
```

	lb_stat	lb_pvalue	eval
1	132.141539	1.393231e-30	True
2	245.646160	4.556318e-54	True
3	342.674826	5.751088e-74	True
4	427.738684	2.817731e-91	True
5	504.796570	7.360195e-107	True
6	575.601854	4.264008e-121	True
7	643.038593	1.305463e-134	True
8	709.484498	6.496271e-148	True
9	779.591231	5.249370e-162	True
10	857.068639	1.100789e-177	True
11	944.390318	1.766396e-195	True
12	1036.481907	2.682212e-214	True

All p-values are less than 0.05, so you reject the null hypothesis of no significant autocorrelation at all lags up to lag 12 (the specified maximum lag). This indicates that significant autocorrelation exists in the original data. This is expected for time series data such as the Air Passengers data that shows strong trends and seasonal patterns.

3. Since the Air Passengers data is not stationary, apply first-order differencing to stabilize its mean and remove trends to ensure stationarity before applying the Ljung-Box test:

```
airp_df_diff = airp_df.diff().dropna()
ljung_test_diff = acorr_ljungbox(airp_df_diff,
                                  lags=12,
                                  return_df=True)
```

```

ljung_test_diff['eval'] = ljung_test_diff['lb_pvalue'] < 0.05
print(ljung_test_diff)
>>

```

	lb_stat	lb_pvalue	eval
1	13.393249	2.525316e-04	True
2	14.927657	5.734566e-04	True
3	23.549402	3.101926e-05	True
4	37.010908	1.792006e-07	True
5	38.340604	3.223072e-07	True
6	39.271899	6.330720e-07	True
7	40.572528	9.778688e-07	True
8	53.920949	7.151547e-09	True
9	59.612069	1.592385e-09	True
10	61.328178	2.030094e-09	True
11	73.902696	2.199762e-11	True
12	182.727561	1.175064e-32	True

All p-values are less than 0.05, so you reject the null hypothesis, indicating that significant autocorrelation exists in the differenced dataset.

Even after differencing, we still see significant autocorrelation (p-values < 0.05). This suggests that first-order differencing alone hasn't captured all the temporal dependencies in our data. This makes sense given that the Air Passengers data has both trend and seasonal patterns, which is why we ultimately use a **Seasonal ARIMA (SARIMAX)** model

4. A more crucial use for the Ljung-Box test is for model diagnostics against the residuals of our model. In the previous recipe, we ran a SARIMAX(2,1,2)(0,1,1,12) model after applying Box-Cox transformation. You will use the standardized residuals for the Ljung-Box test:

```

# evaluate Box-Cox transformed SARIMAX model using standardized residuals
ljung_test_resid = acorr_ljungbox(
    model_bx.filter_results.standardized_forecasts_error[0],
    return_df=True,
    lags=12
)
ljung_test_resid['eval'] = ljung_test_resid['lb_pvalue'] < 0.05
print(ljung_test_resid)
>>

```

	lb_stat	lb_pvalue	eval
1	0.618082	0.431761	False
2	0.733210	0.693083	False
3	0.949263	0.813526	False
4	2.556810	0.634493	False
5	5.172170	0.395232	False

6	6.135678	0.408164	False
7	6.718937	0.458719	False
8	6.752595	0.563546	False
9	10.007680	0.349864	False
10	10.979611	0.359109	False
11	11.017435	0.441804	False
12	11.020071	0.527199	False

All p-values are greater than 0.05, so you fail to reject the null hypothesis of no autocorrelation at all lags up to lag 12. This is exactly what we want to see in model residuals – it indicates that our SARIMAX model has successfully captured the temporal dependencies in the data. If we had seen significant autocorrelation in the residuals, it would suggest that our model was missing important patterns in the data.

The lack of autocorrelation in residuals suggests that the SARIMAX model has successfully captured all relevant patterns in the data.

How it works...

The `acorr_ljungbox` function calculates autocorrelation up to the maximum lag specified – for example:

```
acorr_ljungbox(
    model_bx.filter_results.standardized_forecasts_error[0],
    return_df=True,
    lags=12
)
```

- The `return_df` parameter specifies whether the output should be returned as a pandas DataFrame
- In the recipe, you used `lag=12` because the Air Passenger dataset is monthly, and analyzing up to 12 lags allows you to capture a full annual cycle

You first applied the Ljung-Box test on the raw data, then applied the test after differencing the data by using first-order differencing to stabilize the mean and remove trends. While differencing reduces the autocorrelation (notice the smaller `lb_stat` values), significant autocorrelation still remains (p-values < 0.05). This suggests that differencing alone isn't sufficient to capture all the temporal dependencies in our data.

Next, you examined the standardized residuals from the SARIMAX model and observed a dramatic improvement. All p-values were above 0.05, indicating no significant autocorrelation remains in the residuals. This suggests that the SARIMAX model has effectively captured the temporal patterns in the data. Additionally, the relatively stable `lb_stat` values across different lags further support this conclusion. If significant autocorrelation had remained in the residuals, it would indicate that further model refinements.

There's more...

The `acorr_ljungbox` function can return the results of the Box-Pierce test, which is a simpler version of the Ljung-Box test. The Ljung-Box test is an improvement over the Box-Pierce test, and both have the same null hypothesis:

```
ljung_test_resid = acorr_ljungbox(
    model_bx.filter_results.standardized_forecasts_error[0],
    return_df=True,
    boxpierce=True,
    lags=12)
print(ljung_test_resid)
```

```
>>
```

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
1	0.618082	0.431761	0.605382	0.436532
2	0.733210	0.693083	0.717355	0.698600
3	0.949263	0.813526	0.926009	0.819147
4	2.556810	0.634493	2.467493	0.650466
5	5.172170	0.395232	4.957459	0.421094
6	6.135678	0.408164	5.868172	0.438119
7	6.718937	0.458719	6.415477	0.492158
8	6.752595	0.563546	6.446829	0.597311
9	10.007680	0.349864	9.456668	0.396231
10	10.979611	0.359109	10.348715	0.410450
11	11.017435	0.441804	10.383170	0.496285
12	11.020071	0.527199	10.385553	0.582176

Further, the SARIMAX model gives you access to `test_serial_correlation`, which supports two methods: 'ljungbox' and 'boxpierce':

```
model_bx.test_serial_correlation('ljungbox', lags=12)
model_bx.test_serial_correlation('boxpierce', lags=12)
```

The `test_serial_correlation` will return an array including `test_statistic` and `pvalue`.

See also

- To learn more about the `acorr_ljungbox` function, visit the official documentation here: https://www.statsmodels.org/dev/generated/statsmodels.stats.diagnostic.acorr_ljungbox.html.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



9

Building Univariate Time Series Models Using Statistical Methods

In *Chapter 8*, you explored essential concepts to understand the time series process, such as decomposing time series data, detecting stationarity, applying power transformations, and testing for residual autocorrelation. These techniques will be invaluable as you dive into statistical modeling in this chapter.

When working with time series data, different methods and models can be applied, depending on whether the time series you are working with is **univariate** or **multivariate**, **seasonal** or **non-seasonal**, **stationary** or **non-stationary**, and **linear** or **nonlinear**. If you consider the assumptions you need to validate and test for – for example, stationarity and residual autocorrelation – it becomes apparent why time series data is deemed complex and challenging. Thus, when modeling such a complex system, your goal is to develop an approximation that effectively captures the critical factors of interest while maintaining practical usefulness.

These critical factors might include the underlying trend of your data, seasonal patterns that repeat at fixed intervals, sudden level shifts or structural breaks, and the degree of volatility in your observations. For example, when forecasting retail sales, capturing seasonal holiday spikes is crucial, while in financial time series, modeling volatility clusters may take precedence. The key is to identify which patterns are meaningful for your specific domain and modeling objective, instead of attempting to capture every minor fluctuation. This selective approach avoids overfitting and produces more reliable forecasts.

Popular statistical modeling methods include **exponential smoothing**, **autoregressive integrated moving average (ARIMA)**, **seasonal ARIMA (SARIMA)**, **vector autoregressive (VAR)**, and other variants of these models, such as **ARIMAX**, **SARIMAX**, **VARX**, and **VARMA**. While advanced models exist, this chapter focuses on foundational methods such as exponential smoothing and ARIMA-based approaches to ensure you build a strong understanding before tackling more complex techniques. In *Chapter 10*, you will explore additional models, such as VAR models. Many practitioners, including economists and data scientists, still use these “classical” models. Additionally, these models can be found in popular software packages such as EViews, MATLAB, Orange, KNIME, and Alteryx, as well as libraries in Python and R.

In this chapter, you will focus on how to implement these models in Python. This chapter will provide a brief introduction to the theory and math, with a primary focus on implementation. References will be provided for readers who wish to explore the underlying concepts in greater depth.

In this chapter, we will cover the following recipes:

- Plotting ACF and PACF
- Forecasting univariate time series data with exponential smoothing
- Forecasting univariate time series data with non-seasonal ARIMA
- Forecasting univariate time series data with seasonal ARIMA
- Forecasting univariate time series data with Auto_Arima

Before diving into these recipes, review the upcoming *Technical requirements* section to ensure that your environment is ready. This preparation will streamline your workflow, allowing you to focus fully on the recipes and their concepts.

Technical requirements

You can download the Jupyter notebook and necessary datasets from this book's GitHub repository:

- **Jupyter notebook:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch9/Chapter_9.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch9>

This chapter extensively uses pandas 2.3.3 (released September 29, 2025), which is the latest version as of this writing. You can use a more recent version if available.

There are three additional libraries that we will be using:

- statsmodels (0.14.5 or higher)
- Matplotlib (3.10.6 or higher)
- Scikit-learn(sklearn) (1.7.2 or higher)

You can install these libraries using pip or conda:

Here is the code when using pip:

```
pip install pandas statsmodels matplotlib scikit-learn
```

Before diving into the recipes, follow these steps to set up your environment. While we'll be using specific versions of packages in this chapter, the code should work with newer versions as well. It's a good practice to create a new Python virtual environment when installing new packages; this helps avoid package conflicts and makes your work reproducible.

You can refer to *Chapter 0 on GitHub* for detailed instructions on setting up a Python virtual environment.

Follow these steps:

1. Start by importing the required libraries that will be shared across all the recipes in this chapter:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Time series analysis tools
from statsmodels.tsa.api import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Model evaluation metrics
from statsmodels.tools.eval_measures import rmspe, rmse
from sklearn.metrics import mean_absolute_percentage_error as mape

# Utility functions
from pathlib import Path

plt.rcParams["figure.figsize"] = [12, 5]
plt.style.use('grayscale')
```

2. You will be working with two datasets throughout this chapter: **Life Expectancy from Birth** and **Monthly Milk Production**. These datasets are stored in CSV format (`life_expectancy_birth.csv` and `milk_production.csv`, respectively). Each dataset represents a unique time series process with distinct characteristics, such as trends or seasonality.

Load the datasets into a pandas DataFrame. Once loaded, you will have two DataFrames called `life` and `milk`:

```
# Define file paths for the datasets
life_file = Path('../datasets/Ch9/life_expectancy_birth.csv')
milk_file = Path('../datasets/Ch9/milk_production.csv')
life = pd.read_csv(life_file,
                   index_col='year',
                   parse_dates=True,
                   skipfooter=1)
milk = pd.read_csv(milk_file,
                   index_col='month',
                   parse_dates=True)
```

Ensure that the datasets are located in the `datasets/Ch9` folder relative to your working directory. Adjust the file paths if needed.

3. Inspect the time series visually to observe trends and seasonality. You can always come back to the plots shown in this section for reference:

```
# Create subplots for visual comparison of both time series
fig, ax = plt.subplots(2, 1, figsize=(16, 12))
life.plot(title='Annual Life Expectancy',
          legend=False, ax=ax[0])
milk.plot(title='Monthly Milk Production',
          legend=False, ax=ax[1]);
```

This should display two time series plots:

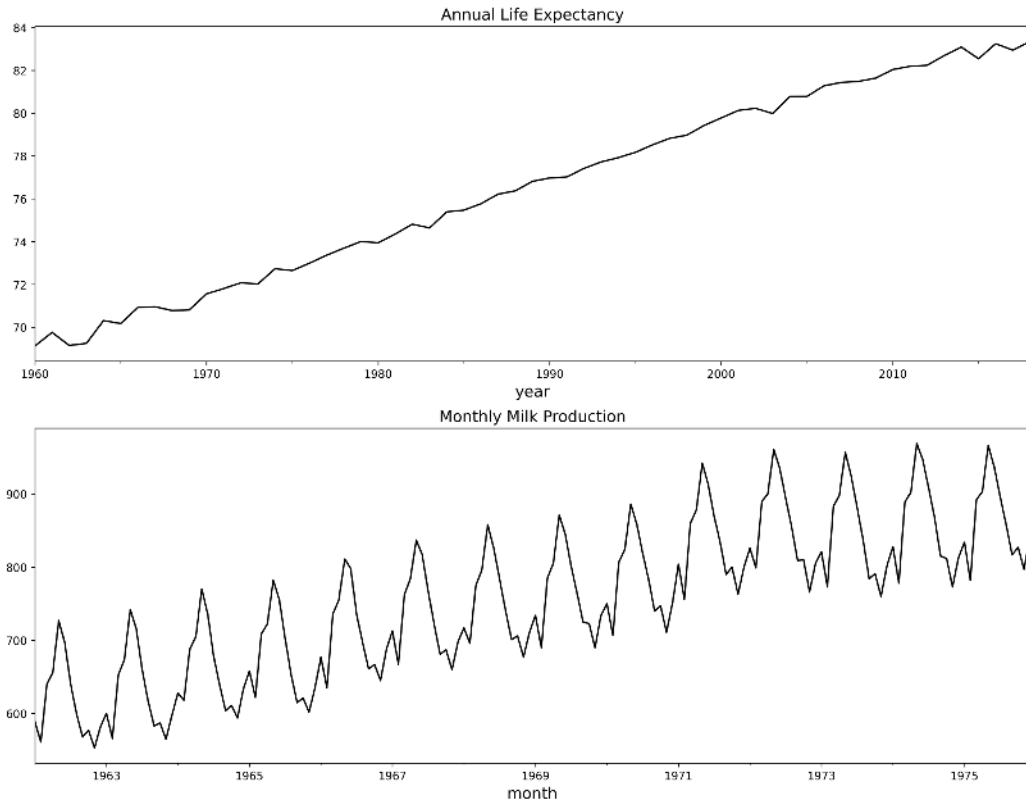


Figure 9.1: Time series plots for Annual Life Expectancy and Monthly Milk Production

Figure 9.1 reveals the distinct characteristics of our two time series plots:

- **Annual Life Expectancy:** The time series plot for the `life` DataFrame shows a positive (upward) trend with no apparent seasonality. The increase is relatively smooth and consistent over time, without any regular periodic patterns. This absence of seasonality makes sense given the annual nature of the measurements and the data being averaged out. As a result, the dataset primarily captures long-term trends rather than short-term fluctuations.

The dataset contains average annual life expectancy records at birth from 1960 to 2018 (59 years). Originally, the dataset included records of individual countries, but this chapter focuses on *world* records.

- **Monthly Milk Production:** The time series plot for the `milk` DataFrame shows a positive (upward) trend and a clear repeating seasonality, with peaks occurring every summer. This dataset contains monthly records from January 1962 to December 1975 (168 months). The seasonal magnitudes and variations over time seem to be steady (meaning the seasonal effects remain relatively stable regardless of the trend level). This stability suggests an additive seasonal model, where the seasonal component remains consistent and independent of the trend. This repeating additive seasonal pattern will become even more apparent when we perform seasonal decomposition later in the chapter.

For more details on seasonal decomposition, please refer to the *Decomposing time series data* recipe in *Chapter 8*.

4. When developing time series models, we need to assess how well they will perform on future, unseen data. To accomplish this, you'll need to split the data into two sets: a training set and a test set. The models will be trained (fitted) using the training set, and their performance will be evaluated using the testing set. It is crucial to preserve the chronological order of observations when splitting time series data to avoid data leakage from future values into the training set.

The split will allow us to make two types of forecasts:

- **In-sample forecast:** These are predictions made on the training data. While useful for model development, they don't tell us how well our model will perform in real-world scenarios. They are useful for internal validation during model development to identify whether a model has overfit or underfit the training data by comparing predictions against historical values.
- **Out-of-sample forecast:** These are predictions made on the test set, simulating real-world forecasting scenarios where we predict truly unseen data. Evaluating out-of-sample performance helps ensure that the model generalizes well.

In this chapter, you will focus on out-of-sample forecasts to evaluate and compare different models effectively.

5. To keep the modeling code focused, several helper functions used in this chapter are defined in a separate `utils.py` module. You can import them as follows:

```
from utils import split_data, check_stationarity, get_top_models_df,  
plot_forecast, combinator
```

These utilities support the full modeling workflow:

- `split_data` splits a time series into train and test sets while preserving temporal order, ensuring that evaluation always reflects true out-of-sample forecasting performance.

- `check_stationarity` applies the Augmented Dickey–Fuller test and reports whether the series appears stationary based on the p-value, helping you decide when transformations are needed before modeling.
 - `get_top_models_df` ranks candidate models by a chosen criterion (for example, AIC, RMSE, or MAPE) and returns a DataFrame with the top configurations, making it easier to review hyperparameter search results.
 - `plot_forecast` visualizes model forecasts against the original train and test segments in a single plot so you can quickly see how well the model tracks the observed data.
 - `combinator` generates Cartesian products of parameter values for grid search, providing a convenient way to enumerate model configurations for systematic tuning.
6. Use the `split_data` function to split the two DataFrames into a test and train set (start with 15% test and 85% train). You can later experiment with different split factors to observe their impact on model performance:

```
test_split = 0.15
milk_train, milk_test = split_data(milk, test_split)
life_train, life_test = split_data(life, test_split)
>>
train: 143 , test: 25
train: 50 , test: 9
```

Let's break down this example:

- The *Milk Production* dataset is split into 143 training points and 25 testing points
- The *Life Expectancy* dataset is split into 50 training points and 9 testing points

We can represent the overall flow as in *Figure 9.2*, which shows how you will be utilizing helper functions you just loaded.

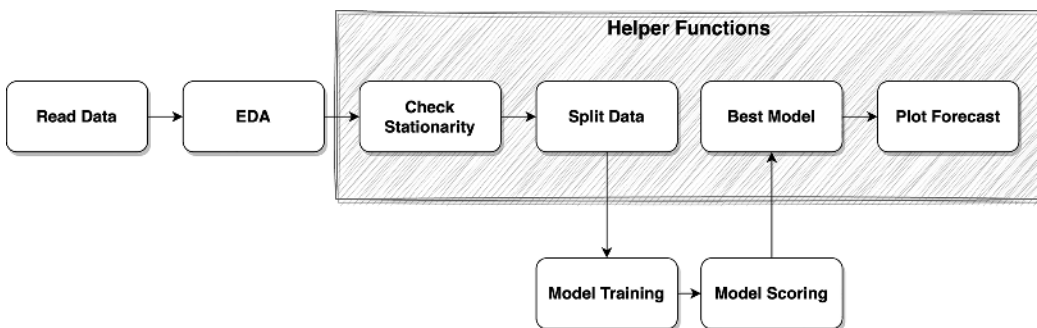


Figure 9.2: The overall process in this chapter utilizing the helper functions

Now, let's dive into the recipes.

In the first recipe, you will be introduced to the **ACF** and **PACF** plots, which are used to evaluate model fit, check stationarity, and determine the orders (parameters) for some of the models that will be used in this chapter, such as the ARIMA model.

Plotting ACF and PACF

In *Chapter 8*, in the *Testing for autocorrelation* recipe, you were introduced to the concept of autocorrelation through the Ljung-Box test. This statistical test allowed you to determine whether past values in your time series significantly influence the current values. While this test tells us whether autocorrelation exists, it doesn't show us the full picture of how these relationships behave over time. To gain deeper insight into autocorrelation patterns over time, we use visual tools, specifically the **autocorrelation function (ACF)** and **partial autocorrelation function (PACF)** plots.

These plots not only detect the presence of autocorrelation but also reveal the internal structure of your time series data, showing you how strong the relationship is between current values and each of their past values, lag by lag.

The ACF and PACF plots are essential tools for analyzing time series data before building statistical forecasting models, such as the following:

- **AR (autoregressive)**: Uses past values to predict future ones
- **MA (moving average)**: Uses past prediction errors
- **ARMA (autoregressive moving average)**: Combines both AR and MA approaches
- **ARIMA**: Adds handling non-stationary data through differencing
- **SARIMA**: Extends ARIMA by modeling seasonality in the data through seasonal terms for autoregression, differencing, and moving averages

These plots serve three crucial purposes:

- They help evaluate how well your model fits the data by examining residual autocorrelation to check whether the model has captured all the important patterns in the data.
- They provide visual cues about data stationarity – a crucial assumption for many time series models.
- They help identify significant lag orders (p for AR and q for MA components). This is critical for building ARIMA-based models.

The core idea behind these time series models is built on the assumption that you can estimate the current value of a variable, x , using its own past values or past prediction errors. Consider this example:

- In an AR model of order p or $AR(p)$, we assume that the current value, x_t , at time t can be estimated using p previous values ($x_{t-1}, x_{t-2}, \dots, x_{t-p}$). The p order determines how many lags (steps back) we need to go to capture meaningful relationships.
- In an MA model of order q or $MA(q)$, the current value, x_t , depends on past forecast errors ($e_{t-1}, e_{t-2}, \dots, e_{t-q}$). Think of the MA component as learning from past mistakes to improve future predictions.

To better understand these order (lag) values, consider a practical example using temperature forecasting:

- With $p=1$: Today's temperature depends only on yesterday's temperature
- With $p=2$: Today's temperature depends on both yesterday's and the day before
- With $p=7$: We are saying the temperature pattern repeats weekly

In essence, when $p=2$, it means we must use two previous periods (x_{t-1}, x_{t-2}) to predict x_t . Depending on the granularity of your time series data, this $p=2$ could represent different time intervals; it might be 2 hours, 2 days, 2 months, 2 quarters, or 2 years.

When we combine this AR concept with an MA component to enhance our forecasting power, we get an ARMA model, denoted as $ARMA(p, q)$. Here, p still represents how many past values we use (the AR order), and q represents how many past prediction errors we consider (the MA order). For example, consider an $ARMA(2,1)$ model:

- The $AR(2)$ component uses the two most recent observations (x_{t-1}, x_{t-2})
- The $MA(1)$ component uses the most recent prediction error (e_{t-1})

This concept will be explored further in the *Forecasting univariate time series data with ARIMA* section of this chapter.

To build these ARIMA-based models, such as $ARMA(p,q)$, you need to specify the values for the p and q orders (also known as lags). These are considered **hyperparameters** since they are supplied by you to influence the model.

The terms *parameters* and *hyperparameters* are sometimes used interchangeably. However, they have different interpretations, and you need to understand the distinction.

Parameters versus hyperparameters

When training an ARMA or an ARIMA model, it learns a set of values called **coefficients**, which are also referred to as the model's **parameters**. For example, the coefficient value for $AR(1)$ is sigma (the variance of the residuals). These parameters are estimated by the algorithm during the training process. We don't set these values ourselves; instead, the model learns them through statistical estimation. These parameters are used for making predictions.



On the other hand, **hyperparameters** are structural decisions you make before training begins. For example, in ARIMA models, you need to provide the values for the (p, d, q) orders. These are called hyperparameters because they are provided by you before training, and are not estimated by the model. These hyperparameters define the model structure and influence the model's parameters estimated during training (for example, the coefficients).

Finding the right hyperparameters is crucial for model performance. While the model can optimize its parameters automatically during training, it needs you to specify the right hyperparameters first. This is why we often use techniques such as **grid search**, in which we try different combinations of p, d , and q values to find the optimal configuration that produces the best-performing model.

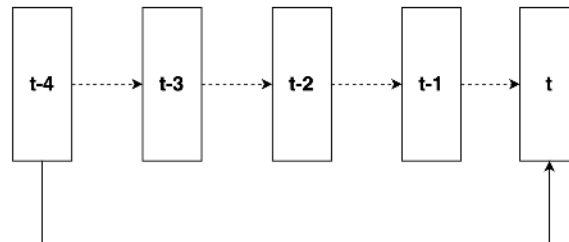
At this point, you might be asking yourself, *How do I find the significant lag values for $AR(p)$ and $MA(q)$ components?* This is where the ACF and PACF plots come into play:

- The **ACF plot** shows the correlation between the time series and its own lagged values (considering both direct relationships and those mediated by intermediate lags)

- The **PACF plot** shows the correlation between the time series and its lagged values after removing the influence of intermediate lags (considers only the direct correlation)

The following figure is an illustration comparing ACF and PACF and how $t-4$ influences t through different pathways:

ACF: Total Correlation at Lag 4 (Direct and Indirect Pathways)



PACF: Direct Correlation at Lag 4 Only

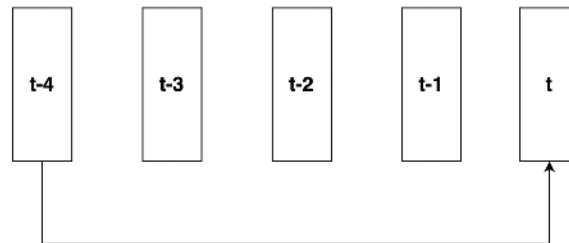


Figure 9.3: Comparing ACF and PACF at lag 4

Both PACF and ACF plots are referred to as **correlograms** since the plots represent the **correlation** statistics.

A key difference between ARMA and ARIMA, written as $ARIMA(p, d, q)$, is in the stationarity assumption. The d parameter in ARIMA is for the differencing order to make it stationary (differencing the data d times). An ARMA model assumes a stationary process, while an ARIMA model does not, since it handles it through differencing. An ARIMA model is a more generalized model since it can satisfy an ARMA model by making the differencing factor $d=0$. Hence, an $ARIMA(1,0,1)$ is $ARMA(1,1)$.

AR order versus MA order



The PACF plot is used to estimate the AR order (p), as it isolates the direct effect of each lag by removing correlations explained by shorter lags. On the other hand, the ACF plot is used to estimate the MA order (q) by showing how residual errors correlate with past observations at different lags.

Both the ACF and PACF plots show values that range from -1 to 1 on the vertical axis (y axis), while the horizontal axis (x axis) indicates the size of the lag. In ACF and PACF plots, significant lags are those whose correlation values extend beyond the shaded confidence interval, typically set at a 95% confidence level.

The statsmodels library provides two functions: `plot_acf` and `plot_pacf`. The correlation (for both ACF and PACF) at lag zero is always one (since it represents autocorrelation of the first observation on itself). Hence, both functions provide a `zero` parameter, which takes a Boolean. Therefore, to exclude the zero lag in the visualization, you can pass `zero=False` instead.

How to do it...

In this recipe, you will explore `acf_plot` and `pacf_plot` from the statsmodels library. Let's get started:

1. Start with the *Life Expectancy* data. Looking at Figure 9.1, you can observe that the series is not stationary due to the presence of a long-term upward trend. The ACF plot can also provide insight into whether the time series is stationary or not. Let's create an ACF plot to confirm this:

```
plot_acf(life, lags=25, zero=False);
```

This should produce the following:

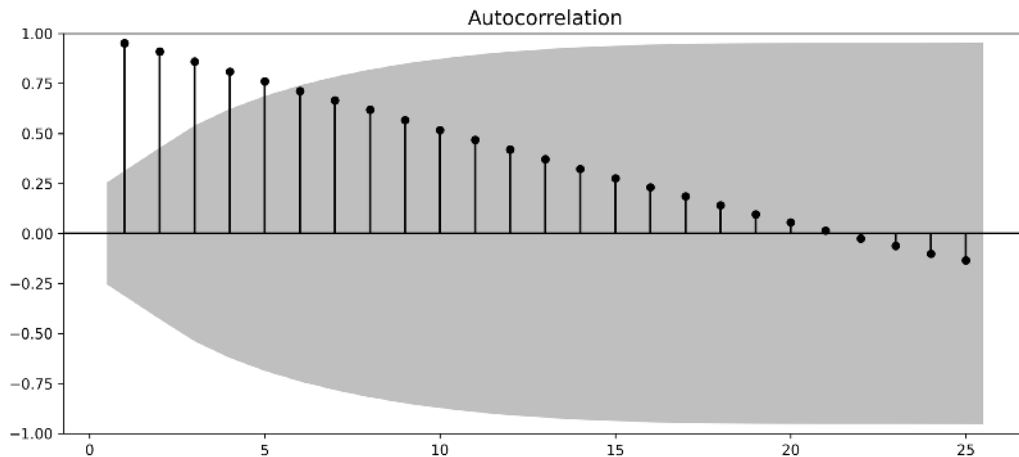


Figure 9.4: The ACF plot for the *Life Expectancy* dataset showing a slow decay, indicating non-stationarity

For stationary data, the ACF plot would typically drop close to zero relatively quickly as the lags increase (rapid decay). In contrast, for a non-stationary time series, the ACF tends to decrease slowly or remain persistently high across several lags. In the *Life Expectancy* data, the slow ACF decay occurs because the long-term trend introduces strong correlations across lags, making the time series values heavily dependent on their past.

2. Perform first-order differencing (detrending) to make the series stationary, then create an ACF and PACF plot excluding lag zero:

```
# Calculate first differences and remove any NA values
life_diff = life.diff().dropna()
# Create subplots for both ACF and PACF
fig, ax = plt.subplots(2,1, figsize=(12,8))
plot_acf(life_diff, zero=False, ax=ax[0])
plot_pacf(life_diff, zero=False, ax=ax[1]);
```

The `zero=False` parameter removes lag 0 from our plots since it always equals 1 (a value perfectly correlates with itself) and doesn't provide useful information for our analysis.

This should produce the following two plots:

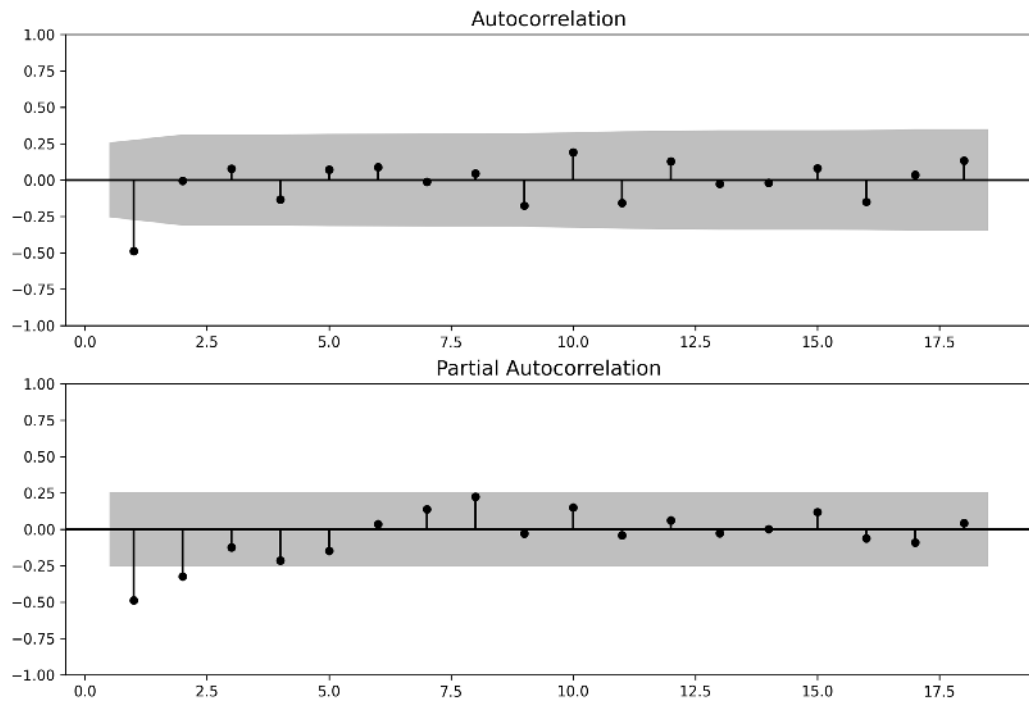


Figure 9.5: The ACF and PACF plots for the Life Expectancy data after differencing

Notice that the ACF plot in *Figure 9.5* decays much faster compared to that in *Figure 9.4*. This can indicate that our series is now stationary (this can be confirmed using a statistical test such as the ADF test from *Chapter 8*).

If you want to inspect the ACF and PACF over more lags, you can increase the `lags` parameter (for example, to 25) in the `plot_acf` and `plot_pacf` calls to display additional lag values:

```
fig, ax = plt.subplots(2,1, figsize=(12,8))
plot_acf(life_diff, lags=25, zero=False, ax=ax[0])
plot_pacf(life_diff, lags=25, zero=False, ax=ax[1]);
```

Looking at *Figure 9.5*, the ACF plot shows a significant spike at lag (order) 1. Significance is represented when a lag (vertical line) goes above or below the shaded area. The shaded area represents the confidence interval, which is set to 95% by default. In the ACF plot, only the first lag is significant, which is below the lower confidence interval, and then cuts off right after. All the remaining lags are not significant (fall within the confidence bands). This pattern strongly suggests an MA process of order 1 or *MA(1)*. This means that the current observation is influenced by the immediate previous error term but not by older errors.

In the PACF plot, you can observe a *gradual* decay with an oscillation pattern. Generally, if PACF shows a gradual decay, it can suggest an MA process. The pattern, where ACF has a sharp cutoff and PACF shows a gradual decay, strengthens our conclusion, as it is characteristic of an MA process.

This combination of patterns suggests that our differenced data would be well-modeled by either an ARMA or ARIMA model:

- An $ARMA(0, 1)$ model if you are working with data that has been differenced to make it stationary
- An $ARIMA(0, 1, 1)$ model if you want to incorporate first-order differencing with $d=1$ into the model.

In either ARMA or ARIMA, the AR order is $p=0$, and the MA order is $q=1$.



To effectively use ACF and PACF plots for identifying orders of AR and MA processes, the time series needs to be stationary.

- Now, let's see how PACF and ACF can be used with a more complex dataset containing strong trends and seasonality. In *Figure 9.1*, the *Monthly Milk Production* plot shows an annual seasonal effect and a positive upward trend, indicating a non-stationary time series. This can also be confirmed using an ACF plot:

```
plot_acf(milk, lags=25, zero=False);
```

This should produce the following:

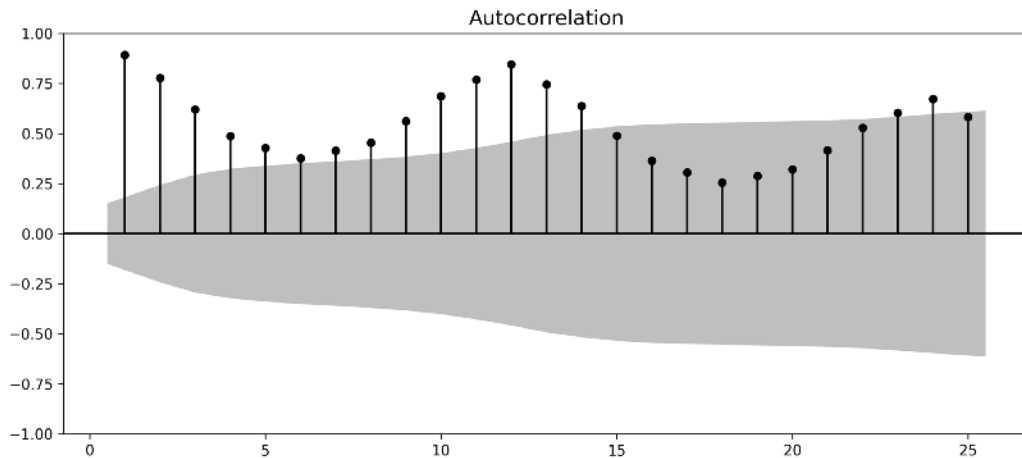


Figure 9.6: The ACF plot for the Milk Production dataset showing a slow decay that remains persistently high across several lags

The resulting ACF plot (Figure 9.6) shows a non-stationary time series with a repeating pattern:

- The slow decay confirms the trend
 - The repeating peaks every 12 lags reveal the annual seasonality
4. Since the *Milk Production* data has both seasonality and trend, a SARIMA model is more suitable. In a SARIMA model, you have two components:
- A regular ARIMA component representing the non-seasonal orders, (p, d, q)
 - A seasonal component with orders (P, D, Q, S) , where S represents the periodicity (e.g., 12 for monthly data or 4 for quarterly data)

In addition to the AR and MA processes for the **non-seasonal components** represented by lowercase p and q , which you saw earlier, you will have additional AR and MA orders for the **seasonal component**, which are represented by uppercase P and Q , respectively. The upper-case D represents the seasonal differencing. This can be written as $SARIMA(p, d, q)(P, D, Q, S)$. You will learn more about the SARIMA model in the *Forecasting univariate time series data with seasonal ARIMA* recipe.

To make such a time series stationary, you will start with seasonal differencing to remove the seasonal effect. Since the observations are taken monthly, the seasonal effects are observed annually (every 12 months or period):

```
milk_diff_12 = milk.diff(12).dropna()
```

5. Use the `check_stationarity` function that you created earlier in this chapter to perform an ADF test to check for stationarity:

```
check_stationarity(milk_diff_12)
>> 'Non-Stationary      p-value:0.16079880527711382      lags:12
```

6. The differenced time series is still not stationary, so you still need to perform a second differencing to remove any remaining trend. You will need to perform first-order differencing (de-trend). When the time series data contains seasonality and trend, you may need to difference it twice to make it stationary. Store the resulting DataFrame in the `milk_diff_12_1` variable and run `check_stationarity` again:

```
milk_diff_12_1 = milk.diff(12).diff(1).dropna()
check_stationarity(milk_diff_12_1)
>> 'Stationary      p-value:1.865423431878876e-05
lags:11
```

Great – now, you have a stationary process.

7. Plot ACF and PACF for the stationary time series in `milk_diff_12_1`:

```
fig, ax = plt.subplots(2, 1, figsize=(12, 8))
plot_acf(milk_diff_12_1, zero=False, ax=ax[0], lags=36)
plot_pacf(milk_diff_12_1, zero=False, ax=ax[1], lags=36);
```

This should produce the following ACF and PACF plots:

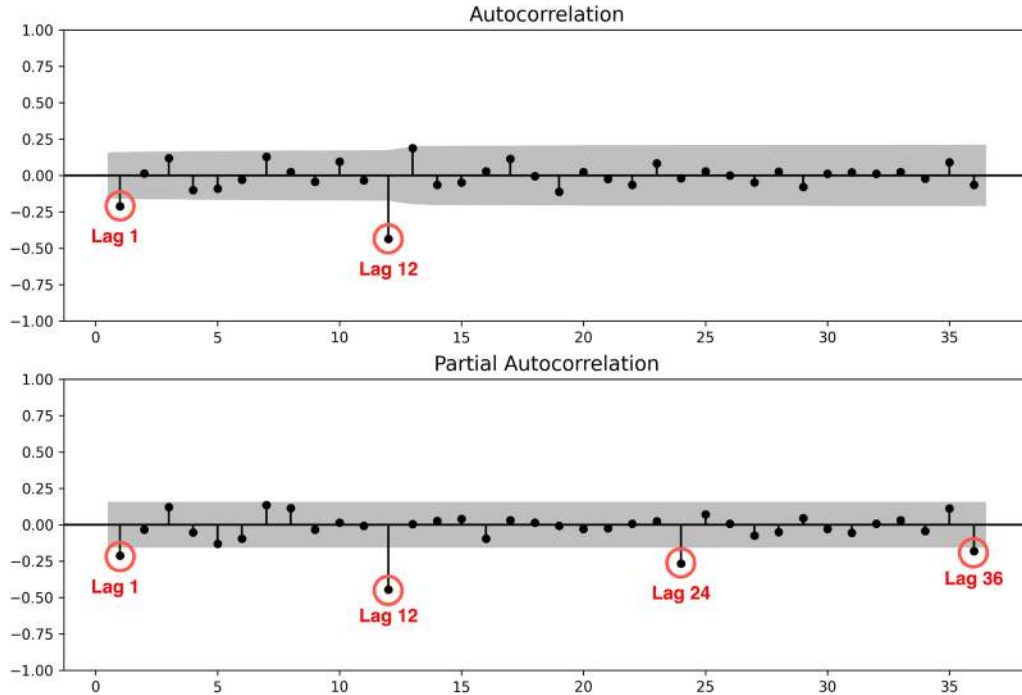


Figure 9.7: PACF and ACF plots for Monthly Milk Production after differencing

For the seasonal orders, P and Q , you should diagnose spikes or behaviors at lags s , $2s$, $3s$, and so on, where s is the number of periods in a season. For example, in the *Milk Production* data, $s=12$ (since there are 12 monthly periods in a season). Then, we observe for significance at 12 (s), 24 ($2s$), 36 ($3s$), and so on.

Starting with the ACF plot, there is a significant spike at lag 1, which represents the non-seasonal order for the MA process ($q=1$). The spike at lag 12 represents the seasonal order for the MA process ($Q=1$). Notice that there is a sharp cut-off right after lag 1, followed by another significant spike at lag 12, followed by a sharp cut-off (no other significant lags afterward). This pattern suggests an MA process: $MA(1)$ for the non-seasonal component and $MA(1)$ for the seasonal component.

The PACF plot confirms this as well, showing an exponential decay at lags 12, 24, and 36, which is consistent with an MA process. The combination of these patterns (ACF and PACF plots) suggests a $SARIMA(p,d,q)(P,D,Q,S)$ model that can be specified as $SARIMA(0, 1, 1)(0, 1, 1, 12)$. Since we already performed seasonal differencing (to remove annual seasonality) followed by non-seasonal differencing (to remove remaining trend) to make the data stationary, we include $D=1$ (seasonal differencing) and $d=1$ (non-seasonal differencing) in the model. This ensures that the data is stationary and ready for SARIMA modeling.



Though using ACF and PACF plots provides valuable insights for model order selection (e.g., orders p and q in ARIMA), they should not be used alone. They are usually used in conjunction with other techniques such as information criteria (AIC or BIC), cross-validation, and residual analysis for optimal model selection. You will explore the use of information criteria in this chapter.

How it works...

The ACF and PACF plots provide visual tools for understanding temporal relationships in time series data - correlations and dependencies that exist between observations in a time series and their past values at different time lags. The ACF plot shows both direct and indirect relationships, while the PACF plot isolates direct relationships by removing the effects of intermediate lags. For these plots to be meaningful in identifying the structure of ARIMA-based models, the time series needs to be stationary.

ACF plots can also help assess stationarity. For stationary series, the AFC would typically decline to zero relatively quickly as lags increase, while for non-stationary series, the ACF decays slowly or remains persistently high.

In these plots, **significant** autocorrelation or partial autocorrelation is identified when spikes exceed the shaded **confidence interval**. This interval, controlled by the `alpha` parameter in `statsmodels`, is set to `0.05` by default, corresponding to a 95% confidence level. Key points to consider when interpreting these spikes are as follows:

- **Direction and magnitude:** Significant spikes closer to +1 indicate strong positive correlations, while those closer to -1 indicate strong negative correlations.
- **Spike location:** Spikes at lower lags are generally more informative. Isolated spikes at higher lags might represent noise unless they fit a clear pattern.
- **Multiple spikes:** Clusters of significant spikes often indicate complex patterns, suggesting the need for higher-order ARIMA components.
- **Isolated spikes:** A single significant spike at a low lag can suggest specific components, such as an $AR(1)$ (PACF) or $MA(1)$ (ACF) process.

Spikes within the shaded area are considered insignificant, as they likely represent random noise.

By analyzing the patterns of significant spikes in the ACF and PACF plots, you can identify whether an AR, MA, or ARMA model fits your data. The following table provides a practical guide:

Process	ACF	PACF
AR(p)	Gradual decay after lag p, which can be oscillating	Cut-off after lag p
MA(q)	Cut-off at lag q	Gradual decay after lag q, which can be oscillating
ARMA(p,q)	Gradual decay after lag p, which can be oscillating	Gradual decay after lag q, which can be oscillating

Table 9.1: Identifying the AR, MA, and ARMA models using ACF and PACF plots

While ACF and PACF plots are powerful diagnostic tools, they may not always yield clear patterns, especially in noisy datasets or when multiple processes overlap. In such cases, complement these visual tools with additional techniques, such as information criteria (e.g., AIC/BIC) scores, residual analysis, and cross-validation to refine model selection.

There's more...

In the recipe, you used ACF and PACF plots to identify the structure of the SARIMA model and estimate the order values (lags) to use in seasonal (p, d, q) and non-seasonal (P, D, Q) components.

The ACF plot can be a useful tool in model diagnosis, more specifically, checking for **residual autocorrelation**. Inspecting a model's residuals is an integral part of evaluating a model. The assumption here is quite simple: if the model captured all the necessary information correctly, then the residuals should not include any correlated data points at any lags (no autocorrelation). Hence, you would expect an ACF plot of the residuals to show autocorrelations closer to zero (no significant spikes).

Let's build the SARIMA model we identified earlier in this recipe as $SARIMA(0,1,1)(0,1,1,12)$, then use the ACF to diagnose the residuals. If the model captured all the information that's been embedded within the time series, you would expect the residuals to have no autocorrelation:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
model = SARIMAX(
    milk, order=(0,1,1),
    seasonal_order=(0,1,1, 12)
).fit(dispatch=False)
```

For residual diagnostics, it is best to use the standardized residuals instead of the raw residuals.



Residual standardization is a common technique that makes comparing residuals between models easier because they are normalized and expressed in terms of standard deviations. This process transforms raw forecast errors by normalizing (scaling) them to have unit variance, ensuring that they are on a uniform scale. Standardized residuals make it easier to evaluate model performance and perform statistical tests. When residuals are standardized, we can effectively assess whether they meet key model assumptions, particularly if they behave like white noise. While standardization does not reduce the influence of outliers directly, it helps highlight extreme deviations from the mean, simplifying their identification.

The `standardized_forecasts_error` property can be accessed from the `FilterResults` object as shown:

```
standardized_resid = model.filter_results.standardized_forecasts_error[0]
```

Create the ACF plot using the standardized residuals:

```
plot_acf(standardized_resid,  
         zero=False,  
         lags=36,  
         title='Standardized Residuals ACF plot')
```

This should produce the following autocorrelation plot:

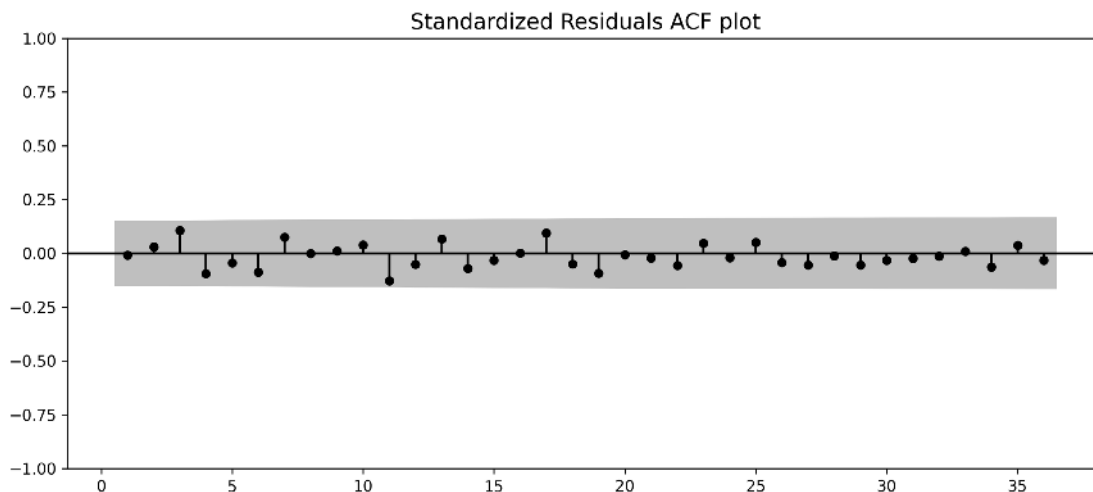


Figure 9.8: Autocorrelation plot of the SARIMA standardized residuals

The ACF plot in *Figure 9.8* shows how well our $SARIMA(0,1,1)(0,1,1,12)$ model has captured the patterns in the data. The plot reveals two key observations:

- There are no spikes extending beyond the shaded area (confidence interval), indicating that the model has successfully captured the main autocorrelation structures in the data.
- You can observe some oscillatory behavior within the confidence bounds. Though these fluctuations aren't statistically significant, they suggest there might be room for minor improvements.

You can further tune the model and experiment with different SARIMA specifications by trying different values for the seasonal and non-seasonal orders. In later recipes, you will explore a grid search method for selecting the optimal hyperparameter values to get the best model.

For a simpler way to diagnose your model's residuals, you can use the `plot_diagnostics` method:

```
model.plot_diagnostics(figsize=(12,7), lags=36);
```

This should produce the following plots:

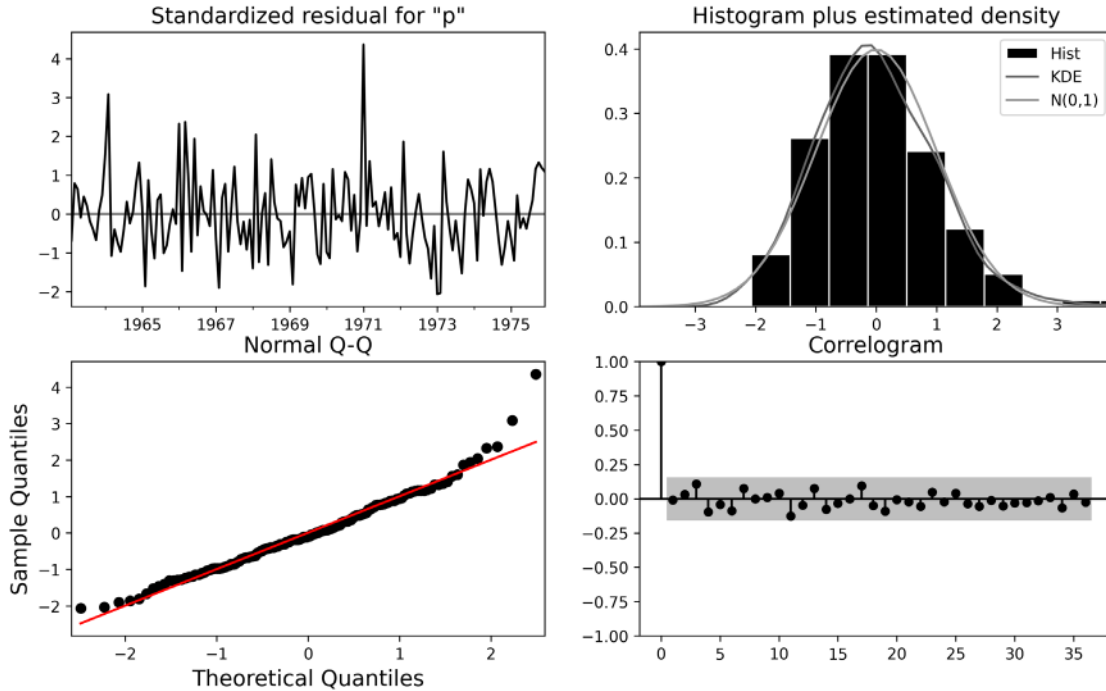


Figure 9.9: Residual analysis using the `plot_diagnostics` method

The diagnostic plots produced are based on the standardized residuals.

In the next recipe, you will be introduced to this chapter's first time series forecasting technique.

See also

- To learn more about ACF plots, visit the official documentation at https://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_acf.html
- To learn more about PACF plots, visit the official documentation at https://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_pacf.html
- To read about the standardized forecast errors in statsmodels, visit the official documentation at https://www.statsmodels.org/stable/generated/statsmodels.tsa.statespace.kalman_filter.FilterResults.standardized_forecasts_error.html

Forecasting univariate time series data with exponential smoothing

Exponential smoothing is a time series forecasting technique that gives more weight (higher importance) to recent observations, meaning what happened yesterday has a larger influence on today's forecast compared to events further in the past. The influence of past observations decays exponentially as you go further back in time. This is achieved through a smoothing parameter that determines how quickly the weights decrease. The concept of decaying weights aligns with the intuitive understanding that more recent events hold more predictive power, and thus, exponential smoothing dampens the impact of random fluctuations or outliers, making it more resilient.

This concept is central to exponential smoothing and distinguishes it from methods such as simple MA, which treats all past observations equally.

In this recipe, you will explore the exponential smoothing technique using the `statsmodels` library, which offers functionality similar to popular implementations from the R forecast package, such as `ets()` and `HoltWinters()`. These methods are widely used in demand forecasting, inventory management, and financial time series analysis due to their simplicity and effectiveness.

In `statsmodels`, there are three different implementations (classes) of exponential smoothing, depending on the nature of the data you are working with:

- **SimpleExpSmoothing:** Simple exponential smoothing is used when the time series process lacks seasonality and trend. This is also referred to as *single exponential smoothing*.
- **Holt:** Holt's exponential smoothing is an enhancement of the simple exponential smoothing and is used when the time series process contains only trend (but no seasonality). It is referred to as *double exponential smoothing* and can handle both additive and multiplicative trends.
- **ExponentialSmoothing:** Holt-Winters' exponential smoothing is an extension of Holt's exponential smoothing and is used when the time series process has both seasonality and trend. It is referred to as *triple exponential smoothing* and can handle both additive and multiplicative seasonality patterns.

You can import the classes as shown:

```
from statsmodels.tsa.api import (ExponentialSmoothing,
                                  SimpleExpSmoothing,
                                  Holt)
```

The `statsmodels` implementation aligns with the definitions and methodologies described in *Forecasting: Principles and Practice*, by Rob J. Hyndman and George Athanasopoulos, which you can explore here: <https://otexts.com/fpp3/expsmooth.html>.

How to do it...

In this recipe, you will perform exponential smoothing on the two datasets (*Life Expectancy* and *Monthly Milk Production*) introduced earlier in the chapter (see the *Technical requirements* section).

While `statsmodels` provides three different classes (`SimpleExpSmoothing`, `Holt`, and `ExponentialSmoothing`), we'll focus on using the `ExponentialSmoothing` class as our primary tool since it's the most versatile implementation. It allows you to handle various types of time series data—whether they exhibit trend, seasonality, or both—using a single implementation.

Let's get started:

1. Start by importing the `ExponentialSmoothing` class:

```
from statsmodels.tsa.api import ExponentialSmoothing
```

2. Before we apply exponential smoothing, let's understand the key parameters (referred to as hyperparameters) that control our model's behavior. These parameters fall into two categories: those that define the model's structure (specified during construction) and those that control how the model learns from the data (specified during fitting).

Model construction:

- `trend`: Specifies the trend type. Choose from `'multiplicative'` (alias `'mul'`), `'additive'` (alias `'add'`), or `None`.
- `seasonal`: Specifies the seasonality type. Choose from `'multiplicative'` (alias `'mul'`), `'additive'` (alias `'add'`), or `None`.
- `seasonal_periods`: Defines the length of one complete seasonal cycle. It takes an integer representing the seasonality period; for example, use 12 for monthly data or 4 for quarterly data.
- `damped_trend`: A Boolean value (`True` or `False`) to specify whether the trend should be damped. For example, when `True`, it assumes that trends will flatten out over time.
- `use_boxcox`: A Boolean value (`True` or `False`) to determine whether a Box-Cox transform should be applied to stabilize the variance.

Model fitting:

- `smoothing_level`: A float specifying the smoothing factor for the level known as **alpha** (α), with valid values between 0 and 1
- `smoothing_trend`: A float specifying the smoothing factor for the trend known as **beta** (β), with valid values between 0 and 1
- `smoothing_seasonal`: A float specifying the smoothing factor for the seasonal trend known as **gamma** (γ), with valid values between 0 and 1



Generally, lower values (closer to 0) for these smoothing parameters result in models that are more stable but less responsive to recent changes (overly smoothed), while higher values (closer to 1) create more responsive but potentially more volatile models (less smooth and potentially capturing noise or short-term fluctuations).

Later, in the *How it works...* section, you will explore the **Holt-Winters** formulas for level, trend, and seasonality and how these parameters are applied.

3. To evaluate different combinations of hyperparameters, create a list that contains all possible combinations. Later, you'll get to evaluate the different combinations of hyperparameter values at each run. Essentially, you will train a different model and capture its scores during each iteration. Once every combination has been evaluated, you will use the `get_top_models_df` function (see the *Technical requirements* section) to determine the best-performing model with its optimal hyperparameter values through this exhaustive grid search. This process can be time-consuming, but fortunately, there is an alternative hybrid technique to shorten the search.
4. You will use the `ExponentialSmoothing` class to find the optimal values for `smoothing_level`, `smoothing_trend`, and `smoothing_seasonal`. This approach reduces the number of parameters in your grid (although you can still do so if you prefer to control the process). While the `ExponentialSmoothing` class can automatically optimize the smoothing parameters (α, β, γ), we still need to determine the best structural parameters (model construction parameters such as trend type and damping). You can identify whether the components are multiplicative or additive by plotting their decomposition using the `seasonal_decompose()` function. If you're still uncertain, the exhaustive grid search remains a viable alternative.

Let's start with the *Life Expectancy* dataset (the `life_train` and `life_test` sets). For this dataset, you only have trend (no seasonality), so you only need to explore different values for the two parameters – trend type (additive or multiplicative) and whether to dampen the trend:

```
trend = ['add', 'mul']
damped = [True, False]

life_ex_comb = combinator([trend, damped])
print(life_ex_comb)
>>
[('add', True), ('add', False), ('mul', True), ('mul', False)]
```

5. Iterate through each combination and train (fit) a different model. Capture evaluation metrics in a dictionary to compare the results later. Example evaluation metrics include RMSE, RMSPE, MAPE, AIC, and BIC, to name a few:
 - **Error metrics** (lower is better), such as RMSE, MAPE, and RMSPE, measure how well the model predicts actual values. They focus on forecast accuracy and are useful when the primary goal is to minimize forecast errors.
 - **Information criteria metrics** (lower is better), such as AIC, AICc, and BIC, balance model fit against complexity.

Generally, most automated tools and software will use information criteria (AIC or BIC) scores behind the scenes to determine the best model:

```
train = life_train.values.ravel()
y = life_test.values.ravel()
score = {}
for i, (t, dp) in enumerate(life_ex_comb):
    exp = ExponentialSmoothing(train,
                               trend=t,
                               damped_trend=dp,
                               seasonal=None)

    model = exp.fit(use_brute=True, optimized=True)
    y_hat = model.forecast(len(y))
    score[i] = {'trend':t,
                'damped':dp,
                'AIC':model.aic,
                'BIC':model.bic,
                'AICc':model.aicc,
                'RMSPE': rmspe(y, y_hat),
                'RMSE' : rmse(y, y_hat),
                'MAPE' : mape(y, y_hat),
                'model': model}
```

In the code, you passed the `life_train` set for training the different models, and the `life_test` set for evaluating the error metrics such as RMSPE, RMSE, and MAPE.

6. With our models trained, you rank them using the `get_top_models_df` function. We'll use AIC as our primary selection criterion, which is the default. To use the function, just pass the score dictionary. Keep the default parameter values for `criterion` and `top_n`:

```
model_eval = get_top_models_df(score)
```

The `get_top_models_df` function will return a DataFrame displaying the top models (top five by default), ranked based on the criterion selected, which is the AIC score in this case. If you would like to sort the models based on BIC and only get the top three, you can do so as shown:

```
get_top_models_df(score, criterion='BIC', top_n=3)
```

The DataFrame not only includes all additional scores but also stores the model instances themselves in a column labeled 'model'.

7. To view the ranking and the various scores, you can execute the following line:

```
model_eval.iloc[:, 0:-1]
```

The preceding code excludes the last column, which contains the model instances, thus displaying a DataFrame that includes a column for each of the evaluation metrics, such as AIC, BIC, RMSE, and so on.

	trend	damped	AIC	BIC	AICc	RMSPE	RMSE	MAPE
model_id								
1	add	False	-137.030604	-129.382512	-135.077115	0.067753	0.561541	0.005570
3	mul	False	-136.344215	-128.696123	-134.390727	0.070527	0.584578	0.005835
2	mul	True	-134.829825	-125.269710	-132.163159	0.064569	0.535100	0.005268
0	add	True	-133.771862	-124.211747	-131.105195	0.051096	0.423379	0.003974

Figure 9.10: Exponential smoothing models for the Life Expectancy data ranked based on AIC scores

The columns in Figure 9.10 are as follows:

- `model_id` is the key of the model in the score dictionary
- `trend` and `damped` represent the different combinations used
- The remaining columns are the different metrics captured

Generally, for model selection based on information criteria such as AIC, BIC, and AICc, lower values are better, indicating a more optimal balance between model fit and complexity. In our case, we opted to use AIC. If you inspect the DataFrame in Figure 9.10, there are a few observations you can make:

- If prioritizing information criteria (AIC, BIC, or AICc), **model 1** (trend: additive, damped: False) would be considered the best model as it scores the lowest across all three information criteria. This model likely offers the best trade-off between model complexity and fit.
- If prioritizing error metrics (RMSPE, RMSE, or MAPE), which measure prediction accuracy, **model 0** (trend: additive, damped: True) would be considered superior due to its lower prediction errors.



Selecting the “winning” model will depend on your specific goals and the context in which the model will be used. If you need a balance between both approaches, you might consider other factors or further validation to decide between models 1 and 0.

We will continue with the AIC as our selection criteria.

8. The four models stored in the DataFrame are instances of the `HoltWintersResultsWrapper` class, which gives you access to several attributes and methods for analysis and forecasting, such as `summary`, `predict`, and `forecast`. You can extract the winning model directly from the DataFrame with the following code:

```
top_model = model_eval.iloc[0,-1]
```


You can access the `summary()` method, which provides a comprehensive view of the model's characteristics:

```
top_model.summary()
```

The preceding code will produce a summary output that provides a tabular layout detailing the model—for example, the parameter values that were used and the coefficients calculated:

ExponentialSmoothing Model Results			
Dep. Variable:	endog	No. Observations:	50
Model:	ExponentialSmoothing	SSE	2.749
Optimized:	True	AIC	-137.031
Trend:	Additive	BIC	-129.383
Seasonal:	None	AICC	-135.077
Seasonal Periods:	None	Date:	Mon, 03 Nov 2025
Box-Cox:	False	Time:	22:59:27
Box-Cox Coeff.:	None		

	coeff	code	optimized
smoothing_level	0.1621215	alpha	True
smoothing_trend	0.1621215	beta	True
initial_level	68.748321	l.0	True
initial_trend	0.2422562	b.0	True

Figure 9.11: Exponential smoothing summary for the Life Expectancy data

The summary will show key information such as the optimal values for **alpha** (`smoothing_level`) and **beta** (`smoothing_trend`) that have been automatically deduced by the fitting process.

- You can forecast future values using the `forecast` method and then evaluate the results against the test set (unseen data by the model). The `plot_forecast()` function, introduced in the *Technical requirements* section, will be used to generate and plot the forecast results alongside the test data. To perform this visualization, pass the model instance stored in `top_model` along with both the training and test datasets to `plot_forecast()`:

```
plot_forecast(top_model, '2000', life_train, life_test)
```

The `start` argument in the `plot_forecast` function slices the data from that point forward to make it easier to compare the results. Think of it as zooming in on a specific segment of the timeline. For example, instead of displaying data spanning from 1960 to 2018 (59 months), you request only the segment starting from the year 2000.

This should produce a plot with the x axis starting from the year 2000. There should be three lines: one line representing the training data, another line for the test data, and a third line depicting the forecast (predicted values):

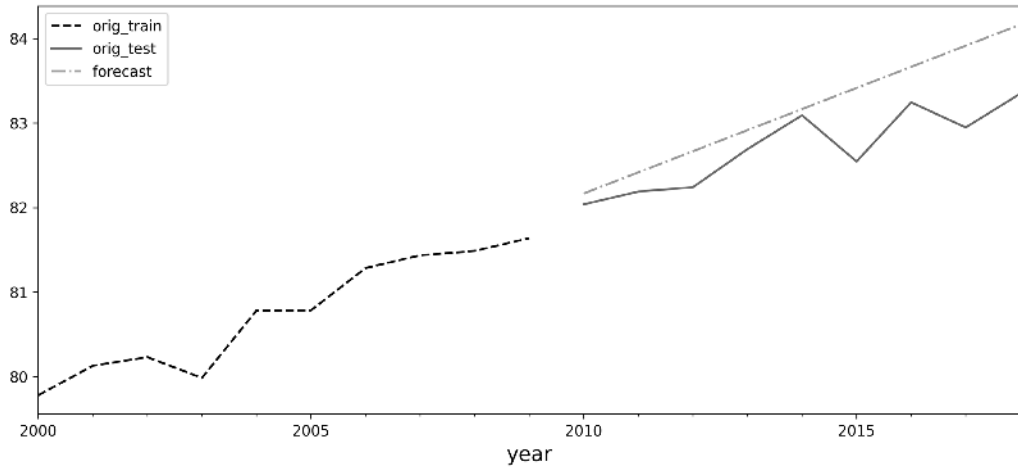


Figure 9.12: Plotting the exponential smoothing forecast versus the actual data for the Life Expectancy dataset

Looking at Figure 9.12, we can observe that our model has captured the underlying upward trend in life expectancy. The forecast line extends this trend into the future, though it's worth noting that it does so in a linear fashion. This makes sense, given our model's structure (additive trend without damping), but it also highlights a potential limitation: real-world trends often don't continue indefinitely in a straight line. While the model shows good statistical performance, any significant future predictions should be considered alongside expert knowledge about factors that might influence life expectancy trends.

10. Now, let's tackle the *Milk Production* dataset, which presents a more complex challenge due to its seasonal patterns. Unlike our *Life Expectancy* data, we need to consider both trend and seasonality. The key difference here is the inclusion of two additional hyperparameters:
 - `seasonal`: Additive or multiplicative.
 - `seasonal_periods`: Length of seasonal cycle. Here, you can explore 4 months (might capture quarterly patterns), 6 months (could identify semi-annual cycles), and 12 months (detect annual seasonality).
11. Build a Cartesian product to construct the grid for the different options. This combination should give you a total of 24 models that you will need to evaluate for (2 trend types x 2 trend damping options x 2 seasonal types x 3 seasonal period options).

Let's train these models and capture their performance:

```
trend , damped= ['add', 'mul'], [True, False]
seasonal, periods = ['add', 'mul'], [4, 6, 12]
milk_exp_comb = combinator([trend, damped, seasonal, periods])
```

Loop through each combination to train and evaluate models:

```
train = milk_train.values.ravel()
y = milk_test.values.ravel()
milk_model_scores = {}
for i, (t, dp, s, sp) in enumerate(milk_exp_comb):
    exp = ExponentialSmoothing(train,
                               trend=t,
                               damped_trend=dp,
                               seasonal=s,
                               seasonal_periods=sp)
    model = exp.fit(use_brute=True, optimized=True)
    y_hat = model.forecast(len(y))
    score[i] = {'trend':t,
               'damped':dp,
               'AIC':model.aic,
               'BIC':model.bic,
               'AICc': model.aicc,
               'RMSPE': rmspe(y, y_hat),
               'RMSE' : rmse(y, y_hat),
               'MAPE' : mape(y, y_hat),
               'model': model}
```

12. Once training is completed, run the `get_top_models_df` function to rank and identify the top five models based on the AIC score:

```
model_eval = get_top_models_df(score, 'AIC', top_n=5)
model_eval.iloc[:, 0:-1]
```

This should display the following DataFrame:

	trend	damped	AIC	BIC	AICc	RMSPE	RMSE	MAPE
model_id								
8	add	False	593.713882	641.119396	599.230011	0.148817	12.769731	0.011725
20	mul	False	594.758581	642.164095	600.274710	0.175151	15.049807	0.014274
2	add	True	595.562717	645.931075	601.741579	0.129827	11.036820	0.010305
11	add	False	614.894416	662.299930	620.410545	0.220576	19.858487	0.017101
23	mul	False	615.504756	662.910271	621.020886	0.246720	22.160203	0.019684

Figure 9.13: Top five exponential smoothing models for the Milk Production data ranked based on AIC scores

To determine the winning model from the results, you typically look at various metrics such as AIC, BIC, AICc (which are information criteria), and error metrics such as RMSPE, RMSE, and MAPE. Lower values in AIC, BIC, and AICc indicate a model with a better balance between goodness of fit and complexity. Lower values in RMSPE, RMSE, and MAPE indicate better predictive accuracy.

If you inspect the DataFrame in *Figure 9.13*, there are a few observations you can make:

- If prioritizing information criteria (AIC, BIC, or AICc), **model 8** (trend: additive, damped: False) appears to be the best model as it has the lowest values across all information criteria. This suggests that it provides a favorable balance between fitting the data well and maintaining simplicity.
- If prioritizing error metrics (RMSPE, RMSE, or MAPE), **model 2** (trend: additive, damped: True) is superior in terms of prediction accuracy. This model has the lowest error rates, indicating it predicts future values most accurately among the models listed.



Selecting the “winning” model will depend on your specific goals and the context in which the model will be used. If you need a balance between both approaches, you might consider other factors or further validation to decide between models 8 and 2.

We will continue with the AIC as our selection criteria.

You can display the best model’s summary with the following:

```
top_model = model_eval.iloc[0,-1]
top_model.summary()
```

This should produce a tabular layout summarizing the best model – for example, the parameter values that were used to build the model and the calculated coefficients:

ExponentialSmoothing Model Results			
Dep. Variable:	endog	No. Observations:	143
Model:	ExponentialSmoothing	SSE	7265.655
Optimized:	True	AIC	593.714
Trend:	Additive	BIC	641.119
Seasonal:	Additive	AICC	599.230
Seasonal Periods:	12	Date:	Tue, 11 Nov 2025
Box-Cox:	False	Time:	20:08:53
Box-Cox Coeff.:	None		
	coeff	code	optimized
smoothing_level	0.6859931	alpha	True
smoothing_trend	0.000000	beta	True
smoothing_seasonal	0.1476242	gamma	True

Figure 9.14: Exponential smoothing summary for the Monthly Milk Production data

Notice the optimal combination of hyperparameter values for Trend, Seasonal, and Seasonal Periods. The optimal Seasonal Periods hyperparameter was at 12 months seasonality (confirming annual patterns in milk production). The summary results table will show the seasonal coefficients for each month, and it will be a long list. The preceding screenshot only shows the top section.

Additionally, the summary shows key information such as the optimal values for alpha (smoothing_level), beta (smoothing_trend), and gamma (smoothing_seasonal).

Recall that the best model is selected based on the AIC score. Therefore, you should explore different metrics that have been captured, for example, using `get_top_models_df(score, 'MAPE', top_n=5)`.

13. Compare your forecast using the best model against the test data:

```
plot_forecast(top_model, '1969', milk_train, milk_test);
```

This should result in a plot starting from the year 1969, featuring three lines representing the training data, test data, and the forecast (predicted values):

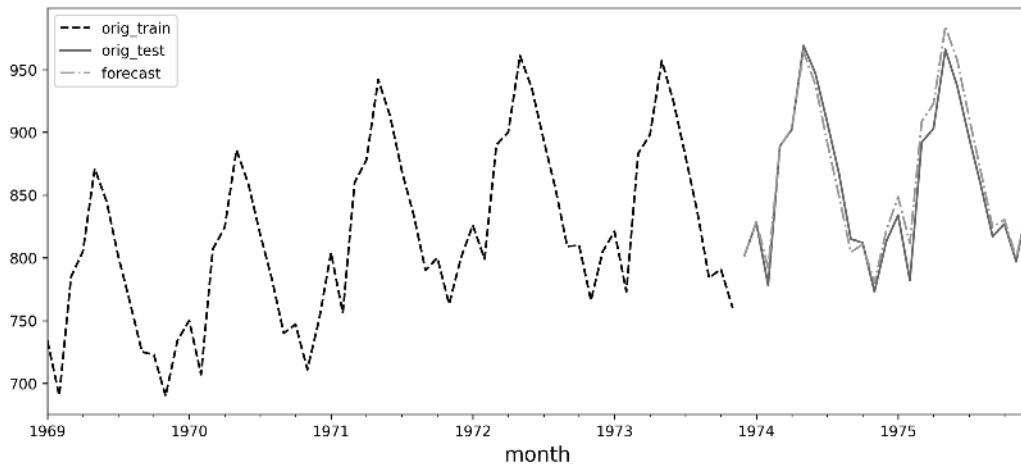


Figure 9.15: Plotting the exponential smoothing forecast versus the actual Monthly Milk Production data

Overall, the model effectively captured both the trend and seasonality, closely aligning with the actual values from the test set.

The resulting visualization (*Figure 9.15*) tells a compelling story:

- Our model successfully captures both the upward trend in milk production
- The seasonal patterns are well-represented, showing regular fluctuations
- The forecast closely tracks the actual test data, validating our model's effectiveness

How it works...

There are various techniques for smoothing time series data; exponential smoothing stands out due to its intuitive approach and simplicity. Unlike an MA model, which treats all past values equally, exponential smoothing models place greater emphasis (weight) on more recent observations, and the influence of older observations decreases (weight decay) exponentially, hence the term *exponential*.

This approach is based on an intuitive assumption that more recent events are likely to be more significant than older ones. For instance, in predicting tomorrow's temperature, yesterday's weather is probably more relevant than the temperature from two weeks ago. Exponential smoothing still considers historical data, but its influence is adjusted through a smoothing parameter to decrease exponentially.

When working with time series data, we often encounter three main patterns:

- **Level:** The baseline value around which the data fluctuates – for example, the average daily sales of a product over a month
- **Trend:** A consistent upward or downward movement over time – for instance, an increasing number of mobile app downloads due to growing popularity
- **Seasonality:** A regular pattern that repeats at fixed intervals, such as higher ice cream sales during the summer months or increased retail sales during the holiday season

Understanding these patterns will help you select the appropriate forecasting method since exponential smoothing methods are designed to address these components:

- **Simple exponential smoothing** focuses on capturing only the level component
- **Holt's method** extends this by incorporating the trend component
- **Holt-Winters' method** adds further enhancement to account for the seasonality component



Each method uses different smoothing parameters: α (alpha) for the level, β (beta) for the trend, and γ (gamma) for seasonal components, which control the exponential decrease of past observations' influence.

Simple exponential smoothing

The simple exponential smoothing method is used when the time series has no trend or seasonality. It smooths the data by estimating a single-level component. The formula is as follows:

$$S_t = \alpha x_t + (1 - \alpha)S_{t-1}$$

$$F_{t+1} = S_t$$

Here, the `ExponentialSmoothing` class aims to find the optimal value for the smoothing parameter, **alpha** (α). Let's break down this formula:

- S_t : Expected (smoothed) level at the current time t
- S_{t-1} : Previous smoothed level value at time $t - 1$

- x_t : Observed value at the current time t
- α : Level smoothing parameter ($0 < \alpha < 1$)

The alpha (α) parameter is critical, serving as the level smoothing parameter, and plays a role in determining whether the model should emphasize the past (S_{t-1}) or the present (x_t). In other words, it influences how quickly the model adapts to changes in the data:

- **When α is close to 1:** The model gives more weight to recent observations, making it more responsive to changes but potentially more sensitive to noise or short-term fluctuations (the smoothing effect diminishes). This is because the term $(1 - \alpha)S_{t-1}$ gets closer to zero, and more emphasis or weight is put on the present x_t term.
- **When α is close to 0:** The model places more weight on historical data, resulting in a smoother forecast but potentially slower response to changes in the data (overly smoothed forecast that does not capture shifts in the time series). This is because the term αx_t gets close to zero, and more emphasis or weight is put on the past S_{t-1} term.

Several factors influence the choice of α , including the degree of randomness in the system. The output value for the coefficient α determines how the model weighs current and past observations to forecast future events, F_{t+1} .

Holt's exponential smoothing

This method extends simple exponential smoothing by adding a trend (T_t) component to account for upward or downward movement in the data.

The formula for Holt's exponential smoothing incorporates the addition of the trend (T_t) at time t and its smoothing parameter beta (β). Hence, once a trend is included, the model will output the values for both coefficients – **alpha** for level (α) and **beta** for trend (β):

$$S_t = \alpha x_t + (1 - \alpha)(S_{t-1} + T_{t-1})$$

$$T_t = \beta(S_t - S_{t-1}) + (1 - \beta)T_{t-1}$$

$$F_{t+1} = S_t + T_t$$

In the formula, we added the following:

- T_t : Trend component at time t
- β : Trend smoothing parameter ($0 < \beta < 1$)

The three equations may look complex at first glance, but let's break down each portion:

- The first equation updates the level S_t , but now it considers both the previous level and trend
- The second equation updates the trend T_t
- The third equation creates the forecast by combining the current level and trend

The beta (β) smoothing parameter has a similar effect on the trend component as did the alpha (α) smoothing parameter on the level component:

- **When β is close to 1:** The trend component adapts quickly to changes in the growth rate. This is because a higher β gives more weight to the difference between the current smoothed level and the previous smoothed level ($S_t - S_{t-1}$).
- **When β is close to 0:** The trend component changes slowly, resulting in more stable trend estimates. This is because, as β gets closer to 0, it places more weight on the previous trend estimate (T_{t-1}).

Note that Holt's method can handle both additive and multiplicative trends, which was explored in the recipe.

Holt-Winters' exponential smoothing

This method incorporates both trend, T_t , and seasonality, C_t , and can handle both additive and multiplicative seasonal patterns. The following equation shows multiplicative seasonality as an example:

$$S_t = \frac{\alpha x_t}{C_{t-L}} + (1 - \alpha)(S_{t-1} + T_{t-1})$$

$$T_t = \beta(S_t - S_{t-1}) + (1 - \beta)T_{t-1}$$

$$C_t = \gamma \left(\frac{x_t}{S_t} \right) + (1 + \gamma)C_{t-L}$$

$$F_{t+1} = (S_t + T_t)C_{(t+1)-L}$$

In the formula, we added the following:

- C_t : Seasonal component at time t
- L : Length of the seasonal cycle (for example, $L = 12$ for monthly data with annual seasonality)
- γ : Seasonal smoothing parameter $0 < \gamma < 1$

These equations may look overwhelming, but let's break down each portion:

- The level equation (S_t) now adjusts the current observation by dividing out the seasonal effect before combining it with the trend-adjusted level
- The trend equation (T_t) works the same way as Holt's method
- The seasonal equation (C_t) updates the seasonal factors, capturing how much higher or lower each season tends to be compared to the overall level
- The forecast equation (F_{t+1}) multiplies the trend-adjusted level by the appropriate seasonal factor

When using `ExponentialSmoothing` to find the optimal α, β, γ parameter values, it does so by minimizing the error rate (the **sum of squared error** or **SSE**). So, in the code implementation, every time iteration in the loop, you were passing new parameter values (for example, `damped` as either `True` or `False`), the model was solving for the optimal set of values for the α, β, γ coefficients by minimizing for SSE. This can be written as follows:

$$\min_{\alpha, \beta, \gamma} \sum_{t=1}^T (x_t - F_t)^2$$

Let's break down the formula:

- x_t : The actual observed value at time t
- F_t : The forecasted value at time t , which depends on the parameters α, β , and γ
- T : The total number of time points
- α, β, γ : The smoothing parameters for level, trend, and seasonality, respectively

In some textbooks, you will see different letters used for level, trend, and seasonality, but the overall structure of the formulas holds.

Generally, exponential smoothing is a fast and effective technique for smoothing a time series for improved analysis, dealing with outliers, data imputation, and forecasting (prediction).

There's more...

The Darts library is a versatile time series forecasting framework that supports a wide range of models, including traditional methods such as `ExponentialSmoothing` (a wrapper around `statsmodels`), advanced implementations such as `StatsForecastAutoETS` (via Nixtla's **StatsForecast**), and modern machine learning-based approaches such as **Prophet** and **Auto-ARIMA** (via **pmdarima**). This makes Darts a one-stop solution for working with time series data, offering flexibility to experiment with different models and techniques.



It's a good practice to create a new Python virtual environment when installing new packages. This helps avoid package conflicts and makes your work reproducible. You can refer to *Chapter 0* on GitHub for detailed instructions on setting up a Python virtual environment.

To install Darts using `pip`, run the following command:

```
pip install "u8darts[notorch]"
```

Load the `ExponentialSmoothing` and `TimeSeries` classes:

```
from darts import TimeSeries
```

Darts introduces its own `TimeSeries` class, which is a core abstraction that provides a unified interface for time series operations. It can handle univariate and multivariate time series and support deterministic and stochastic series, making it versatile for various forecasting tasks.

Darts expects the data to be an instance of the `TimeSeries` class, so you need to convert your pandas `DataFrame` before using it to train the model. The `TimeSeries` class provides the `from_dataframe` method, which you will be using:

```
ts = TimeSeries.from_dataframe(milk.reset_index(),
                              time_col='month',
                              value_cols='production',
                              freq='MS')
```

When creating the `TimeSeries` object using `from_dataframe`, you need to specify the following:

- Which column contains the date (`time_col`).
- Which column contains the observations (`value_cols`). If `value_cols` is set to `None` (default), then the entire `DataFrame` is considered.
- The frequency of your data (`freq`).

Split the new `ts` dataset into train and test:

```
train, test = split_data(ts, 0.15)
```

Using Darts exponential smoothing

Start by importing the `ExponentialSmoothing` class and instantiating your model. You can keep the default values, and provide `seasonal_periods` (though if not provided, it can be inferred from the frequency of the series):

```
from darts.models import ExponentialSmoothing
model = ExponentialSmoothing(seasonal_periods=12)
```

You can train the model using the `.fit()` method:

```
model.fit(train)
```

Once trained, you can forecast using the `.predict()` method.

```
forecast = model.predict(len(test), num_samples=100)
```

To plot the results, you can use the `.plot()` method:

```
train.plot()
forecast.plot(label='forecast', low_quantile=0.05, high_quantile=0.95);
```

This should produce the following plot:

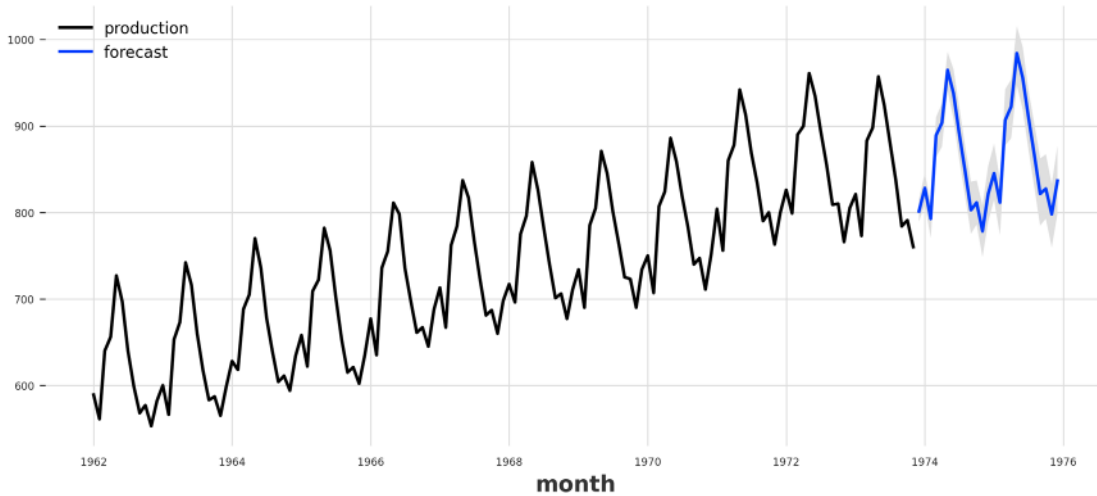


Figure 9.16: The *ExponentialSmoothing* forecast for the Monthly Milk Production data using Darts

Darts' *ExponentialSmoothing* class is a wrapper to *statsmodels*'s *ExponentialSmoothing* class, which means you have access to familiar methods and attributes, such as the `summary()` method:

```
model.model.summary()
```

This should produce the familiar *statsmodels* tabular summary of the model and the optimized parameter values. Compare the summary from Darts with the results shown in Figure 9.14.

Using AutoETS

The Darts library features *AutoETS*, which is another powerful class that draws its functionality from the automatic exponential smoothing (**AutoETS**) implementation in the *StatsForecast* library. Compared to the traditional *ExponentialSmoothing* class, *AutoETS* is often lauded for its faster performance. *AutoETS* selects the optimal parameter values based on the information criteria, AICc by default. ETS is similar to *ExponentialSmoothing*, except that its components include error (*E*), trend (*T*), and seasonality (*S*), hence the term *ETS*.

Exponential smoothing versus ETS



ETS and exponential smoothing are very closely related, as they both use smoothing of past data points for forecasting time series data. However, they differ in their approach. Exponential smoothing estimates parameters by minimizing the SSE, whereas ETS models use **maximum likelihood estimation (MLE)**, which enables better uncertainty quantification. Both methods provide point forecasts; however, ETS extends exponential smoothing by incorporating probabilistic frameworks that allow for modeling error, trend, and seasonality components and generate prediction intervals in addition to point forecasts.

Darts makes it easy to explore and experiment with different models due to the unified `TimeSeries` class. Since our data has already been converted and split, you are ready to go. The process/steps are similar to what you just did earlier with Darts' `ExponentialSmoothing` method.

To explore the capabilities of `AutoETS`, consider the following code snippet:

```
from darts.models import AutoETS
# Create the model instance
model_ets = AutoETS(season_length=12)
# Fit the model
model_ets.fit(train)

# Create your forecast
ets_forecast = model_ets.predict(len(test), num_samples=100)

# Plot your forecast
train.plot()
ets_forecast.plot(label='AutoETS');
```

This should produce the following plot:

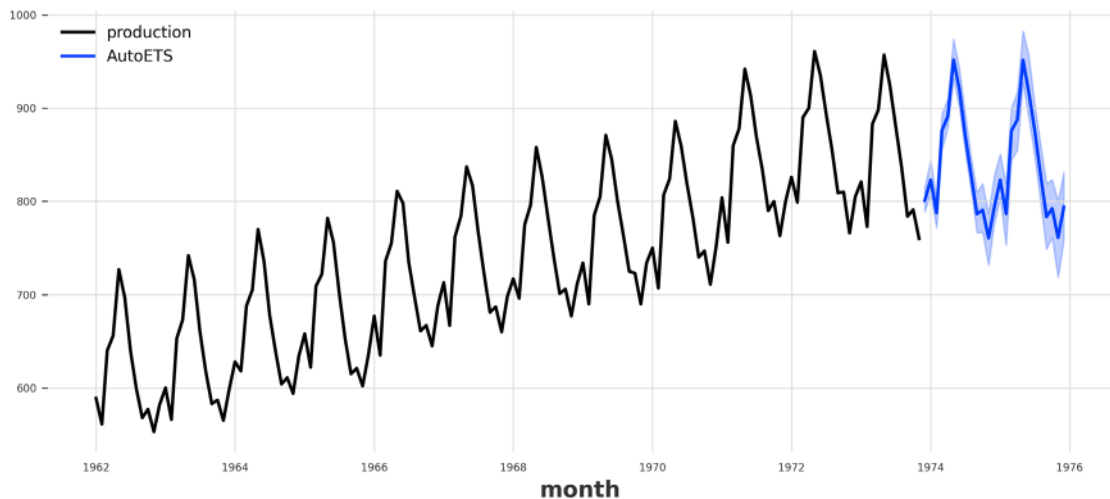


Figure 9.17: `AutoETS` forecast for the Monthly Milk Production data using Darts

A powerful feature of Darts is its ability to generate probabilistic forecasts. As shown in Figures 9.16 and 9.17, the shaded area in our plots represents the range of possible future values that our model considers plausible. This was controlled through the `num_samples` parameter, which determines how many Monte Carlo samples the model generates. When `num_samples=1`, the model produces a single deterministic forecast. However, when we set a larger value, such as `num_samples=100`, the model generates 100 different potential future scenarios, creating the shaded area around the main forecast line. This can be useful in situations where understanding prediction uncertainty is critical.

Simply, using `num_samples > 1` enables probabilistic forecasting, giving us not just a single prediction but a range of likely outcomes.

You can compare the two forecasts from `ExponentialSmoothing` and `AutoETS` using the following code:

```
forecast.plot(label='ExponentialSmoothing')
ets_forecast.plot(label='StatsForecastAutoETS');
```

This should produce a plot where the top line represents the `ExponentialSmoothing` forecast and the bottom line represents the `StatsForecastAutoETS` forecast.

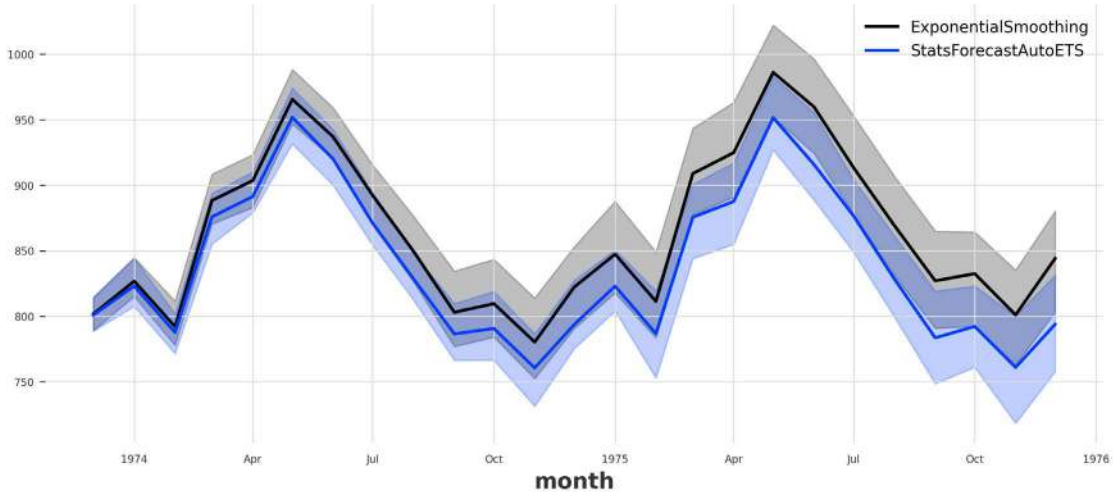


Figure 9.18: Comparing `AutoETS` and `ExponentialSmoothing`

Did you notice that you did not have to test for stationarity with exponential smoothing?

Exponential smoothing effectively handles non-stationary time series without requiring stationarity testing. **Simple exponential smoothing (SES)** adapts to changing levels in data without trend or seasonality, while Holt's and Holt-Winters' methods explicitly model non-stationary components such as trends and/or seasonal patterns.

Similarly, **ETS (Error-Trend-Seasonal)** models do not require stationarity testing. ETS models automatically identify and account for components such as error, trend, and seasonality patterns, making them robust for various time series behaviors without pre-processing steps such as differencing.

In the next recipe, while building an ARIMA model, you will be testing for stationarity to determine the differencing factor and leverage the ACF and PACF plots that were discussed earlier in this chapter.

See also

- To learn more about the `ExponentialSmoothing` class, you can visit the `statsmodels` official documentation at <https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html>.

Forecasting univariate time series data with non-seasonal ARIMA

ARIMA models are a popular choice for forecasting univariate time series data because they can effectively capture trends and autocorrelation in non-seasonal data, allowing for accurate predictions of future values. By applying ARIMA models, you can gain valuable insights into the underlying dynamics of your time series data and make informed decisions. In this recipe, you will explore the ARIMA model for time series forecasting using the `statsmodels` library. ARIMA combines three principal components:

- **Autoregressive (AR(p)):** This component captures relationships between an observation and a specific number of lagged (past) observations (p)
- **Integrated process (I(d)):** This applies differencing to the data to make a non-stationary series stationary
- **Moving average (MA(q)):** This models the relationship between an observation and its past forecast errors from an MA model applied to lagged observations (q)

An ARIMA model for a non-seasonal time series is denoted as $\text{ARIMA}(p, d, q)$, where the p and q parameters denote the orders or **number of lags** for AR and MA components, respectively, and d is the order of differencing required to achieve stationarity. The p and q parameters are called lags since they represent the number of “past” periods considered in the model. You may also come across another reference for p and q , namely, **polynomial degrees**.

ARIMA models handle non-stationary time series data through differencing, a time series transformation technique to stabilize trends. The integration or degree of differencing, d , is one of the parameters that you will need to select a value for when building the model. For a refresher on stationarity, please refer to the *Detecting time series stationarity* recipe in *Chapter 8*.

While ARIMA models are designed to handle trends through the integrated factor, d , they traditionally assume the absence of seasonality in the dataset. However, when seasonality is present, the SARIMA model extends ARIMA by incorporating seasonal differencing, making it the appropriate choice for seasonal data.

Getting ready

To implement this recipe, you will need the following classes and functions from the `statsmodels` library:

```
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.stats.diagnostic import acorr_ljungbox
```

These imports will allow you to define an ARIMA model and evaluate its performance using statistical tests such as the Ljung-Box test for residual autocorrelation.

How to do it...

Different time series models are suited to various kinds of data. Therefore, it is crucial to choose a model that aligns with the characteristics of your dataset and the specific problem you're addressing. In this recipe, you will use the `life` DataFrame, which exhibits a trend but no seasonality.

You will combine visual inspection (using the ACF and PACF plots) and statistical tests to make an informed decision for the AR and MA components of the model, known as the *p* and *q* orders. These methods were covered in *Chapter 8*, in the *Testing data for autocorrelation*, *Decomposing time series data*, and *Detecting time series stationarity* recipes. Let's get started:

1. Begin by **decomposing** the dataset to separate it into its three principal components: trend, seasonality, and residual (often considered as noise). You can achieve this using the `seasonal_decompose` function. Ensure that your dataset has a `DatetimeIndex` or a set frequency for this function to work. In our case, the `life` DataFrame already has a `DatetimeIndex`:

```
decomposed = seasonal_decompose(life)
decomposed.plot();
```

Note that this function assumes an additive model by default (`model='additive'`), which was discussed in the *Decomposing time series data* recipe in *Chapter 8*.

You can see the plot as follows:

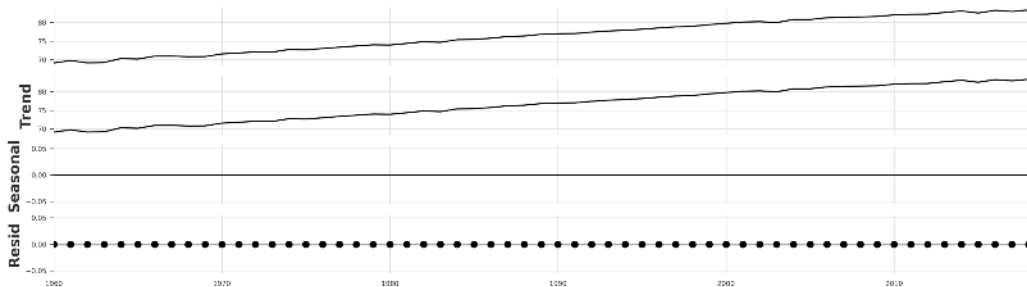


Figure 9.19: Decomposition of Life Expectancy data

Observe that the decomposition shows a positive (upward) trend within the dataset. This indicates a consistent increase over time. However, there is no observable seasonality effect, which aligns with our expectations for the `life` dataset.

2. You will need to **detrend** the data first to make it stationary. Perform a first-order differencing and then test for stationarity by using the `check_stationarity` function you created earlier in this chapter:

```
check_stationarity(life)
>>
Non-Stationary      p-value:0.6420882853800064      lags:2
```

The high p-value indicates that the null hypothesis of non-stationarity cannot be rejected. This confirms that the data is not stationary.

```
# apply first-order differencing
life_df1 = life.diff().dropna()

# recheck stationarity after differencing
check_stationarity(life_df1)

>>
Stationary      p-value:1.5562189676003248e-14      lags:1
```

First-order differencing is appropriate here because the dataset exhibits a trend but no seasonality.

Now, the data is **stationary**. The p-value is significant, and you can reject the null hypothesis.

Note that the default periods value for `diff()` is 1. Generally, `diff(periods=n)` is the difference between the current observation at period t and its lagged version at period $t-n$. In the case of `diff(1)` or `diff()`, the lagged version is $t-1$ (for example, the prior year's observation). The `dropna()` method removes any NaN values created during the differencing process.

You can plot the differenced time series data using the `plot` method:

```
life_df1.plot();
```

This should produce the following plot:

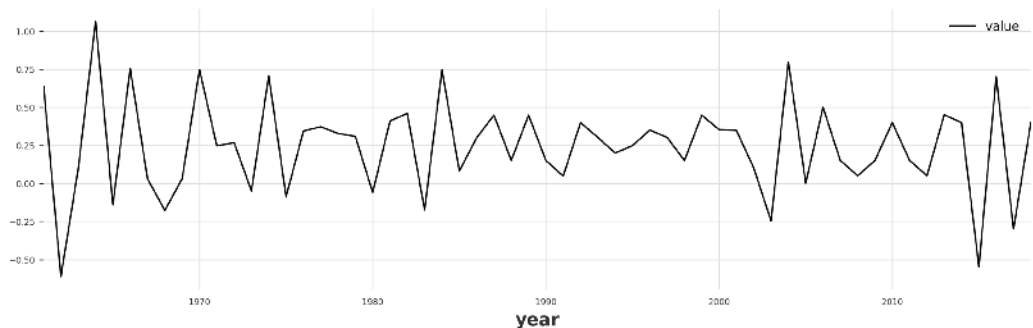


Figure 9.20: First-order differencing for Life Expectancy data (detrending)

Next, you will need to determine the p and q orders for the $ARIMA(p, d, q)$ model.

3. The ACF and PACF plots will help you estimate the appropriate p and q values for the AR and MA models, respectively. Use `plot_acf` and `plot_pacf` on the stationary `life_df1` data:

```
# Plot ACF and PACF to estimate p (AR order) and q (MA order)
fig, ax = plt.subplots(1,2)
plot_acf(life_df1, ax=ax[0])
plot_pacf(life_df1, ax=ax[1]);
```


It produces the following plots:

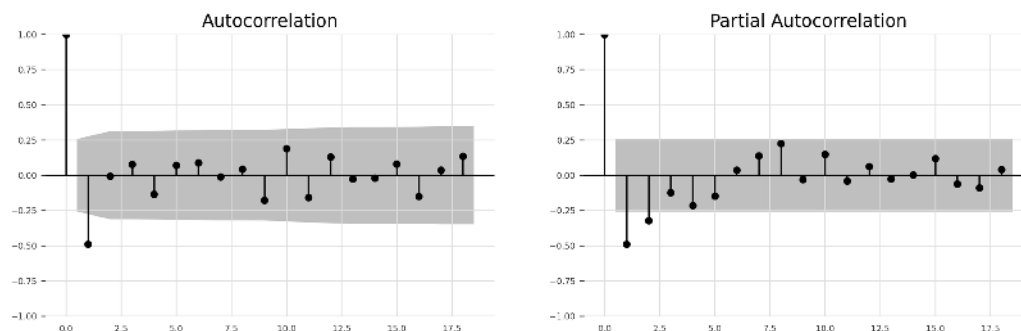


Figure 9.21: ACF and PACF plot for Life Expectancy data after differencing

When interpreting these plots (Figure 9.21), note that lag(0) is included to provide a reference point for comparison with other (past) lags. The ACF and PACF at lag(0) are always one; they are sometimes omitted from plots because they do not provide any meaningful information for model selection. Therefore, it's more important to focus on lag(1) and subsequent lags to determine their significance.



Identifying significant lags through ACF and PACF plots serves a crucial purpose in model development. These plots help you determine which lags have meaningful relationships with the current observation, allowing you to build an ARIMA model that captures the data's essential patterns without becoming unnecessarily complex. This balance is vital because including too many lags can lead to overfitting, where the model performs well on training data but fails to generalize to new, unseen data (e.g., test set). By focusing only on statistically significant lags, you create a model that strikes the right balance between capturing meaningful patterns and maintaining simplicity.

The ACF plot helps identify significant lags for the MA(q) component. The ACF plot shows a significant spike at lag 1 followed by a sharp cut-off, which is a characteristic pattern indicating an MA(1) model. Conversely, the PACF plot helps determine the significant lags for the AR(p) component. You can observe a gradual decay with oscillation after lag 1 in the PACF plot, combined with the sharp cut-off in the ACF, which confirms an MA(1) process rather than an AR process. This indicates a lack of an AR process, so the p order should be zero, resulting in AR(0). Please refer to Table 9.1 for more details on interpreting these patterns.

An MA(1) process, also called a first-order MA process, indicates that the current value (at time t) is influenced by the forecast error from the immediately preceding period (at time $t-1$).

4. Now, you can construct an ARIMA(p , d , q) model using $p=0$, $q=1$, and $d=1$ specifications, resulting in ARIMA(0,1,1). Often, the optimal lag values (orders) for p and q are not immediately apparent, so you will need to experiment by evaluating multiple ARIMA models using different parameters. This can be achieved through approaches such as grid search, similar to the method described and used in the *Forecasting univariate time series data with exponential smoothing* recipe:

```
# Initialize the ARIMA model
model = ARIMA(life_train, order=(0,1,1))
```

Train the ARIMA model on the training set (`life_train`) and inspect the model's summary. It is important not to use the previously differenced dataset (`life_df1`) because ARIMA internally applies differencing based on the d parameter. In this example, since first-order differencing was satisfactory to detrend and make the data stationary, you set $d=1$ in the model initialization:

```
# Fit the ARIMA model and display the summary
results = model.fit()
results.summary()
```

You will see the summary as shown in Figure 9.22:

SARIMAX Results						
Dep. Variable:		value		No. Observations:		50
Model:		ARIMA(0, 1, 1)		Log Likelihood		-24.161
Date:		Tue, 11 Nov 2025		AIC		52.321
Time:		20:27:51		BIC		56.105
Sample:		01-01-1960		HQIC		53.757
- 01-01-2009						
Covariance Type:		opg				
	coef	std err	z	P> z	[0.025	0.975]
ma.L1	0.0827	0.200	0.413	0.680	-0.310	0.475
sigma2	0.1569	0.032	4.918	0.000	0.094	0.219
Ljung-Box (L1) (Q):		12.54	Jarque-Bera (JB):		0.56	
Prob(Q):		0.00	Prob(JB):		0.76	
Heteroskedasticity (H):		0.43	Skew:		0.07	
Prob(H) (two-sided):		0.10	Kurtosis:		3.51	

Figure 9.22: Summary of ARIMA(0,1,1) for the Life Expectancy data

Notice that the AIC and BIC scores are provided in the model summary to help assess model fit. Lower values generally indicate a better model, but these metrics are most meaningful when comparing multiple models trained on the same dataset. AIC and BIC metrics help in assessing model fit while penalizing excessive complexity.

In this example, the ARIMA model is primarily an MA process with an integration (differencing) factor of $d=1$. The summary table provides coefficient values for the $MA(1)$ component only, as there are no AR terms ($p=0$). More on that in the *How it works...* section.

Further, the model summary provides three diagnostic tests that assess different aspects of our model's residuals (for more information on these tests, refer to *Chapter 8*):

- The Ljung-Box test checks for autocorrelation in the residuals. The test score is 12.54, and $\text{Prob}(Q)$ (p-value) is < 0.05 , so we reject the null hypothesis of no autocorrelation.
- The Jarque-Bera test evaluates whether residuals follow a normal distribution. The test score is 0.56, and $\text{Prob}(JB)$ (p-value) > 0.05 , so we fail to reject the null hypothesis of normality, suggesting the residuals are approximately normal.
- Finally, the heteroskedasticity test checks whether the residuals have constant variance over time. The test score is 0.43, and $\text{Prob}(H)$ (p-value) > 0.05 , so we fail to reject the null hypothesis of homoskedasticity, suggesting our residuals have relatively consistent variability.

The presence of autocorrelation suggests there is room to improve our model.

5. In addition to the model summary, you can further analyze the model's residuals to validate whether the $ARIMA(0, 1, 1)$ model adequately captured the information (patterns) in the time series. In a well-fitted model, the residuals should be white noise (random) and not follow any systematic pattern. More specifically, you would expect the residuals to exhibit no autocorrelation.

You can perform the `acorr_ljungbox` statistical test followed by the ACF plot on the residuals. You will be using the standardized residuals (`standardized_forecasts_error`) instead of the raw residuals (`results.resid`):

```
# Extract standardized residuals
standardized_residuals = results.filter_results.standardized_forecasts_
error[0]

# Perform Ljung-Box test on standardized residuals
ljungbox_results = acorr_ljungbox(
    standardized_residuals, lags=25, return_df=True)
print(ljungbox_results)

# Count significant p-values (<0.05)
significant_lags = (ljungbox_results['lb_pvalue'] < 0.05).sum()
```

```
print(f"Number of significant lags: {significant_lags}")
>>
Number of significant lags: 24
```

The Ljung-Box test result shows that 24 out of 25 lags show significant autocorrelation (p-value < 0.05), indicating that our model hasn't captured all patterns in the data. This comprehensive test of residual autocorrelation suggests our model hasn't fully captured the temporal dependency in the data.

Next, visualize the residual autocorrelation with the ACF plot:

```
plot_acf(standardized_residuals, zero=False);
```

If the model performs well, you would expect no significant lags (spikes are within the shaded area). In other words, all the vertical lines should be closer to zero or at zero for all lags, indicating that the model captured all patterns in the data.

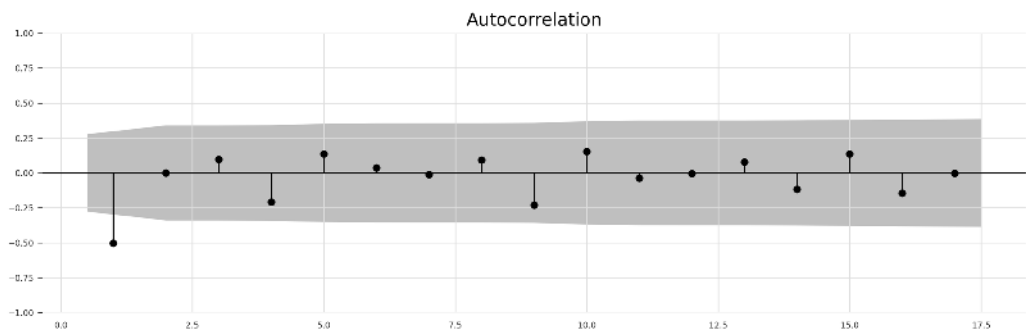


Figure 9.23: ACF plot showing no autocorrelation for the residuals

In Figure 9.23, you can see a significant spike at lag 1, indicating residual autocorrelation, further confirming that the model can be improved (e.g., adjusting the ARIMA parameters).

6. In addition to statistical tests, you should perform a visual inspection to further diagnose the residuals using a **Q-Q plot** and a **kernel density estimation (KDE)** plot to observe the distribution and assess normality.

Use the `plot_diagnostics` method to generate diagnostic plots:

```
results.plot_diagnostics(figsize=(12,8));
```

The following plots will be produced:

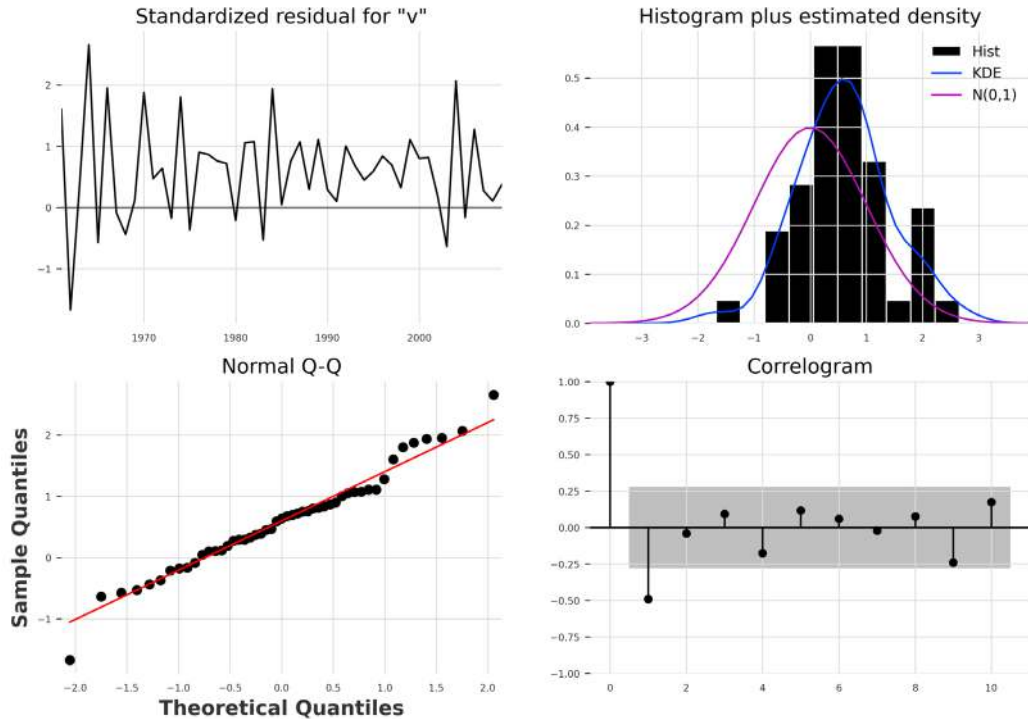


Figure 9.24: Visual diagnostics for the ARIMA(0,1,1) model

The diagnostic plots reveal several insights:

- The **standardized residual plot** checks for randomness, ensuring that there is no pattern in the residuals.
- The **histogram** and **KDE plot** show a slight deviation from a normal distribution. In a well-fitted model, you would expect a perfect bell-curved KDE with a mean of zero.
- The **Q-Q plot** shows a close alignment of points along the line but with visible departures in the tails, confirming slight non-normality. In a well-fitted model, all the points will be perfectly aligned on the line in the Q-Q plot.
- The **correlogram** should be similar to that in *Figure 9.23* and used to ensure that no significant spikes remain.

The overall diagnostics suggest that ARIMA(0, 1, 1) is a decent model that captures some of the underlying patterns, but there is room for improvement. Remember, building an ARIMA model is often an iterative process involving multiple rounds of model specification and diagnostics to achieve the best results.

- The final step in the ARIMA modeling process is to forecast future values and compare these predictions with your test dataset, which represents unseen or out-of-sample data. Use the `plot_forecast()` function, which you created earlier in this chapter in the *Technical requirements* section:

```
plot_forecast(results, '1998', life_train, life_test)
```

When executed, this function will generate a plot where the x axis starts from the year 1998. The plot will display three lines: the actual data is split into two lines, one for the training data (`orig_train`) and another for the test data (`orig_test`), and a third line for the forecast (predicted values).

This visual representation helps in assessing how well the ARIMA model has captured the underlying patterns of the dataset and how accurately it forecasts future values:

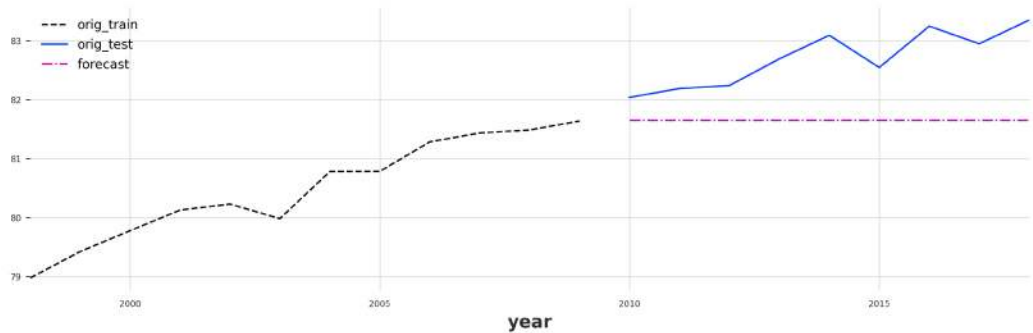


Figure 9.25: ARIMA(0,1,1) forecast versus the actual Life Expectancy data

The dashed line (forecast) doesn't seem to align with the expected trend. This contrasts with the results from the exponential smoothing model shown in *Figure 9.12*, which demonstrated better performance in capturing the data's trend. To address this, you can explore several ARIMA models with different parameter combinations (p , d , q). By evaluating their performance metrics, such as RMSE and MAPE, along with information criteria such as AIC and BIC, you can identify the model specifications that best capture the data's underlying patterns. You will explore this option in the *There's more...* section.

How it works...

An autoregressive model ($AR(p)$) is a linear model that uses observations from previous time steps as inputs into a regression equation to determine the predicted value of the next step. Hence, the *auto* in *autoregression* refers to *self*, and can be described as the regression of a variable on past versions of itself. A typical linear regression model will have this equation:

$$Y = m + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n + \epsilon$$

Here, y is the predicted variable, m is the intercept, $(x_1, x_2, \dots, x_n)$ are the features or independent variables, and $(\theta_1, \theta_2, \dots, \theta_n)$ are the **coefficients** for each of the independent variables. In regression, your goal is to solve for these coefficients, including the intercept (think of them as weights), since they are later used to make predictions. The error term, ϵ , denotes the residual or noise (the unexplained portion of the model).

Compare that with the autoregressive equation, and you will see the similarities:

$$AR(p) = y_t = \alpha + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \dots + \phi_p y_{t-p} + \epsilon_t$$

This can be written as follows:

$$y_t = \alpha + \sum_{j=1}^p \phi_j Y_{t-j} + \epsilon_t$$

This is an AR model of order p written as $AR(p)$. The main difference between an autoregressive and a regression model is that the predicted variable is y_t (y at the current time t), and the independent variables $y_{t-1}, y_{t-2}, \dots, y_{t-p}$ are lagged (previous) versions of y itself.

Unlike an autoregressive model that uses past values, the MA or $MA(q)$ uses past forecast errors to make a prediction:

$$MA(q) = y_t = \beta + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \dots + \theta_q \epsilon_{t-q} + \epsilon_t$$

This can be written as follows:

$$y_t = \beta + \sum_{j=1}^q \theta_j \epsilon_{t-j} + \epsilon_t$$

Combining the $AR(p)$ and $MA(q)$ models produces an $ARMA(p,q)$ model (autoregressive moving average), written as follows:

$$y_t = c + \sum_{j=1}^p \phi_j Y_{t-j} + \sum_{j=1}^q \theta_j \epsilon_{t-j} + \epsilon_t$$

Both the AR and ARMA processes assume a stationary time series. However, if the time series is not stationary due to the presence of a trend, you cannot use the AR or ARMA models directly unless you perform some transformations, such as differencing.

The ARIMA model improves on the ARMA model by incorporating this differencing operation through its integrated (d) component to handle non-stationary data. Differencing involves subtracting each value from its previous (lagged) value to remove the trend. For example, a first-order differencing operation can be written as $y_t - y_{t-1}$. In pandas, you can apply differencing using the `diff` method, which defaults to `periods=1`.

The Life Expectancy data was non-stationary due to the presence of the trend, hence we had to use an ARIMA model. More specifically, you applied an $ARIMA(0, 1, 1)$ model, which can be interpreted as follows:

- $AR(0)$: This indicates that no autoregressive terms are included in the model
- $I(1)$: First-order differencing was used to remove the trend and make the data stationary
- $MA(1)$: A first-order moving average component ($q=1$) was used to model the relationship between current observations with past residual errors

To determine the appropriate values for the p (AR order) and q (MA order) parameters, you leveraged both ACF and PACF plots. The ACF measures the correlation between a current observation and its lagged version. The purpose of the ACF plot is to determine how reliable past observations are in making predictions and is used to identify how many lagged residuals (errors) should be included in an $MA(q)$ model.

On the other hand, the PACF measures autocorrelation at specific lags but with the influences of intervening observations removed. The PACF plot helps determine how many lagged observations should be included in an $AR(p)$ model.

ACF versus PACF through an example



Consider a time series where strong correlations are observed at lags 1, 2, 3, and 4 in the ACF. When examining the ACF, the correlation measure at lag 1 is influenced by indirect correlations through lag 2, which, in turn, is influenced by lag 3, creating a chain of dependencies. The ACF measure at lag 1 includes these indirect influences from all subsequent lags.

In contrast, the PACF removes these cascading effects to reveal only the direct correlation between the current observation and each lag, after controlling for the intermediate lags. This “pure” relationship helps in determining the appropriate order (p) for the autoregressive component of your model.

ARIMA's popularity stems from its ability to generalize other series models:

- $ARIMA(1, 0, 0)$ represents a first-order $AR(1)$ model
- $ARIMA(1, 1, 0)$ represents a differenced first-order AR model
- $ARIMA(0, 0, 1)$ represents a first-order $MA(1)$ model
- $ARIMA(1, 0, 1)$ represents an ARMA $(1, 1)$ model
- $ARIMA(0, 1, 1)$ represents a simple exponential smoothing model under specific parameter constraints

There's more...

Identifying whether the time series follows an MA or AR process and determining the optimal order values (p , q) can be challenging, even with ACF and PACF analysis. To address this, you perform a grid search to evaluate multiple ARIMA specifications to find the best-fitting model.

Here, you will leverage the `combinator()` function created earlier in the *Technical requirements* section to generate combinations of (p , d , q). You will train multiple ARIMA models and use `get_top_models_df()` to find the best-performing model.

As an example, you'll test the 0, 1, and 2 values for each hyperparameter (p , d , and q), resulting in evaluating $3 \times 3 \times 3 = 27$ ARIMA models:

```
# Create lists [0,1,2] for p, d, and q
pv, dv, qv = [list(range(3))]*3
# Generate all possible combinations
vals = combinator([pv, dv, qv ])

# Initialize a dictionary to store results
score = {}

# Train multiple ARIMA models
for i, (p, d, q) in enumerate(vals):
    m = ARIMA(life_train, order=(p,d,q))
    res = m.fit()
    y = life_train.values.ravel()
    y_hat = res.forecast(steps=len(y))
    # Store multiple performance metrics for comparison
    score[i] = {'order': (p,d,q),
               'AIC':res.aic,
               'RMSPE': rmspe(y, y_hat),
               'BIC': res.bic,
               'AICc':res.aicc,
               'RMSE' : rmse(y, y_hat),
               'MAPE' : mape(y, y_hat),
               'model': res}

# Display top 5 models sorted by AIC
top_5_models = get_top_models_df(score, 'AIC')
top_5_models
```

This will produce a DataFrame of the top 5 models (default, `top_n=5`) sorted by AIC:

	order	AIC	RMSPE	BIC	AICc	RMSE	MAPE	model
model_id								
8	(0, 2, 2)	7.697707	1.654410	13.311310	8.243162	12.370909	0.164673	<statsmodels.tsa.arima.model.ARIMAResultsWrapp...
14	(1, 1, 2)	9.533199	1.638446	17.100481	10.442290	12.239986	0.162932	<statsmodels.tsa.arima.model.ARIMAResultsWrapp...
17	(1, 2, 2)	9.675660	1.653313	17.160464	10.605893	12.362193	0.164557	<statsmodels.tsa.arima.model.ARIMAResultsWrapp...
26	(2, 2, 2)	10.671686	1.663602	20.027691	12.100257	12.444132	0.165643	<statsmodels.tsa.arima.model.ARIMAResultsWrapp...
23	(2, 1, 2)	11.510115	1.636793	20.969217	12.905464	12.226714	0.162755	<statsmodels.tsa.arima.model.ARIMAResultsWrapp...

Figure 9.26: Results from the 27 ARIMA models sorted by AIC score

You can select the best-performing model :

```
best_m = top_5_models.iloc[0, -1]
```

Running `best_m.summary()` displays the summary of the top model, which, in this case, is `ARIMA(0, 2, 2)` according to AIC. This result is interesting for two reasons:

- It confirms our earlier observation that dominance of an MA component ($p=0$).
- It suggests that our initial choice of $d=1$ might not have been optimal, as double differencing ($d=2$) appears to work better for this dataset. For `ARIMA(0, 1, 1)`, $AIC=52.32$, while for `ARIMA(0, 2, 2)`, $AIC=7.69$.

AIC is a metric that aims to find a balance between a model's goodness of fit and its complexity. While more complex models can appear to learn the patterns better, they risk overfitting, meaning they perform well on training data but poorly once they are presented with new, unseen data. The AIC score includes a penalty term that increases as the number of parameters increases (increase in complexity):

$$AIC = 2k - 2 \ln(L^*)$$

Here, $2k$ is considered the penalty term:

- k : Represents the number of estimated parameters in the model
- L^* : Maximum likelihood of the model.

BIC is very similar to AIC but imposes a stronger penalty term for model complexity. In general, the BIC penalty term is more significant, so it favors models with fewer parameters than AIC. Therefore, if you change the sorting or evaluation criteria from AIC to BIC, you may see different results. Simpler models are preferred more with BIC:

$$BIC = K \ln(n) - 2 \ln(L^*)$$

Let's break down the formula:

- k : Number of estimated parameters
- L^* : Maximum likelihood of the model
- n : Number of data points

To plot a forecast using the best model identified through grid search, use the following command:

```
plot_forecast(best_m, '1998', life_train, life_test);
```

This should produce the following plot:

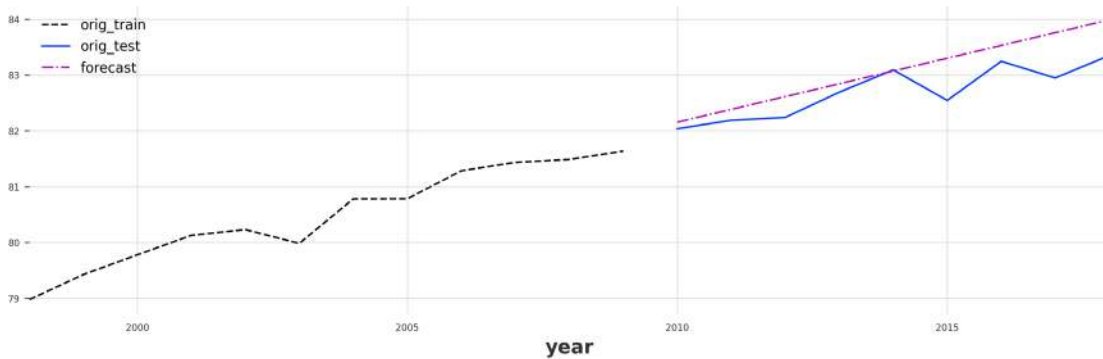


Figure 9.27: ARIMA(0,2,2) forecast versus the actual Life Expectancy data

Compare the output in Figure 9.25 for the ARIMA(0, 1, 1) model with Figure 9.27 for the ARIMA(0, 2, 2) model. How do they compare with Figure 9.12 using exponential smoothing? The ARIMA(0, 2, 2) model captured the trend better than the ARIMA(0, 1, 1) model.

To further evaluate the ARIMA(0, 2, 2) model, inspect its summary and diagnostic:

```
best_m.summary()
best_m.plot_diagnostics(figsize=(12,8));
```

Now, let's apply the grid search method to the *Milk Production* dataset. Unlike the *Life Expectancy* data, this dataset exhibits both trend and seasonality, which requires models capable of handling such patterns:

```
# Create Lists [0,1,2] for p, d, and q
pv, dv, qv = [list(range(3))]*3
# Generate all possible combinations
vals = combinator([pv, dv, qv])
# Initialize a dictionary to store results
score = {}
# Train multiple ARIMA models
for i, (p, d, q) in enumerate(vals):
    m = ARIMA(milk_train, order=(p,d,q))
    res = m.fit()
    y = milk_test.values.ravel()
    y_hat = res.forecast(steps=len(y))
    # Store multiple performance metrics for comparison
    score[i] = {'order': (p,d,q),
```

```

        'AIC':res.aic,
        'BIC': res.bic,
        'AICc':res.aicc,
        'RMSPE': rmspe(y, y_hat),
        'RMSE' : rmse(y, y_hat),
        'MAPE' : mape(y, y_hat),
        'model': res}

milk_top_models = get_top_models_df(score, 'AIC')
milk_best_model = milk_top_models.iloc[0,-1]

```

Inspect the best ARIMA model for the *Milk Production* dataset:

```

milk_best_model.summary()
milk_best_model.plot_diagnostics(figsize=(12,8));

```

The grid search identified ARIMA(2,2,2) as the optimal model based on AIC. Let's visualize its forecasting performance:

```

plot_forecast(milk_best_model, '1971', milk_train, milk_test);

```

The resulting forecast plot will likely show poor predictions overall, as depicted in *Figure 9.28* – forecasting against actual *Milk Production* data:

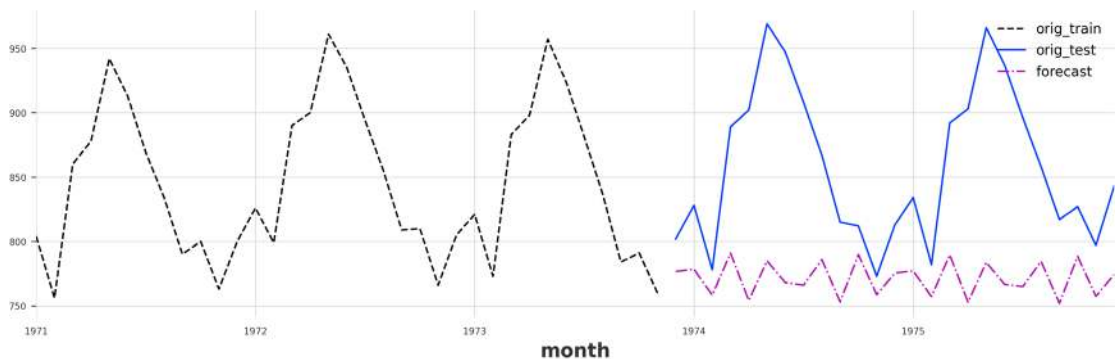


Figure 9.28: ARIMA(2,2,2) forecast versus the actual Milk Production data

This outcome is expected since standard ARIMA models are not designed to handle seasonality. This reinforces the limitation of ARIMA for datasets with seasonal patterns.

The next recipe will introduce SARIMA, which is better equipped to model time series data with both seasonal patterns and trends.



Forecast versus predict methods

In `statsmodels`, the SARIMA family of models (ARMA and ARIMA) provides two methods for making predictions: `predict` and `forecast`. The `predict` method makes both **in-sample** (historical) and **out-of-sample** (future) predictions by specifying start and end parameters. The `forecast` method is simpler, taking only a `steps` parameter for the number of out-of-sample forecasts needed beyond the training set.

Choose `predict` when you need predictions for specific date ranges or want dynamic forecasts. Use `forecast` when you only need future values beyond your training data. For confidence intervals, use the `get_forecast()` method instead.

In our `plot_forecast` function, we used the `forecast` method since we focused solely on generating future predictions beyond the training period.

See also

- To learn more about the ARIMA class, you can visit the official `statsmodels` documentation at <https://www.statsmodels.org/dev/generated/statsmodels.tsa.arima.model.ARIMA.html>.

Forecasting univariate time series data with seasonal ARIMA

In this recipe, you will be introduced to an enhancement to the ARIMA model for handling seasonality, known as SARIMA. While ARIMA models are effective for time series data with trends, they fall short when seasonality is present. SARIMA addresses this limitation by incorporating seasonal components into the model.

Similar to an $ARIMA(p, d, q)$ model, a SARIMA model requires the non-seasonal orders (p, d, q) . However, SARIMA extends ARIMA by incorporating the seasonal component, which is denoted as (P, D, Q, s) .

Combining both non-seasonal and seasonal components, the model is expressed as $SARIMA(p, d, q, P, D, Q, s)$:

- Lowercase (p, d, q) represents non-seasonal orders
- Uppercase (P, D, Q) represents seasonal orders
- The new parameter, s , is the number of time steps per seasonal cycle (e.g., $s=12$ for monthly data or $s=4$ for quarterly data)

In `statsmodels`, you will use the `SARIMAX` class to build a SARIMA model.

In this recipe, you will be working with the *Milk Production* dataset, which exhibits both trend and seasonality. This was prepared in the *Technical requirements* section.

How to do it...

1. Start by importing the necessary libraries:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

2. From the dataset characteristics (Figure 9.1), you can observe both trend and seasonality. The seasonal effect is additive with a periodicity of 12 months (seasonal cycles). This can be confirmed using an ACF plot:

```
# Examine ACF on original data
plot_acf(milk, lags=40, zero=False);
```

This should produce an ACF plot for the milk data with a noticeable cyclical pattern of significant spikes at specific intervals (lags 1, 12, and 24):

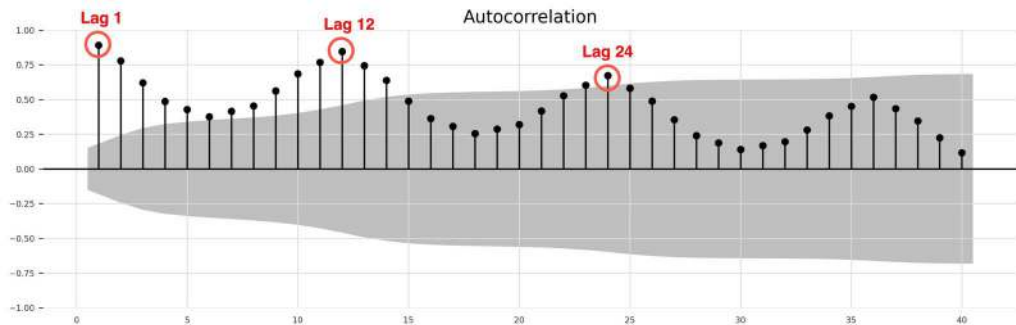


Figure 9.29: ACF plot showing significant spikes at lags 1, 12, and 24

There is a repeating pattern every 12 months (lags). If the pattern is not easy to spot, you can try the ACF plot after you difference the data – for example, detrend with first-order differencing, then plot the ACF:

```
plot_acf(milk.diff(1).dropna(), lags=40, zero=False);
```

This resulting ACF plot (Figure 9.30) highlights the seasonal spikes and makes them more apparent at specific intervals (lags 1, 12, 24, and 36):

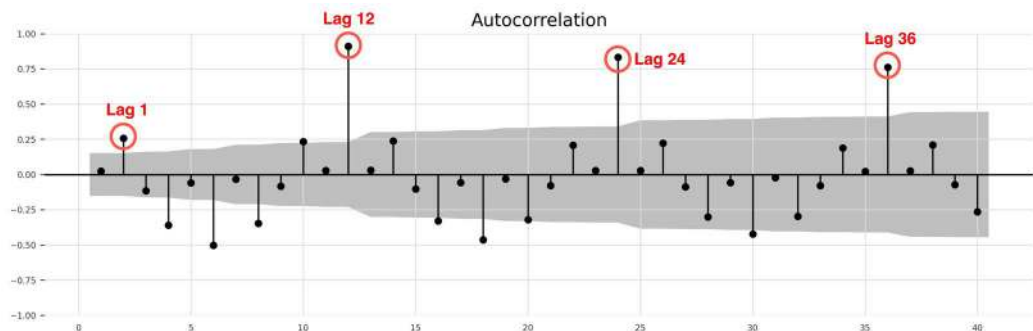


Figure 9.30: ACF plot after differencing shows significant spikes at lags 1, 12, 24, and 36

Alternatively, you can decompose the time series and extract the seasonal component to be used for the ACF plot:

```
decomposed = seasonal_decompose(milk,
                                period=12,
                                model='additive')

milk_s = decomposed.seasonal
plot_acf(milk_s, zero=False, lags=40);
```

The ACF plot will show the autocorrelation using the seasonal component after decomposition and will tell a similar story to what's shown in *Figure 9.29* and *Figure 9.30*.

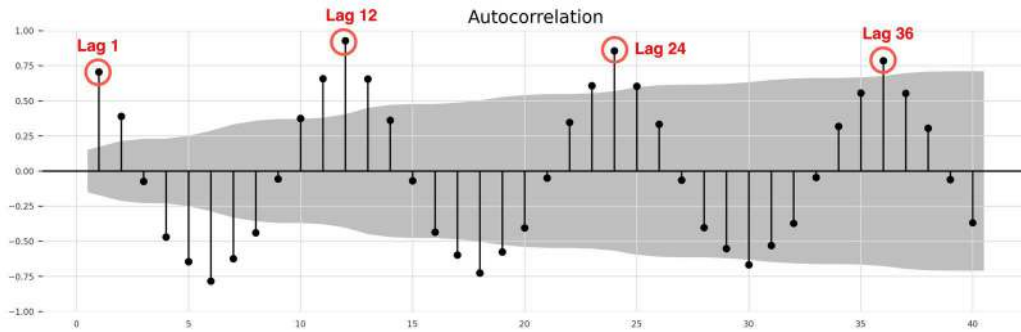


Figure 9.31: ACF on the seasonal component after decomposition shows significant spikes at lags 1, 12, 24, and 36

Generally, you can assume a 12-month cycle when working with monthly data. For example, for the non-seasonal ARIMA portion, start with $d=1$ for detrending, and for the seasonal ARIMA portion, start with $D=1$ as well, given that $s=12$.

3. If you are not sure about the values for d (non-seasonal differencing) and D (seasonal differencing), you can use the `check_stationarity` function after differencing to determine whether stationarity was achieved (e.g., whether seasonal differencing was enough or not). In most cases, if the time series has both trend and seasonality, you may need to difference twice. First, you perform seasonal differencing, followed by a first-order differencing for detrending.

Start with seasonal differencing by using `diff(12)` (deseasonalize) and test whether that is enough to make the time series stationary. If not, then you will need to follow it with a first-order differencing, `diff()`:

```
milk_dif_12 = milk.diff(12).dropna()
milk_dif_12_1 = milk.diff(12).diff(1).dropna()
sets = [milk, milk_dif_12, milk_dif_12_1]
desc = ['Original', 'Deseasonalize', 'Deseasonalize and Detrend']
fig, ax = plt.subplots(2,2, figsize=(20,10))
index, l = milk.index, milk.shape[0]
```

```

for i, (d_set, d_desc) in enumerate(zip(sets, desc)):
    v, r = i // 2, i % 2
    outcome, pval = check_stationarity(d_set)
    d_set.plot(ax= ax[v,r], title=f'{d_desc}: {outcome}',
               legend=False)
    pd.Series(
        d_set.mean().values.tolist()*1, index=index
    ).plot(ax=ax[v,r])
    ax[v,r].title.set_size(20)
ax[1,1].set_visible(False)
plt.show()

```

This should produce three plots:

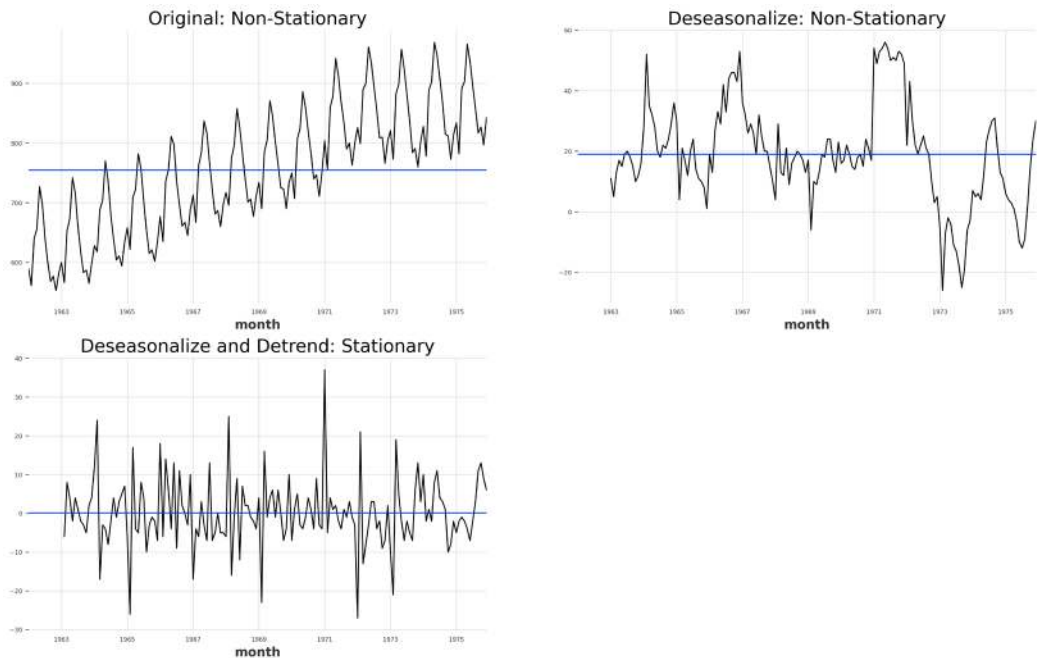


Figure 9.32: Stationarity comparison for original, seasonally differenced, and differenced twice time series

The comparison in Figure 9.32 shows that seasonal differencing ($D=1$) alone might not suffice, and combining it with first-order differencing ($d=1$) achieves stationarity.

4. Use the ACF and PACF plots on the stationary `milk_dif_12_1` DataFrame to estimate the AR and MA orders for the non-seasonal (p, q) and seasonal (P, Q) components:

```

fig, ax = plt.subplots(1,2)
plot_acf(milk_dif_12_1,
         zero=False,

```



```
lags=36,
ax=ax[0],
title=f'ACF - {d_desc}')
plot_pacf(milk_dif_12_1,
          zero=False,
          lags=36,
          ax=ax[1],
          title=f'PACF - {d_desc}');
```

This should produce ACF and PACF plots on the same row:

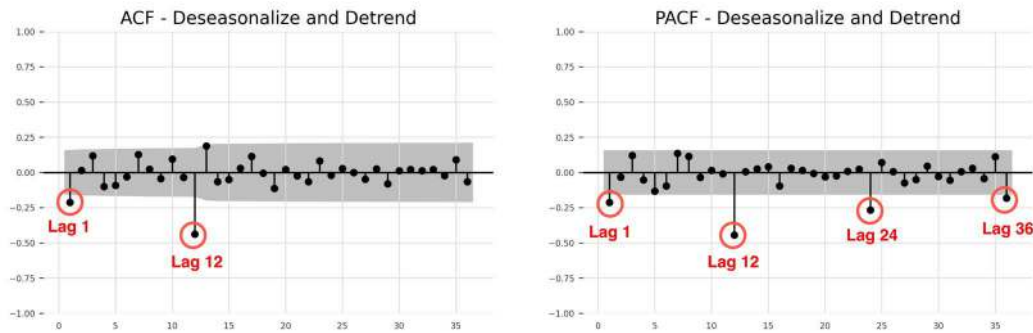


Figure 9.33: ACF and PACF plots for the Milk Production data after becoming stationary

Starting with the ACF plot, there is a significant spike at lag 1, which represents the non-seasonal order for the MA process. The spike at lag 12 represents the seasonal order for the MA process. Notice that there is a cut-off right after lag 1, then a spike at lag 12, followed by another cut-off (no other significant lags afterward). These are indications of an MA model – more specifically, an order of $q=1$ and $Q=1$.

The PACF plot confirms this as well; an exponential decay at seasonal lags 12, 24, and 36 indicates an MA model. The clear patterns in both ACF and PACF plots give us high confidence in selecting $\text{SARIMA}(0,1,1)(0,1,1,12)$.

- Using the values determined earlier ($p=0$, $d=1$, $q=1$ and $P=0$, $D=1$, $Q=1$, $s=12$), fit a SARIMA model on the training dataset:

```
sarima_model = SARIMAX(milk_train,
                        order=(0,1,1),
                        seasonal_order=(0,1,1,12))

# disp=0 suppresses optimization output
model = sarima_model.fit(disp=0)
```

- Once the model is fit, use the `plot_diagnostics` method to visualize its residuals and assess performance:

```
model.plot_diagnostics(figsize=(15,7));
```

The following plots will be produced:

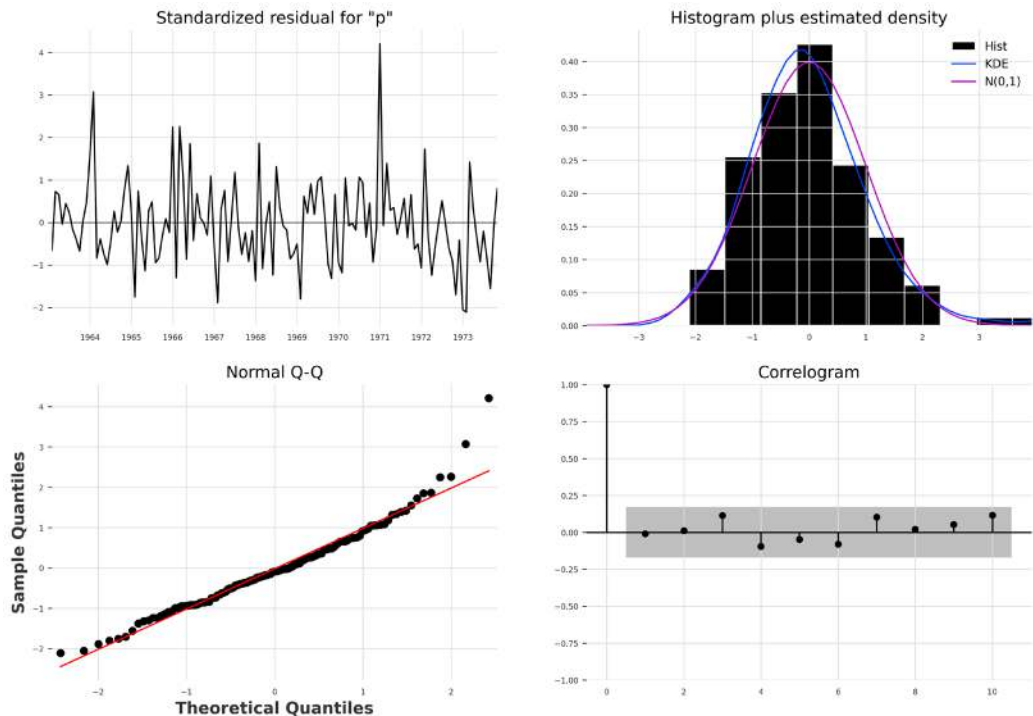


Figure 9.34: SARIMA(0,1,1)(0,1,1,12) diagnostic plots

Let's analyze the diagnostic plots from the SARIMA model:

- Standardized residuals appear to be centered around zero and random
- The histogram with KDE looks approximately normally distributed, with minor deviations in the tails
- The Q-Q plot shows the points being mostly aligned with the diagonal line, indicating near-normality, although there are slight deviations at the tails
- The correlogram (residual ACF) does not show any significant spike beyond lag 0, which indicates no autocorrelation

Overall, the results show that the model successfully captured both trend and seasonality patterns with minor deviations from normality.

Further, you can examine the model summary to complement the diagnostic plots with additional statistical tests. You can obtain the summary using the summary method:

```
model.summary()
```

SARIMAX Results						
Dep. Variable:	production			No. Observations:	143	
Model:	SARIMAX(0, 1, 1)x(0, 1, 1, 12)			Log Likelihood	-448.668	
Date:	Mon, 03 Nov 2025			AIC	903.335	
Time:	22:59:48			BIC	911.938	
Sample:	01-01-1962			HQIC	906.831	
- 11-01-1973						
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
ma.L1	-0.2811	0.084	-3.331	0.001	-0.446	-0.116
ma.S.L12	-0.6334	0.083	-7.653	0.000	-0.796	-0.471
sigma2	55.5239	5.753	9.650	0.000	44.247	66.800
Ljung-Box (L1) (Q):	0.01	Jarque-Bera (JB):	35.77			
Prob(Q):	0.92	Prob(JB):	0.00			
Heteroskedasticity (H):	1.15	Skew:	0.82			
Prob(H) (two-sided):	0.65	Kurtosis:	4.98			

Figure 9.35: SARIMA(0,1,1)(0,1,1,12) model summary

This should print out additional information regarding the model in a tabular format, including the coefficients for the seasonal and non-seasonal components. Let's examine the model summary key metrics:

- Residuals show no significant autocorrelation in the residuals (Ljung-Box p-value = 0.92)
- There is a mild deviation from normality in the residuals (JB = 35.77, p-value = 0.00)
- Homoscedastic residuals (H=1.15, p-value = 0.65)
- Slight right skewness (0.82) and heavy tails (kurtosis = 4.98)

7. Use the `plot_forecast` function to generate a forecast and compare it with the test set:

```
plot_forecast(model, '1971', milk_train, milk_test);
```

This should produce a plot with the x axis starting from the year 1971:

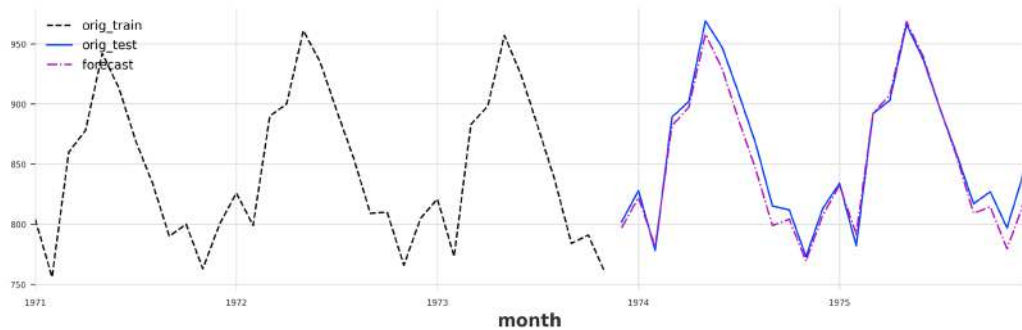


Figure 9.36: Milk Production forecast versus actual production using $SARIMA(0,1,1)(0,1,1,12)$

The SARIMA model successfully captured both seasonal patterns and the overall trend in the milk production data. You can always iterate and test different parameter values for (p, q, P, Q) and use evaluating metrics such as RMSE, MAPE, or AIC, to name a few. More on that in the *There's more...* section.

How it works...

The SARIMA model extends the ARIMA model by incorporating seasonal components, requiring an additional set of parameters (P, D, Q, s) .

In `statsmodels`, the `SARIMAX` class generalizes simpler models such as AR, MA, ARMA, and ARIMA. For example, when seasonal components are zero, as in $SARIMA(1,1,1)(0,0,0,0)$, it reduces to the $ARIMA(1, 1, 1)$ model. The seasonal component essentially applies ARIMA operations at seasonal lags. For example, with monthly data ($s=12$), a seasonal $AR(1)$ term relates each observation to the value 12 months prior, helping capture yearly patterns.

This is how SARIMA handles both trend and seasonality:

- Regular differencing (d) removes trend, while seasonal differencing (D) removes seasonal patterns
- Regular AR/MA terms (p, q) capture short-term dependencies
- Seasonal AR/MA terms (P, Q) capture patterns that repeat every s periods

This explains why our $SARIMA(0,1,1)(0,1,1,12)$ model worked well with the *Milk Production* data – it addresses the following:

- The upward trend using regular differencing ($d=1$)
- Yearly seasonal patterns using seasonal differencing ($D=1, s=12$)
- Capturing residual dependencies using both regular, $MA(q=1)$, and seasonal, $MA(Q=1)$

SARIMA's ability to handle seasonality makes it a robust extension of ARIMA for time series with both trend and recurring seasonal behaviors.

There's more...

Similar to the approach in the *Forecasting univariate time series data with ARIMA* recipe, you can execute a grid search to evaluate various combinations of the non-seasonal (p , d , q) and seasonal (P , D , Q , s) parameters to identify the best SARIMA model.

This can be done by leveraging the `combinator()` function to loop through all possible parameter combinations, fitting a SARIMA model at each iteration. Use the `get_top_models_df` function to rank the models based on evaluation metrics such as AIC, BIC, or RMSE.

For instance, consider testing the following:

- **Non-seasonal parameters** (p , d , q): Values (0, 1, 2)
- **Seasonal parameters** (P , D , Q): Values (0, 1)
- **Seasonal period** (s): Fixed at 12 (monthly data)

This setup results in a total of $3 \times 3 \times 3 \times 2 \times 2 \times 2 = 216$ SARIMA models. While this brute-force approach can be computationally intensive, it is effective for parameter optimization. Automated time series libraries, such as `Auto_ARIMA`, often employ similar exhaustive grid searches to identify optimal model parameters:

```
# Create Lists [0,1,2] for p, d, and q
P_ns, D_ns, Q_ns = [list(range(3))]*3
# Create Lists [0,1] for P, D, and Q
P_s, D_s, Q_s = [list(range(2))]*3
# Generate all possible combinations
vals = combinator([P_ns, D_ns, Q_ns, P_s, D_s, Q_s])
# Initialize a dictionary to store results
score = {}
# Train multiple SARIMA models
for i, (p, d, q, P, D, Q) in enumerate(vals):
    if i%15 == 0:
        print(f'Running model #{i} using SARIMA({p},{d},{q})({P},{D},{Q},12)')
    m = SARIMAX(milk_train,
                order=(p,d,q),
                seasonal_order=(P, D, Q, 12),
                enforce_stationarity=False)
    res = m.fit(dis=0)
    y = milk_test.values.ravel()
    y_hat = res.forecast(steps=len(y))
    score[i] = {'non-seasonal order': (p,d,q),
               'seasonal order': (P, D, Q),
```

```

'AIC': res.aic,
'AICc': res.aicc,
'BIC': res.bic,
'RMSPE': rmspe(y, y_hat),
'RMSE': rmse(y, y_hat),
'MAPE': mape(y, y_hat),
'model': res}

```

After executing the previous code, it should print a status output every 15 iterations.

Notice the `enforce_stationarity=False` parameter to avoid a `LinAlgError` error that may occur when running a grid search.

Use the `get_top_models_df` function to rank the models by a chosen metric, such as sorting by AIC:

```
get_top_models_df(score, 'AIC')
```

	non-seasonal order	seasonal order	AIC	AICc	BIC	RMSPE	RMSE	MAPE
model_id								
211	(2, 2, 2)	(0, 1, 1)	795.421765	796.206812	811.838956	0.178190	15.135862	0.014454
67	(0, 2, 2)	(0, 1, 1)	795.515920	795.882892	806.460713	0.328240	27.797214	0.028315
43	(0, 1, 2)	(0, 1, 1)	796.763902	797.127539	807.743631	0.134089	11.487077	0.010387
187	(2, 1, 2)	(0, 1, 1)	796.876071	797.653849	813.345664	0.128260	10.998611	0.010075
139	(1, 2, 2)	(0, 1, 1)	797.117088	797.672643	810.798080	0.289726	24.535286	0.024769

Figure 9.37: Top five SARIMA models ranked by AIC for the Milk Production data

Notice that the first two models have similar AIC scores. Generally, when two models have similar AIC scores, the simpler model is preferred. For example, a $\text{SARIMA}(0, 2, 2)(0, 1, 1)$ model is simpler than a $\text{SARIMA}(2, 2, 2)(0, 1, 1)$ model. Other metrics, such as AICc and BIC, impose a stronger penalty for model complexity than AIC. In this example, BIC would favor the second model (`model_id=67` with $\text{BIC}=806.45$) over the first model (`model_id=211` with $\text{BIC}=811.84$). Similarly, if you consider RMSPE or MAPE, they may favor different models.



Occam's Razor

Occam's Razor is a principle suggesting that when multiple models produce similar quality (i.e., similarly good fit to the data), the simpler model should be preferred. This principle is especially valuable when evaluating multiple models (e.g., different SARIMA models). If several models emerge as strong candidates, generally, the simplest model is favored, assuming that all are equally likely candidates to begin with.

Sort the models based on BIC score:

```
get_top_models_df(score, 'BIC')
```

Examine the results displayed in Figure 9.38, which shows the top five SARIMA models ranked by BIC for the Milk Production data.

model_id	non-seasonal order	seasonal order	AIC	AICc	BIC	RMSPE	RMSE	MAPE
67	(0, 2, 2)	(0, 1, 1)	795.515920	795.882892	806.460713	0.328240	27.797214	0.028315
43	(0, 1, 2)	(0, 1, 1)	796.763902	797.127539	807.743631	0.134089	11.487077	0.010387
35	(0, 1, 1)	(0, 1, 1)	802.462674	802.676959	810.723444	0.134090	11.489677	0.010370
139	(1, 2, 2)	(0, 1, 1)	797.117088	797.672643	810.798080	0.289726	24.535286	0.024769
211	(2, 2, 2)	(0, 1, 1)	795.421765	796.206812	811.838956	0.178190	15.135862	0.014454

Figure 9.38: Top five SARIMA models ranked by BIC for the Milk Production data

Comparing Figures 9.38 and 9.37, note the appearance of the SARIMA(0,1,1)(0,1,1) model (model_id = 35), which was derived earlier.

Extract the best model based on the BIC score:

```
best_model = get_top_models_df(score, 'BCI').iloc[0,-1]
```

Lastly, you can visualize the model's forecast alongside the actual data using the plot_forecast function:

```
plot_forecast(best_model, '1971', milk_train, milk_test);
```

This should produce a plot with the x axis starting from the year 1962, as shown in Figure 9.39:

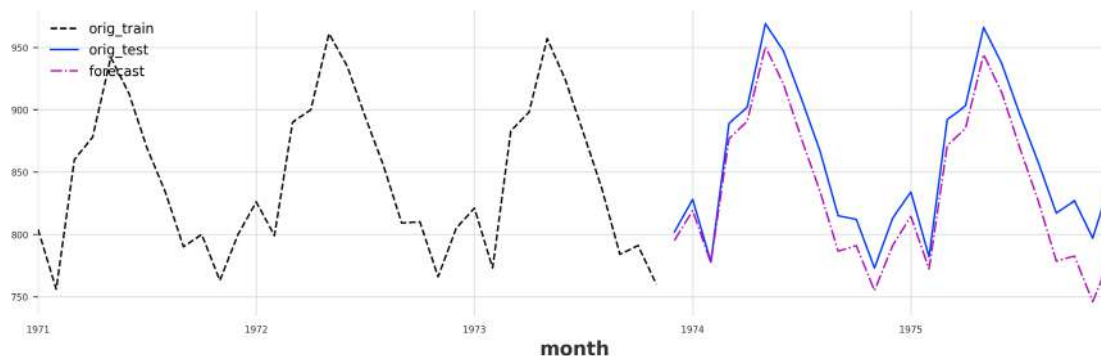


Figure 9.39: Milk Production forecast versus actual production using SARIMA(0,2,2)(0,1,1,12)

See also

- To learn more about the SARIMAX class, you can visit the official statsmodels documentation at <https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html>.

Forecasting univariate time series with `auto_arima`

For this recipe, you will need to install `pmdarima`, a Python library that includes `auto_arima` – a tool designed to automate the optimization and fitting of ARIMA models. The `auto_arima` implementation in Python is inspired by the popular `auto.arima` from the `forecast` package in R.

As you’ve seen in earlier recipes, determining the correct orders for the AR and MA components can be challenging. While techniques such as examining ACF and PACF plots are helpful, finding the optimal model often involves training multiple models – a process known as **hyperparameter tuning**, which can be quite labor-intensive. This is where `auto_arima` shines, as it simplifies the effort.

Instead of the naive brute-force approach of conducting a grid search to try every parameter combination, `auto_arima` uses a more efficient approach to finding the optimal parameters. The `auto_arima` function offers three approaches to finding optimal parameters:

- **Stepwise search:** The default, with `stepwise=True`, conducts a stepwise search that iteratively refines the model parameters. The **stepwise algorithm** is faster and more efficient than a full grid search or random search.
- **Exhaustive grid search:** When `stepwise=False`, it performs an exhaustive “brute-force” grid search across all parameter combinations. This approach takes the longest time, depending on the number of hyperparameters.
- **Random search:** With `random=True` and `stepwise=False`, it performs a random search by randomly sampling combinations from the parameter space. This can be faster than brute-force grid search and can be effective, especially when dealing with large datasets or when computational resources are limited.

The default stepwise algorithm was proposed in 2008 by Rob Hyndman and Yeasmin Khandakar in their paper, *Automatic Time Series Forecasting: The forecast Package for R*, published in the Journal of Statistical Software 27, no. 3 (2008) (<https://doi.org/10.18637/jss.v027.i03>). In a nutshell, stepwise is an optimization technique that combines **unit root tests**—namely, the **KPSS** test to determine the number of non-seasonal differencing steps (d) and the **Canova-Hansen (CH)** test to determine the number of seasonal differencing steps (D), with model selection based on information criteria (defaulting to AIC) and MLE.

Additionally, `auto_arima` supports both seasonal and non-seasonal ARIMA models. For seasonal models, setting `seasonal=True` enables the optimization of seasonal parameters (P , D , Q), similar to the approach explored in the previous recipe.

Getting ready

Before proceeding with this recipe, you will need to install the `pmdarima` library.

To install it using `pip`, use the following:

```
pip install pmdarima
```


The `.summary()` method provides a detailed configuration of the chosen SARIMA model, including information criteria scores such as AIC and BIC:

SARIMAX Results						
Dep. Variable:	y			No. Observations:	143	
Model:	SARIMAX(0, 1, 1)x(0, 1, 1, 12)			Log Likelihood	-448.668	
Date:	Tue, 11 Nov 2025			AIC	903.335	
Time:	21:36:24			BIC	911.938	
Sample:	01-01-1962			HQIC	906.831	
	- 11-01-1973					
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
ma.L1	-0.2811	0.084	-3.331	0.001	-0.446	-0.116
ma.S.L12	-0.6334	0.083	-7.653	0.000	-0.796	-0.471
sigma2	55.5239	5.753	9.650	0.000	44.247	66.800
Ljung-Box (L1) (Q):	0.01	Jarque-Bera (JB):	35.77			
Prob(Q):	0.92	Prob(JB):	0.00			
Heteroskedasticity (H):	1.15	Skew:	0.82			
Prob(H) (two-sided):	0.65	Kurtosis:	4.98			

Figure 9.40: Summary of the best SARIMA model selected using `auto_arima`

Interestingly, the selected model, $SARIMA(0, 1, 1)(0, 1, 1, 12)$, aligns with the one you derived in the *Forecasting univariate time series data with seasonal ARIMA* recipe (Figure 9.35), where you estimated the non-seasonal order (p, q) and seasonal orders (P, Q) using the ACF and PACF plots. This demonstrates that `auto_arima` can reliably automate the process of identifying optimal parameters.

- To monitor the performance of each model configuration evaluated during the stepwise search, enable the `trace=True` parameter in the `auto_arima` function:

```
auto_model = pm.auto_arima(milk_train,
                           seasonal=True,
                           m=12,
                           test='adf',
                           stepwise=True,
                           trace=True)
```

This setting will print the AIC results for each SARIMA model tested by the stepwise algorithm, as shown in Figure 9.41:

```

Performing stepwise search to minimize aic
ARIMA(2,1,2)(1,1,1)[12] : AIC=908.834, Time=0.33 sec
ARIMA(0,1,0)(0,1,0)[12] : AIC=951.320, Time=0.01 sec
ARIMA(1,1,0)(1,1,0)[12] : AIC=917.481, Time=0.04 sec
1 ARIMA(0,1,1)(0,1,1)[12] : AIC=903.335, Time=0.07 sec
ARIMA(0,1,1)(0,1,0)[12] : AIC=940.547, Time=0.01 sec
ARIMA(0,1,1)(1,1,1)[12] : AIC=905.280, Time=0.09 sec
ARIMA(0,1,1)(0,1,2)[12] : AIC=905.265, Time=0.24 sec
ARIMA(0,1,1)(1,1,0)[12] : AIC=917.295, Time=0.06 sec
ARIMA(0,1,1)(1,1,2)[12] : AIC=inf, Time=1.03 sec
ARIMA(0,1,0)(0,1,1)[12] : AIC=911.816, Time=0.04 sec
ARIMA(1,1,1)(0,1,1)[12] : AIC=905.240, Time=0.09 sec
2 ARIMA(0,1,2)(0,1,1)[12] : AIC=905.159, Time=0.13 sec
ARIMA(1,1,0)(0,1,1)[12] : AIC=903.959, Time=0.06 sec
ARIMA(1,1,2)(0,1,1)[12] : AIC=907.070, Time=0.26 sec
ARIMA(0,1,1)(0,1,1)[12] intercept : AIC=905.043, Time=0.07 sec

Best model: ARIMA(0,1,1)(0,1,1)[12]
Total fit time: 2.548 seconds

```

Figure 9.41: `auto_arma` evaluating different SARIMA models based on AIC

The best model was selected based on AIC, which is determined by the `information_criterion` parameter in `auto_arma`. By default, this is set to `aic`, but it can be changed to other supported criteria such as `bic`, `hqic`, or `oob`.

Looking at Figure 9.41, you can observe the following:

- The model marked as (1), `ARIMA(0,1,1)(0,1,1)[12]`, achieves an AIC of 903.335 with a quick fitting time of 0.07 seconds
- The model marked as (2), `ARIMA(1,1,0)(0,1,1)[12]`, has a slightly higher AIC of 903.959, taking 0.04 seconds to fit

These two models demonstrate an interesting trade-off in time series modeling:

- Model 1 relies solely on MA terms for the non-seasonal component
- Model 2 uses only AR terms for the non-seasonal component
- Both models maintain the same seasonal structure, `(0,1,1)[12]`

Despite their different approaches to capturing non-seasonal patterns, both models achieve similar AIC scores. This illustrates that multiple configurations can effectively capture the same underlying patterns in time series data.

While `auto_arma` significantly simplifies model selection by automating hyperparameter tuning and minimizing information criteria such as AIC or BIC, careful judgment is still required to interpret the results effectively.

- To explore how the choice of information criterion affects model selection, you can change the `information_criterion` parameter to `bic`. This applies a stricter penalty on model complexity and may favor simpler models:

```
auto_model = pm.auto_arima(milk_train,
                           seasonal=True,
                           m=12,
                           test='adf',
                           information_criterion='bic',
                           stepwise=True,
                           trace=True)
```

```
Performing stepwise search to minimize bic
ARIMA(2,1,2)(1,1,1)[12]          : BIC=928.907, Time=0.36 sec
ARIMA(0,1,0)(0,1,0)[12]        : BIC=954.187, Time=0.01 sec
ARIMA(1,1,0)(1,1,0)[12]        : BIC=926.083, Time=0.04 sec
1 ARIMA(0,1,1)(0,1,1)[12]      : BIC=911.938, Time=0.07 sec
ARIMA(0,1,1)(0,1,0)[12]        : BIC=946.282, Time=0.01 sec
ARIMA(0,1,1)(1,1,1)[12]        : BIC=916.750, Time=0.09 sec
ARIMA(0,1,1)(0,1,2)[12]        : BIC=916.735, Time=0.25 sec
ARIMA(0,1,1)(1,1,0)[12]        : BIC=925.897, Time=0.04 sec
ARIMA(0,1,1)(1,1,2)[12]        : BIC=inf, Time=1.05 sec
ARIMA(0,1,0)(0,1,1)[12]        : BIC=917.551, Time=0.04 sec
ARIMA(1,1,1)(0,1,1)[12]        : BIC=916.710, Time=0.10 sec
ARIMA(0,1,2)(0,1,1)[12]        : BIC=916.630, Time=0.09 sec
2 ARIMA(1,1,0)(0,1,1)[12]      : BIC=912.562, Time=0.06 sec
ARIMA(1,1,2)(0,1,1)[12]        : BIC=921.407, Time=0.23 sec
ARIMA(0,1,1)(0,1,1)[12] intercept : BIC=916.513, Time=0.07 sec

Best model: ARIMA(0,1,1)(0,1,1)[12]
Total fit time: 2.518 seconds
```

Figure 9.42: `auto_arima` evaluating different SARIMA models based on BIC

As illustrated in Figure 9.42, this produces output from each iteration based on BIC scores during stepwise search, notably the following:

- The winning model remains `ARIMA(0,1,1)(0,1,1)[12]`, which achieved a BIC score of 911.938 and is consistent with the selection based on AIC (Figure 9.41)
- However, model 2, `ARIMA(1,1,0)(0,1,1)[12]`, achieves a BIC score of 912.562 but becomes less competitive under BIC compared to its performance under AIC

This difference illustrates how AIC and BIC evaluate models differently:

- AIC prioritizes goodness-of-fit with a moderate penalty on complexity
- BIC applies a stricter penalty on complexity and often favors simpler models

Despite these differences in evaluation criteria, both converged on `ARIMA(0,1,1)(0,1,1)[12]` as the best model for the dataset. Note that this can also be written as `SARIMA(0,1,1)(0,1,1,12)`.

This highlights that while `auto_arima` automates model selection effectively using statistical criteria such as AIC or BIC, careful judgment is still required to interpret results and validate assumptions through residual diagnostics.

- Evaluate the overall performance of the selected model using the `plot_diagnostics` method. This is the same method you used previously from `statsmodels`:

```
auto_model.plot_diagnostics(figsize=(15,7));
```

This should produce the diagnostic plots for residual analysis of the selected SARIMA model, as shown in *Figure 9.43*:

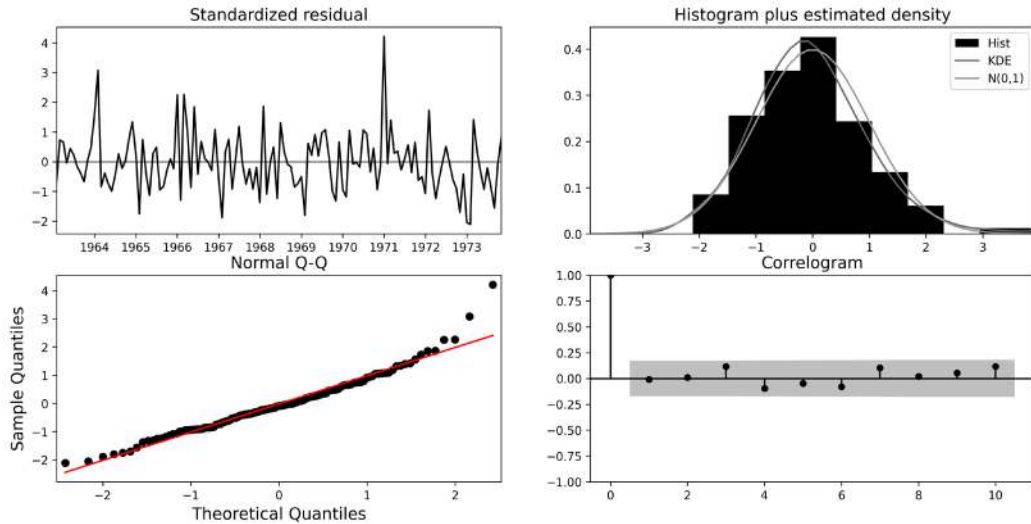


Figure 9.43: Diagnostic plots for residual analysis based on the selected SARIMA model

The diagnostic plots in *Figure 9.43* should look similar to those in *Figure 9.34*.

- To access the model summary, use the `summary` method. This will produce a similar output to that in *Figure 9.40*, showing the SARIMA model summary as selected by `auto_arima`:

```
auto_model.summary()
```

- To forecast future periods, use the `predict` method from your trained model (`auto_model`). Specify the number of periods to forecast forward (`n_periods`) based on the size of your test set (`milk_test`):

```
n = milk_test.shape[0]
index = milk_test.index

# Plot test data and forecast
ax = milk_test.plot(style='--', alpha=0.6, figsize=(12,4))
pd.Series(auto_model.predict(n_periods=n),
          index=index).plot(style='-', ax=ax)
plt.legend(['test', 'forecast']);
```

The produced plot, shown in Figure 9.44, compares the forecasted values against actual test data:

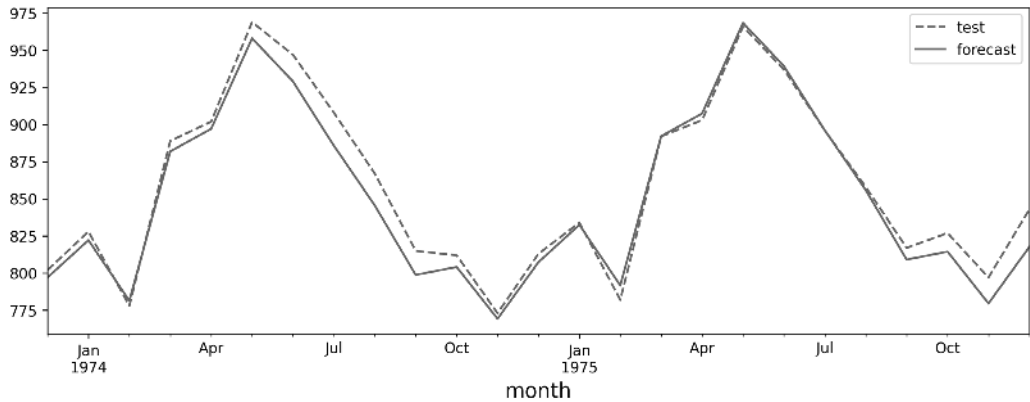


Figure 9.44: Plotting the forecast from `auto_arima` against actual test data

9. To include confidence intervals, set the `return_conf_int` parameter to `True`. This provides both forecasted values and their corresponding lower and upper bounds. The default confidence interval is set to 95% (`alpha=0.05`).

The `fill_between` function in Matplotlib is used to create a shaded area between the lower and upper bounds, providing a clear representation of the uncertainty around the forecasted values.

The following code uses the `predict` method, returns the confidence intervals, and plots the predicted values against the test set:

```
n = milk_test.shape[0]
forecast, conf_interval = auto_model.predict(n_periods=n,
                                             return_conf_int=True,
                                             alpha=0.05)

lower_ci, upper_ci = zip(*conf_interval)

index = milk_test.index
# Plot test data, forecast, and confidence intervals
ax = milk_test.plot(style='--', alpha=0.6, figsize=(12,4))
pd.Series(forecast, index=index).plot(style='-', ax=ax)
plt.fill_between(index, lower_ci, upper_ci, alpha=0.2)
plt.legend(['test', 'forecast']);
```

This should produce a plot (Figure 9.45) with a shaded area representing the likelihood that actual values would lie within this range. Note that narrower confidence intervals indicate higher certainty in predictions.

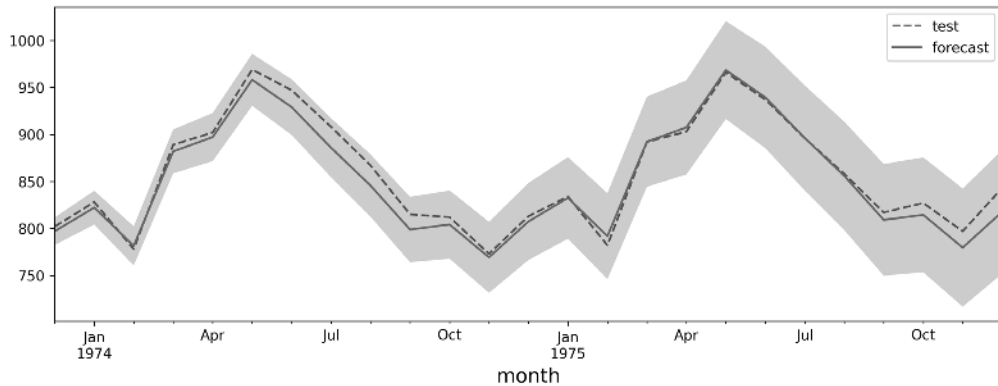


Figure 9.45: Plotting the forecast from `auto_arima` against actual test data with the confidence intervals

Notice that the forecast line lies in the middle of the shaded area. This represents the mean of the upper and lower bounds.

You can verify that the forecast values align with the mean of their confidence intervals using the following:

```
mean_ci = conf_interval.mean(axis=1)
np.array_equal(np.round(forecast, 4), np.round(mean_ci, 4))
>> True
```

How it works...

The `auto_arima` function from the `pmdarima` library simplifies the process of identifying the best-fitting SARIMA model by automating parameter selection. With `stepwise=True` (default), the `auto_arima` optimization process involves several key steps working together:

- **Stationary testing:**
 - It uses unit root tests (ADF, KPSS, or PP) to determine the required non-seasonal differencing order (d)
 - For seasonal data, it applies the OCSB test (or alternatives such as CH) to determine the seasonal differencing order (D)
 - Both orders can be manually specified or automatically determined
- **Model search process:**
 - Non-seasonal components start at $p=2$ and $q=2$ and can reach a maximum of 5
 - Seasonal components start at $P=1$ and $Q=1$ and can reach a maximum of 2
 - It uses statistical measures (information criteria) to evaluate and compare models:

- **AIC:** Evaluates model goodness of fit while penalizing complexity to avoid overfitting
 - **AICc:** A version of AIC adjusted for small sample sizes to reduce bias in model selection
 - **BIC:** Puts a stronger penalty on model complexity (additional parameters), favoring simpler models as sample size increases
 - **HQIC (Hannan-Quinn Information Criterion):** A balanced approach between AIC and BIC, with a penalty term that grows more slowly than BIC
- It has a maximum of 50 iterations (default, `maxiter=50`) using the L-BFGS-B optimization method (default, `method='lbfgs'`)
 - **Model estimation:**
 - Fits models using MLE
 - Returns the model that minimizes the selected information criterion

There's more...

In addition to using `pmdarima` directly, you can leverage the Darts library's `AutoARIMA` class for time series forecasting. In the Forecasting univariate time series data with exponential smoothing recipe, you were introduced to the Darts library in the *There's more...* section.

The `AutoARIMA` class, which wraps around the `statsforecasts` library's `AutoARIMA`, offers a much faster implementation than `pmdarima`'s `AutoARIMA` and may offer better performance for large datasets. The following code demonstrates how you can use `StatsForecast`'s `AutoARIMA`:

```
from darts.models import AutoARIMA
# Initialize StatsForecast AutoARIMA with seasonal periodicity

ts = TimeSeries.from_dataframe(milk_train.reset_index(),
                              time_col='month',
                              value_cols='production',
                              freq='MS')

model = AutoARIMA(season_length=12)

# Fit the model on training data
model.fit(ts)

# Generate forecasts
pred = model.predict(len(milk_test))

# Plot training data and forecasts
ts.plot(label='Training')
pred.plot(label='StatsforecastsAutoArima');
```


This code produces a plot (Figure 9.46) showing the forecast:

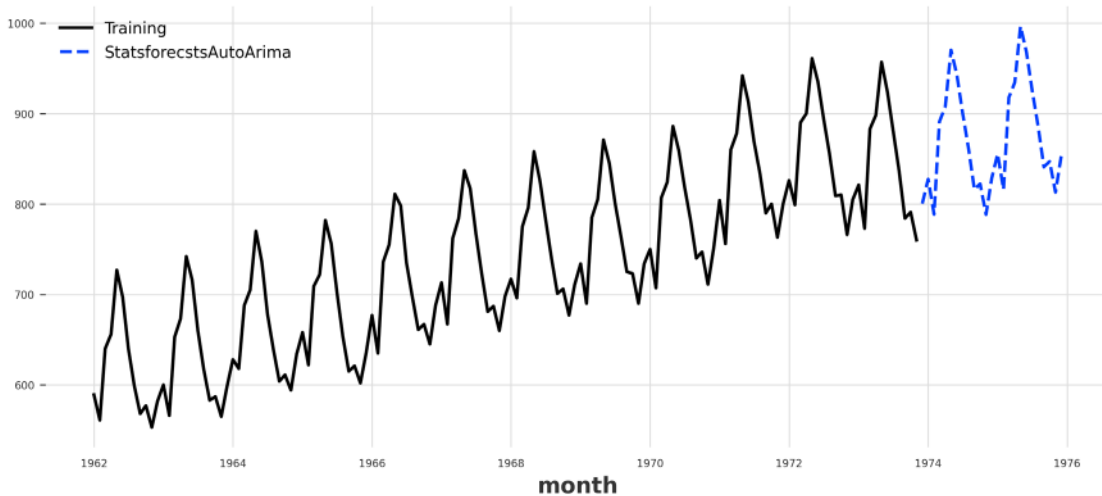


Figure 9.46: Plotting the forecast using `StatsForecastAutoARIMA` from the `Darts` library

See also

- To learn more about the `auto_arma` implementation, visit the official documentation at https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arma.auto_arma.html.
- To learn more about Dart's `StatsForecastAutoARIMA`, visit the official documentation at https://unit8co.github.io/darts/generated_api/darts.models.forecasting.sf_auto_arma.html

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



10

Additional Statistical Modeling Techniques for Time Series

In *Chapter 9*, you were introduced to popular forecasting techniques such as exponential smoothing, non-seasonal ARIMA, and seasonal ARIMA. These classical statistical approaches are widely used due to their speed, simplicity, and interpretability. Building on that foundation, this chapter introduces advanced statistical methods and powerful libraries that automate time series forecasting and model optimization.

You will learn about **Prophet** (developed by Facebook, now Meta), a robust tool for handling seasonality and holiday effects in time series forecasting. We'll also look into statsmodels' **vector autoregressive (VAR)** class for multivariate time series modeling and the **arch** library, which supports **GARCH** models—essential for forecasting volatility in financial data.

The objective of this chapter is to expand your time series modeling toolkit with these advanced techniques. You will learn how to apply **automated forecasting** with Prophet, explore **multivariate time series modeling** using VAR models, and gain a solid understanding of how to model and forecast **volatility** in financial time series—a crucial aspect for risk management and decision-making in finance.

Additionally, you will explore the StatsForecast library, which includes the implementation of AutoARIMA, AutoETS, and AutoTheta. You will learn how to perform cross-validation and forecast with exogenous variables.

In this chapter, we will cover the following recipes:

- Forecasting univariate time series data using Prophet
- Forecasting univariate time series data with Theta
- Forecasting multivariate time series data using VAR
- Evaluating VAR models
- Forecasting volatility in financial time series data with GARCH
- Automated univariate forecasting using the StatsForecast library

Technical requirements

You can download the Jupyter notebooks to follow along and the necessary datasets for this chapter from this book's GitHub repository:

- **Jupyter notebooks:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch10/Chapter_10.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch10>

There are common libraries that you will be using throughout the recipes in this chapter. For convenience, you can import them in advance by using the following code:

```
# Import commonly used libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

# Suppress warnings for cleaner output
import warnings
warnings.filterwarnings('ignore')

# Set default figure size for plots
plt.rc("figure", figsize=(16, 5))
```

To keep the code consistent across the upcoming recipes, this chapter uses a small utilities module, `utils.py`. You can import the helper functions as follows:

```
from utils import split_data, check_stationarity
```

These functions provide two core capabilities:

- `split_data` splits a time series into training and test sets based on a specified number of periods for testing, while preserving temporal order so that evaluation always reflects future data.
- `check_stationarity` applies the Augmented Dickey–Fuller test and returns a simple ‘Stationary’ or ‘Non-Stationary’ label based on the p-value, giving a quick indication of whether further transformation may be needed before modeling.

Forecasting univariate time series data using Prophet

The Prophet library is a popular open source project that was initially developed at Facebook (now Meta), based on a 2017 paper that proposed an algorithm for time series forecasting titled *Forecasting at Scale*. The project gained popularity due to its simplicity, ability to create performant forecasting models, and handle complex seasonality and holiday effects while being robust to missing data and outliers.

Prophet automates many aspects of designing a forecasting model, such as built-in cross-validation and hyperparameter tuning, while providing rich built-in visualizations. Additional features include building growth models (e.g., **saturated forecasts**), working with uncertainty in trend and seasonality, and detecting **change points**.

In this recipe, you will use the *Milk Production* dataset for benchmarking performance. This is the same dataset introduced in *Chapter 9*. Using the same dataset helps in understanding and comparing different methods, particularly how Prophet's approach differs from traditional methods such as SARIMA and exponential smoothing, which we covered earlier.

At its core, Prophet is an **additive regression model** that incorporates a piecewise linear or logistic growth curve to model trends. It can effectively handle **non-linear trends**, especially when strong seasonal effects are present. Prophet can handle multiple seasonal patterns, such as yearly, weekly, and daily patterns, modeled using a Fourier series. The model decomposes time series data into three main components: *trend*, *seasonality*, and *holidays*. The decomposition is represented mathematically as follows:

$$y(t) = g(t) + s(t) + h(t) + \epsilon(t)$$

Let's break this down:

- $g(t)$ is the trend function
- $s(t)$ represents the periodic seasonality function
- $h(t)$ accounts for the effects of holidays
- $\epsilon(t)$ is the residual error term not explained by the model

Prophet uses **Bayesian inference** to automate the tuning and optimization of the model components. Under the hood, it relies on **Stan**, a state-of-the-art platform for Bayesian modeling, accessed through a modern Python interface such as **cmdstanpy** (which replaces the earlier **PyStan**). Recent updates have improved compatibility with Apple M-series chips and enhanced model customization options for handling holidays and scaling output.

Getting ready

You can find the Jupyter notebooks and necessary datasets from this book's GitHub repository. Please refer to the *Technical requirements* section of this chapter for more information. We use Prophet version 1.1.7 for this recipe.



Creating a new Python environment when installing new libraries such as **Prophet** is always a good idea. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup* recipe from *Chapter 0* on GitHub. The chapter covers two methods: using conda and venv.

To install Prophet using pip, you can run the following:

```
pip install prophet
```

How to do it...

Prophet requires the input data to be in a pandas DataFrame with two specific columns: a datetime column named `ds` and a target variable column named `y` – this is the variable you wish to forecast. Prophet does not work with a `DatetimeIndex` directly. Hence, it's crucial to have these columns explicitly named. If your data contains more than two columns, Prophet will only recognize `ds` and `y` and ignore the rest by default. However, if you want to include additional predictors (regressors), you can use the `add_regressor` method to incorporate external factors that might influence your forecasts. If these required columns (`ds` and `y`) are missing or incorrectly named, Prophet will throw an error.

To ensure that your data is formatted properly, follow these steps:

1. Start by reading the *Milk Production* dataset (`milk_production.csv`) and renaming its columns to match Prophet's requirements:

```
from prophet import Prophet
milk_file = Path('../datasets/Ch10/milk_production.csv')
milk = pd.read_csv(milk_file,
                  header=0,
                  parse_dates=['ds'],
                  names=['ds', 'y'])

milk.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 168 entries, 0 to 167
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    ds      168 non-null      datetime64[ns]
1    y        168 non-null      int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 2.8 KB
```

2. Split the data into training and test sets using the provided `test_split()` function. Hold out the last 12 months as your test set:

```
test_split = 12
train, test = split_data(milk, test_split)

print(f'Train: {train.shape}')
print(f'Test: {test.shape}')
>>
Train: (156, 2)
Test: (12, 2)
```

3. Create an instance of the Prophet class and fit it on the training set using its `fit` method. The milk production time series exhibits both a trend and seasonality. The seasonal magnitudes and variations over time are relatively steady (not increasing with the trend), suggesting an additive seasonal model. The default `seasonality_mode` type in Prophet is 'additive', which we will keep as is:

```
from prophet import Prophet
model = Prophet(seasonality_mode='additive').fit(train)
```

Other parameters are left with their default values, such as `yearly_seasonality='auto'`, `weekly_seasonality='auto'`, and `daily_seasonality='auto'`.

By default, Prophet automatically detects yearly seasonality if present in your data. You could manually set this by specifying `yearly_seasonality=True`, but 'auto' works well here.

4. Check which seasonal components are included in the model:

```
model.seasonalities
>>
OrderedDict([('yearly',
               {'period': 365.25,
                'fourier_order': 10,
                'prior_scale': 10.0,
                'mode': 'additive',
                'condition_name': None}]])
```

Prophet identified a yearly seasonal component with a `fourier_order` of 10, which controls the flexibility of the seasonal pattern. A higher value allows more flexibility in capturing complex seasonal patterns but may risk overfitting.

5. Before you can use the model to make predictions, you will need to extend the training data forward using the `make_future_dataframe` method. You will need to specify how far into the future you want predictions. Set the number of periods to be the length of the test dataset and specify the frequency as month start ('MS'):

```
future = model.make_future_dataframe(len(test), freq='MS')
```

6. This extends the training data by 12 months (adding rows corresponding to each month in the test set), creating a total of 168 rows: 156 from the training and 12 for forecasting. The frequency is set to **month start** with `freq='MS'`. The future DataFrame contains only one column (`ds`) of the `datetime64[ns]` type, used to populate the predicted values:

```
print(future.tail(12))
>>
      ds
156 1975-01-01
157 1975-02-01
```

```
158 1975-03-01
159 1975-04-01
160 1975-05-01
161 1975-06-01
162 1975-07-01
163 1975-08-01
164 1975-09-01
165 1975-10-01
166 1975-11-01
167 1975-12-01
```

The `predict` method generates predictions and returns a `DataFrame` with the same length as the `future` `DataFrame`, but with additional columns:

```
forecast = model.predict(future)
forecast.columns.tolist()
>>
['ds',
 'trend',
 'yhat_lower',
 'yhat_upper',
 'trend_lower',
 'trend_upper',
 'additive_terms',
 'additive_terms_lower',
 'additive_terms_upper',
 'yearly',
 'yearly_lower',
 'yearly_upper',
 'multiplicative_terms',
 'multiplicative_terms_lower',
 'multiplicative_terms_upper',
 'yhat']
```

The forecast `DataFrame` includes additional columns that provide insight into the model's performance, such as uncertainty intervals and seasonal components. The important columns in the forecast `DataFrame` are `ds` (dates corresponding to each prediction), `yhat` (the predicted value), and `yhat_lower` and `yhat_upper` (the uncertainty intervals for the predictions).

The `model` object provides two built-in plotting methods: `.plot()` and `.plot_components()`. Start by using `.plot()` to visualize both the historical data (training set) and the future forecast generated by Prophet:

```
model.plot(forecast,
           ylabel='Milk Production in Pounds',
           xlabel='Month',
           include_legend=True);
```

This should produce a plot of the Prophet forecast: the dots show actual observations (training data points), the solid line represents the model's estimated forecast for the historical data (over the dots), and future predictions are shown with the line extending beyond the training points. The shaded area around the forecast is derived from the `yhat_lower` and `yhat_upper` columns in the forecast DataFrame:

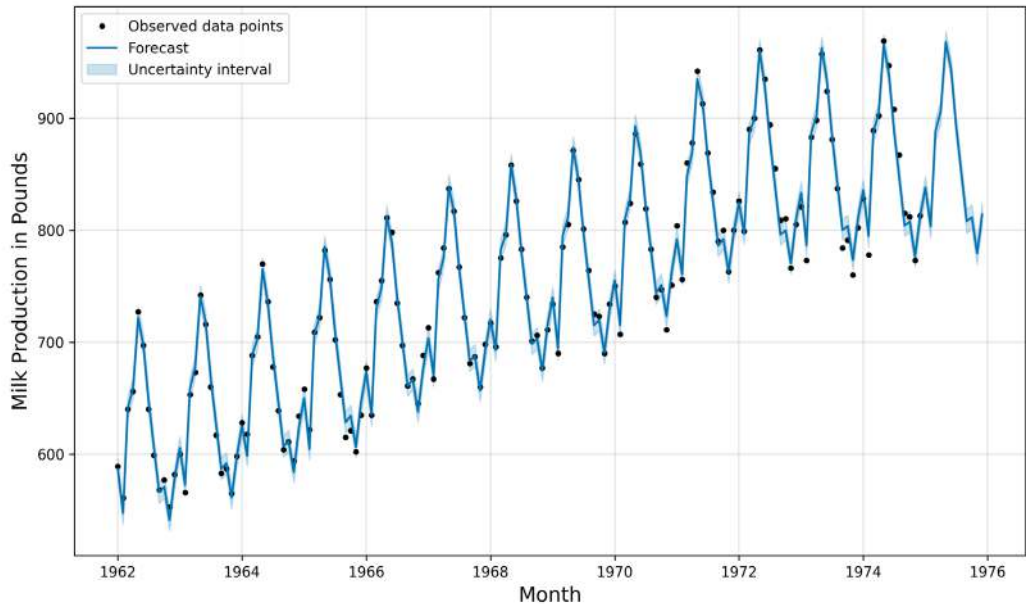


Figure 10.1: Plotting the forecast (historical and future) using Prophet

7. If you only want to visualize the forecast just for the periods beyond the training set (i.e., future predictions), you can use the following code:

```
model.plot(model.predict(test),
           ylabel='Milk Production in Pounds',
           xlabel='Month',
           include_legend=True);
```

Here, you are generating predictions for the length of the test DataFrame. The `predict` method will only capture the `ds` column (datetime values) from the test DataFrame. This should produce a plot that's similar to the one shown in Figure 10.1, but instead of showing the complete historical fit and forecast, it will only show the predictions for future periods (beyond the training points):

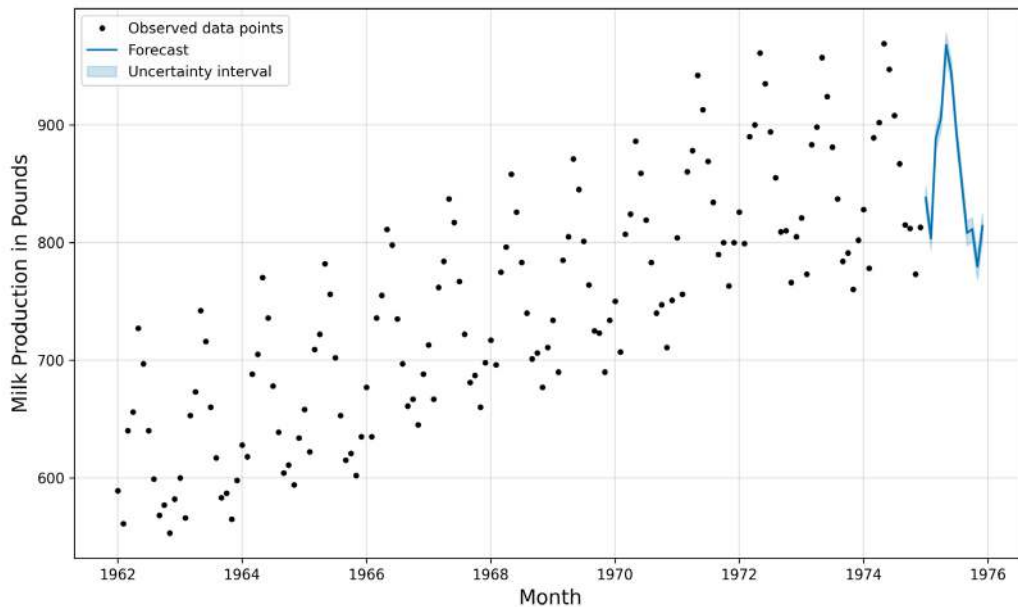


Figure 10.2: Plotting the forecast using Prophet

The shaded area in *Figures 10.1* and *10.2* represents the uncertainty intervals (defaults to 80% confidence interval) derived from the `yhat_lower` and `yhat_upper` columns in the forecast DataFrame. These intervals help understand the range of likely values for each prediction (model's confidence), with wider intervals indicating greater uncertainty in the forecast.

8. You can change the confidence interval when you initialize the Prophet model by updating the `interval_width` parameter, as in this example:

```
# setting interval width to 95% instead of the default 80%
model_95 = Prophet(seasonality_mode='additive',
                    interval_width=0.95).fit(train)
forecast_95 = model_95.predict(future)
```

Changing the `interval_width` parameter will only affect the `yhat_lower` and `yhat_upper` columns in the forecast DataFrame.

The `plot_components()` method is used to visualize individual components of the model by decomposing the forecast into its constituent parts, such as trend, seasonality, and holidays. The trend line is based on the `trend`, `trend_lower`, and `trend_upper` columns in the forecast DataFrame. This provides deeper insight into how each component contributes to the overall forecast:

```
model.plot_components(forecast);
```

The number of subplots will depend on the number of components identified in the forecast. For example, if holidays were included, then it would show a subplot for the holiday component. In our example, there are two subplots: trend and yearly seasonality:

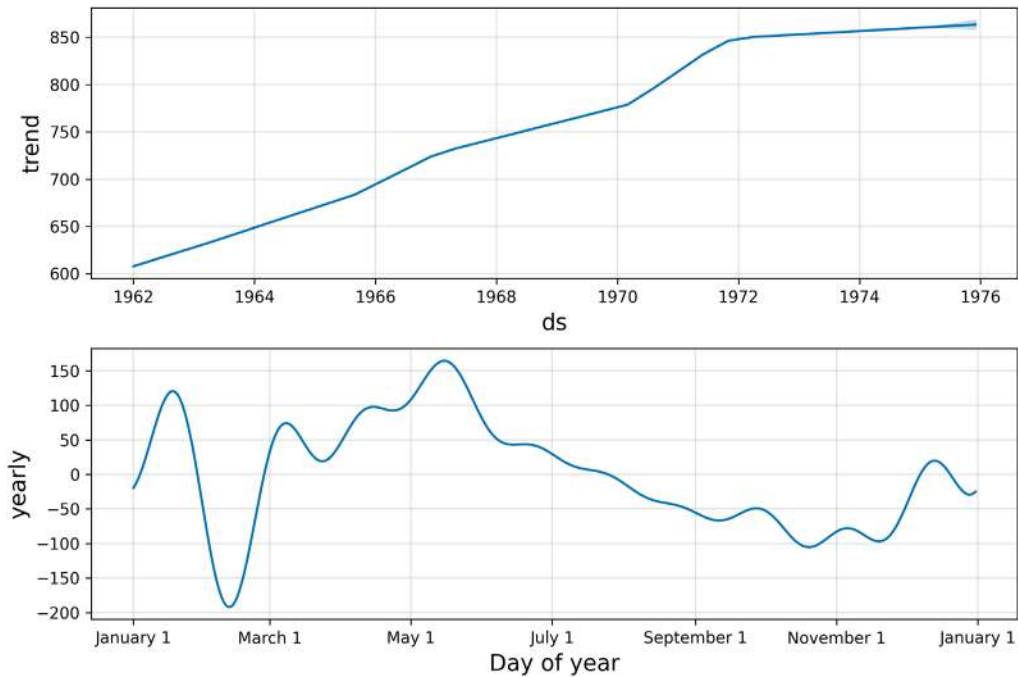


Figure 10.3: Plotting the components showing trend and seasonality (annual)

Figure 10.3 breaks down the trend and seasonality of the training data. If you look at Figure 10.3, you will see a positive upward trend that shows a clear growth rate change around 1972, transitioning from steep growth to a more steady, plateau-like pattern. This kind of trend change is automatically detected by Prophet’s changepoint detection feature, which identifies points in time where significant shifts in the trend occur. Additionally, the yearly seasonal pattern shows clear peaks in milk production during summer months (particularly around June), followed by significant drops (significant in mid-February).

The shaded area in the trend plot represents the uncertainty interval for estimating the trend based on the `trend_lower` and `trend_upper` columns of the forecast DataFrame.

Since our `seasonality_mode` type was ‘additive’, this means that the two components (trend and yearly) can be combined by adding them to give us our forecast `yhat`. The following code illustrates this concept:

```
forecast_tmp = forecast.set_index('ds')
(forecast_tmp['trend']+forecast_tmp['yearly']).plot();
```

This should produce a similar plot to Figure 10.1.

- Finally, compare Prophet's predictions with out-of-sample data (the test set) to evaluate how well the model performs:

```
n = len(test)
ax = test.plot(x='ds', y='y',
               label='Actual',
               style='-.',
               figsize=(12,4), xlabel='Date')

forecast.tail(n).plot(x='ds', y='yhat',
                     label='Predicted',
                     ax=ax,
                     title='Milk Production Actual vs Forecast',
                     xlabel='Date',
                     ylabel='Milk Production (Pounds)');
```

This produces a plot comparing actual milk production values (`test['y']`) with predicted values (`predicted['yhat']`).

Compare the following plot with *Figure 9.44* from *Chapter 9* to see how Prophet compares to the SARIMA model that you obtained using `auto_arima`:

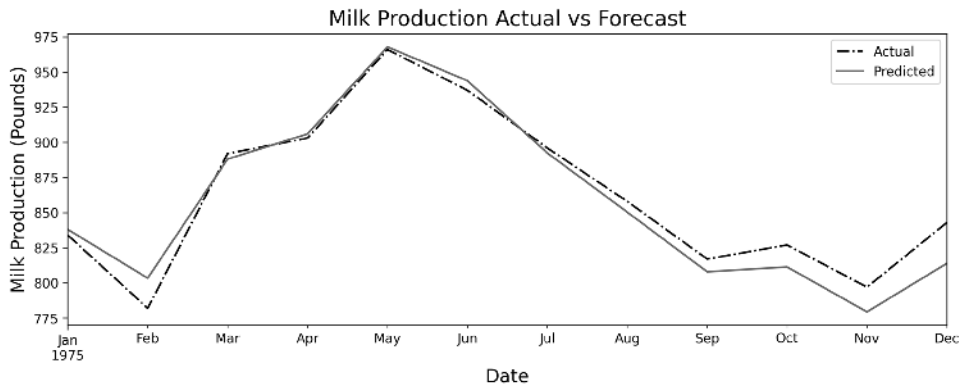


Figure 10.4: Comparing Prophet's forecast against test data

For highly seasonal data such as milk production, Prophet performs exceptionally well by accurately capturing both trend and seasonal patterns. Generally, Prophet shines when working with time series data that exhibits strong seasonality or complex seasonal effects.

How it works...

Prophet streamlines many aspects of building and optimizing time series models, but requires a few key instructions at the start to allow Prophet to properly tune the model. For example, when initializing the model, you need to decide whether the seasonal effect should be additive or multiplicative. Additionally, when extending the forecast horizon using the `make_future_dataframe` method, you specify parameters such as the number of periods to forecast and the frequency of the data (e.g., `freq='MS'` for monthly data).

When you instantiate a model with `model = Prophet()`, Prophet uses the default values for parameters such as `yearly_seasonality='auto'`, `weekly_seasonality='auto'`, and `daily_seasonality='auto'`. This allows Prophet to automatically determine which seasonal components to include based on the data. For instance, Prophet will include yearly seasonality if the data spans more than two years, weekly seasonality if there are at least two weekly cycles in the data and the spacing between the dates is less than seven days, and daily seasonality for sub-daily (e.g., hourly or minute-level data). You can add custom seasonality patterns such as monthly, quarterly, or hourly, using the `add_seasonality` method.

In the case of the *Milk Production* dataset, only yearly seasonality was detected. Additional parameters that you can modify to control the model outcome in Prophet include trend configuration, seasonality configuration, holiday configuration, and uncertainty configuration.

Trend configuration

- `growth`: The type of trend – linear, logistic, or flat. The default is linear.
- `changepoints`: To manually specify the dates where the trend might change based on domain knowledge. The default is `None`, allowing Prophet to automatically select changepoints based on the data.
- `n_changepoints`: The number of potential changepoints that Prophet will consider. The default is 25.
- `changepoint_range`: Specifies the proportion of the historical data in which changepoints are considered. The default is 0.8, meaning that only the first 80% of the time series data is considered to avoid overfitting to recent fluctuations.
- `changepoint_prior_scale`: A regularization parameter that controls the flexibility of the trend at changepoints. Larger values (up to 0.5) allow for more sudden changes at changepoints, while smaller values (down to 0.001) produce a smoother trend. The default is 0.05.

Seasonality configuration

- `seasonality_mode`: Determines whether the seasonal component interacts with the trend in an additive or multiplicative manner. The default is additive.
- `yearly_seasonality`: Controls annual seasonal patterns. The default is `auto`, and can be set to `True` for forced inclusion, or an integer to specify the number of Fourier terms (the default order is 10). Higher orders capture more complex patterns but may overfit, while lower orders produce smoother seasonality but may underfit.
- `weekly_seasonality`: Controls weekly seasonal patterns. Behaves similarly to `yearly_seasonality` with the same parameter options (`auto`, `True`, or integer for Fourier terms). The default Fourier order is 3.
- `daily_seasonality`: Controls daily seasonal patterns. Follows the same parameter structure as `yearly_seasonality` and `weekly_seasonality`, particularly useful for high-frequency data. The default Fourier order is 3.
- `seasonality_prior_scale`: A regularization parameter controlling the flexibility of the seasonality. A large value (up to 10) allows the seasonality to fit large fluctuations (more variation in seasonality due to minimal regularization) while a small value (down to 0.01) enforces more regularization. The default is 10, meaning minimal to no regularization is applied.

To understand the effect of the Fourier order and how it changes the seasonality, you can try the following example by creating a model with a lower Fourier order (4 instead of the default 10) to see how it affects the seasonal pattern:

```
from prophet.plot import plot_yearly

# Example of adjusting Fourier order for yearly seasonality
model_y4 = Prophet(seasonality_mode='additive',
                    interval_width=0.80,
                    yearly_seasonality=4).fit(train)
forecast_y4 = model_y4.predict(future)
plot_yearly(model_y4);
```

This should produce a plot as shown in *Figure 10.5*:

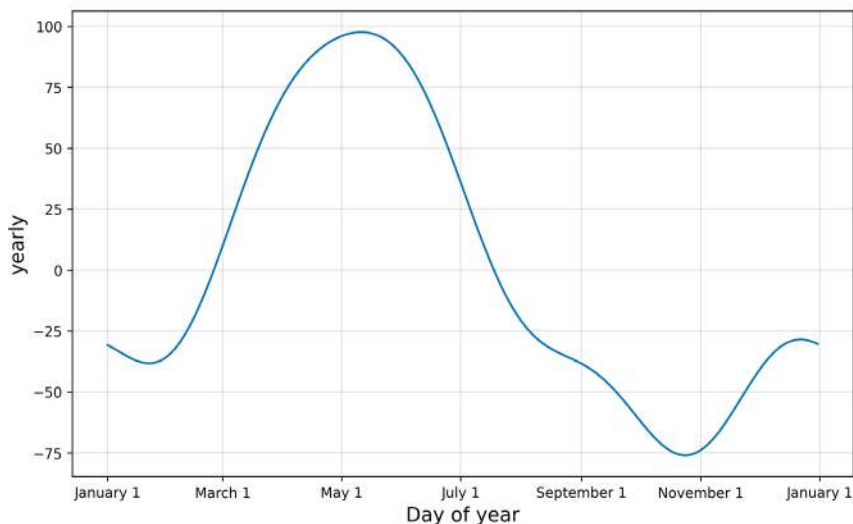


Figure 10.5: Yearly seasonal pattern after adjusting Fourier order with `yearly_seasonality=4`

Figure 10.5 shows a smoother seasonal pattern compared to the default Fourier order of 10 (Figure 10.3). Notice how the curve has fewer fluctuations and captures the main seasonal pattern. A Fourier order of 4 uses 8 parameters (4 sine and 4 cosine terms) to model the seasonal pattern, while the default order of 10 uses 20 parameters (10 sine and 10 cosine terms). Higher Fourier orders allow the model to fit more complex seasonal patterns but may overfit, while a lower Fourier order produces a smoother seasonal curve, reduces overfitting, but may miss finer detail in the data.

Holiday configuration

- **holidays:** Takes a DataFrame containing holiday dates (`ds` column) and holiday names (`holiday` column) to be included in the forecast.
- **holidays_mode:** Similar to `seasonality_mode`, this can be additive or multiplicative. It defaults to the value specified in `seasonality_mode`.

- **holidays_prior_scale**: Controls the flexibility of the holiday effects. Larger values (up to 10) allow more flexibility, and smaller values (down to 0.01) constrain the holiday effect. The default is 10.

Uncertainty configuration

- **interval_width**: Controls the width of the uncertainty intervals. It defaults to 0.8 (80% confidence interval).



All `prior_scale` parameters (**seasonality**, **holidays**, and **changepoints**) operate on a log scale, meaning small changes can have significant effects on the model's flexibility.

Prophet provides **uncertainty intervals** for predictions, which are influenced by three factors:

- **Observation noise**: Refers to the random variations in the observed data that cannot be explained by the model.
- **Parameter uncertainty**: Refers to the uncertainty in estimating the model parameters. For example, adjusting the `mcmc_samples` parameter greater than zero enables **Markov Chain Monte Carlo (MCMC)** sampling to estimate uncertainty in seasonal components. When `mcmc_samples=0` (default), Prophet uses **Maximum a Posteriori (MAP)** estimation, which is faster but may underestimate uncertainty.
- **Future trend uncertainty**: Refers to uncertainty about future changes in trends based on historical data. For example, increasing the `changepoint_prior_scale` parameter (the default value is 0.05) can increase the forecast uncertainty.

Additionally, the width of the uncertainty intervals can be adjusted with the `interval_width` parameter (the default is 80% confidence interval).

By default, the `uncertainty_samples` parameter is set to 1000, which means Prophet will run 1,000 simulations using a **Hamiltonian Monte Carlo (HMC)** algorithm to estimate uncertainty. You can adjust this to control the number of simulations or even disable uncertainty estimates entirely by setting `uncertainty_samples=0` or `uncertainty_samples=False`. If uncertainty samples are disabled, Prophet will omit uncertainty intervals such as `yhat_lower` and `yhat_upper` from the forecast results:

```
model_no_uncertainty = Prophet(uncertainty_samples=False).fit(train)
forecast_no_uncertainty = model_no_uncertainty.predict(future)
forecast_no_uncertainty.columns.tolist()
>>
['ds', 'trend', 'additive_terms', 'yearly', 'multiplicative_terms', 'yhat']
```

Notice that `yhat_lower` and `yhat_upper` are not present in the forecast columns when `uncertainty_samples=False`.

Prophet's strength lies in its ability to automatically detect **changepoints**, which are points in time where the trend shifts significantly. After Prophet detects changepoints, you can inspect them using the model's `changepoints` attribute. For example, the following code displays the first five changepoints:

```

model.changepoints.shape
>>
(25,)

model.changepoints.head()
>>
5    1962-06-01
10   1962-11-01
14   1963-03-01
19   1963-08-01
24   1964-01-01
Name: ds, dtype: datetime64[ns]

```

These changepoints can also be visualized on a plot. The following code overlays the changepoints on the original time series data:

```

ax = milk.set_index('ds').plot(figsize=(12,5))
milk.set_index('ds').loc[model.changepoints].plot(
    style='x',
    ax=ax,
    markersize=10
)
plt.xlabel('Date')
plt.ylabel('Milk Production')
plt.title('Time Series with Detected Changepoints')
plt.legend(['Original Data', 'Changepoints']);

```

This should generate a plot with the original time series and the 25 detected changepoints, showing moments where Prophet identified shifts in the trend.

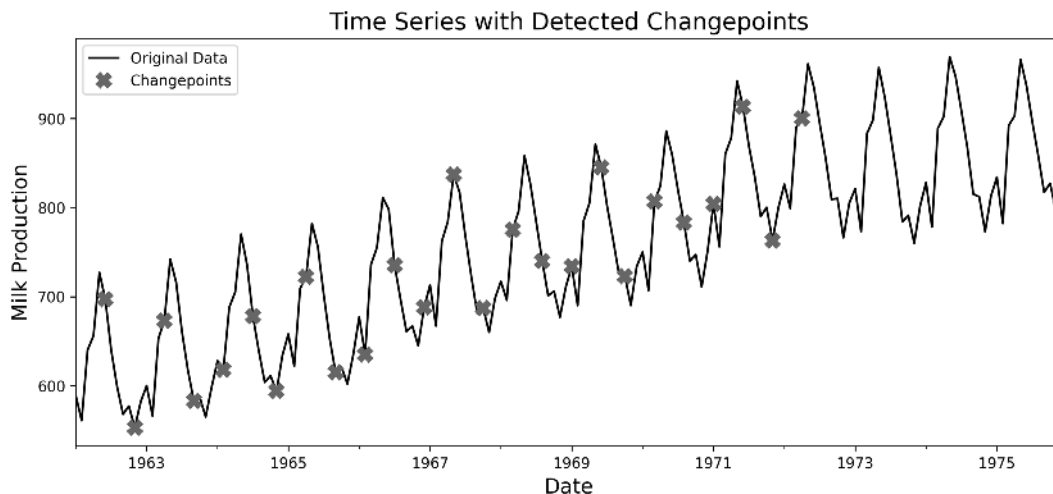


Figure 10.6: The 25 potential changepoints detected by Prophet

The changepoints in *Figure 10.6* were estimated based on the first 80% of the training data.

There's more...

In this section, you will explore additional Prophet features and capabilities such as changepoint detection, cross-validation, and incorporating holidays and extra regressors.

Detecting changepoints

1. To plot the significant changepoints that capture the impactful changes in trend, you can use the `add_changepoints_to_plot` function, as shown in the following code:

```
from prophet.plot import add_changepoints_to_plot

fig = model.plot(forecast, ylabel='Milk Production in Pounds')
add_changepoints_to_plot(fig.gca(), model, forecast);
```

This should produce a plot similar to *Figure 10.6*, but with the additional changepoint lines and a trend line. There are nine (the vertical lines in *Figure 10.7*) out of the 25 potential changepoints that are considered significant by Prophet. The trend line should look the same as the trend component shown in *Figure 10.3*:

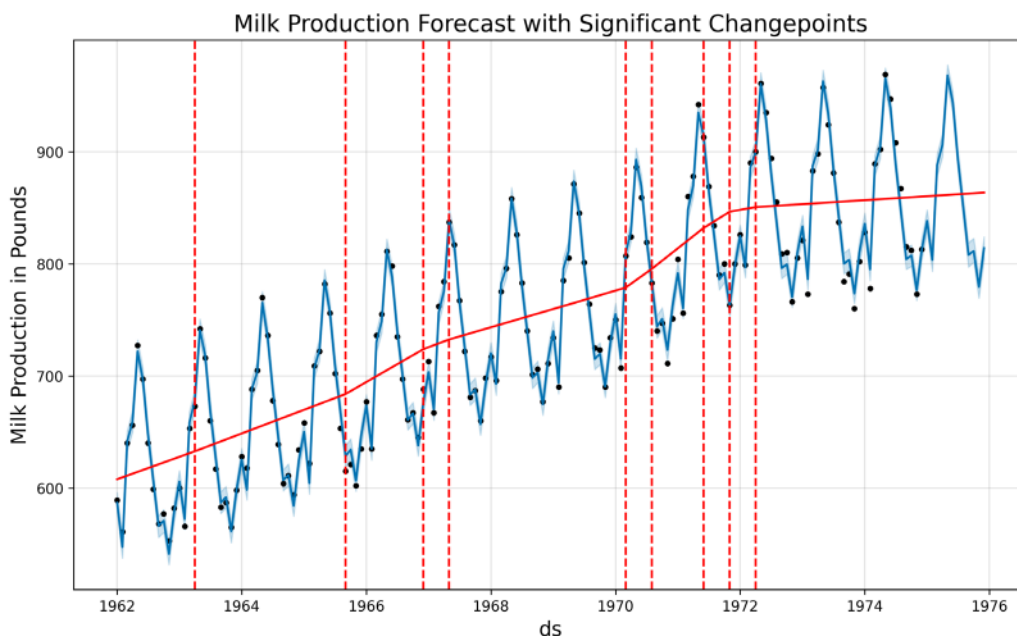


Figure 10.7: Showing the nine significant changepoints and the trend line

Notice how the trend line changes at the identified changepoints. This is how Prophet detects changes in the trend. The line is not an exact straight line since **piecewise regression** was applied to build the trend model. When thinking about *piecewise linear models*, you can think of multiple linear regression lines between the significant changepoints (segments) that are then connected. This gives the model the flexibility to capture non-linear changes in trends and make future predictions.

Prophet's default behavior does not include all potential changepoints in the plot; it only shows those above a significant threshold controlled by the `threshold` parameter. The default is `threshold=0.01`. You can adjust the threshold; for example, if you increase the threshold to `threshold=0.1`, it will only show 5 significant changepoints, while `threshold=0.001` will show 11 significant changepoints. The threshold parameter is based on absolute values, which is the magnitude of the trend change regardless of whether it is positive (trend increase) or negative (trend decrease).

The change in the trend's growth rate at each changepoint is referred to as **delta**, where positive values indicate acceleration (steeper slope) in the trend and negative values indicate deceleration (shallower slope). You can observe these values (normalized) using the following:

```
print(model.params['delta'])
>>
[[ 4.45860e-09 -6.20724e-09  1.49544e-02  4.04784e-05  5.86443e-09
  1.89663e-10 -9.72030e-09  3.44925e-08  1.45392e-01  1.49468e-10
 -2.43211e-08 -1.48081e-01 -6.21233e-02 -1.27773e-08 -1.09484e-09
  9.59719e-11  6.09539e-04  2.68757e-03  1.21536e-03  3.08689e-01
  5.50464e-02 -2.76747e-08 -1.23803e-01 -3.30631e-01 -8.33208e-02]]
```

2. You can visualize the magnitude of trend changes using the following:

```
# Extract delta values
deltas = model.params['delta'].ravel()
indices = list(range(len(deltas)))

# Plot bar chart
plt.bar(indices, deltas)
plt.axhline(0, color='black', linewidth=1, linestyle='--')
plt.title('Magnitude of Trend Changes at Changepoints')
plt.xlabel('Potential Changepoints')
plt.ylabel('Delta (Rate Change)')
plt.tight_layout();
```

This creates a plot showing nine prominent bars, while the other potential changepoints have delta values too small to be visually significant:

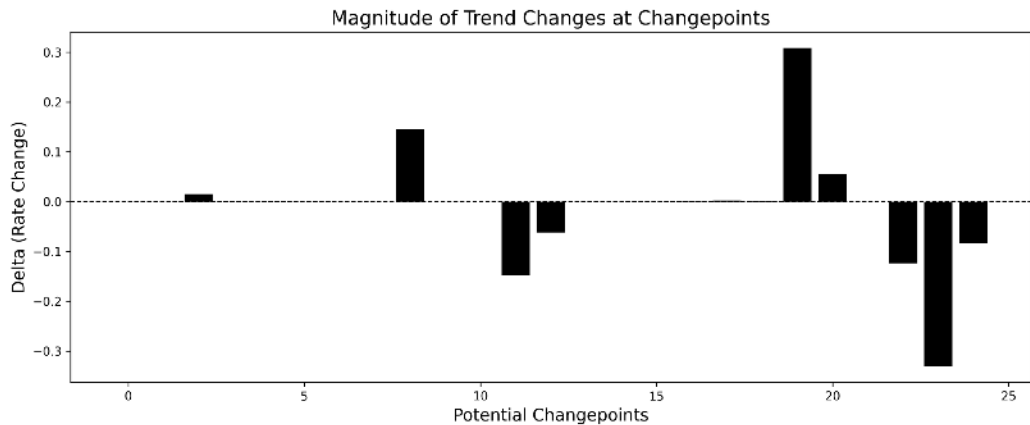


Figure 10.8: Magnitude of trend changes at changepoints

The bar plot in *Figure 10.8* shows the magnitude of trend changes at each potential changepoint, where larger bars indicate more significant changes in the trend's growth rate. The dashed line at zero helps distinguish between these increases and decreases in the trend

Performing cross-validation

Prophet includes built-in **cross-validation** capabilities that help evaluate the model's forecasting performance across different time periods. Cross-validation is crucial for understanding how the model might perform on future, unseen data. You want to ensure that the model generalizes well to unseen data and is not overfitting or underfitting. Additionally, cross-validation can help you determine how far into the future the forecasts remain reliable.

Prophet provides the `cross_validation` and `performance_metrics` functions, which allow you to split the data into training and testing sets across multiple horizons. This method helps evaluate the model's accuracy by making predictions at different points in time and comparing them to the actual value.

Here is how to implement cross-validation in Prophet:

```
from prophet.diagnostics import cross_validation, performance_metrics

df_cv = cross_validation(model, initial='1095 days',
                        period='180 days', horizon='365 days')

df_cv.head()
>>
```

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	1965-02-01	625.892938	621.523363	630.663773	622	1965-01-17
1	1965-03-01	710.385078	696.710482	725.050533	709	1965-01-17
2	1965-04-01	727.222480	702.377261	752.380902	722	1965-01-17
3	1965-05-01	795.152898	756.630939	832.962859	782	1965-01-17
4	1965-06-01	763.399318	710.172130	813.119081	756	1965-01-17

In the preceding code, we specified the following parameters:

- **initial:** The size of the initial training period. In our code, we specified 1,095 days, which is roughly 3 years, or 36 months, for our monthly Milk Production data. The initial period should be long enough to ensure that Prophet has enough data to capture both trend and yearly seasonality.
- **period:** The spacing between cutoff points for making forecasts. In this example, we specified 180 days (roughly 6 months). This means after the initial training, Prophet will step forward in 6-month increments to create new training/test splits.
- **horizon:** The forecast horizon (how far into the future you are predicting). In this example, we specified 365 days (12 months). This means that for each cutoff date, Prophet will forecast for the next 12 months.

Notice that Prophet uses days for these parameters even if your data is monthly.

After running the cross-validation, you can evaluate the model's performance using the `performance_metrics` function, which takes the cross-validation DataFrame as input:

```
df_perf = performance_metrics(df_cv)
print(df_perf.iloc[:, :-1].head())
>>
```

	horizon	mse	rmse	mae	mape	mdape	smape
0	37 days	242.972743	15.587583	12.148291	0.015787	0.013839	0.015745
1	41 days	244.548865	15.638058	12.205994	0.015949	0.013839	0.015892
2	42 days	251.287816	15.852060	12.427641	0.016183	0.013839	0.016144
3	43 days	243.250786	15.596499	11.882913	0.015513	0.011751	0.015468
4	46 days	246.641600	15.704827	12.063200	0.015748	0.013839	0.015710

The function computes six key evaluation metrics: **Mean Squared Error (MSE)**, **Root Mean Squared Error (RMSE)**, **Mean Absolute Error (MAE)**, **Mean Absolute Percent Error (MAPE)**, **Median Absolute Percent Error (MdAPE)**, and **Symmetric Mean Absolute Percent Error (SMAPE)**.

To visualize the performance of the cross-validated forecasts, use the `plot_cross_validation_metric` function, which takes the cross-validation DataFrame as input:

```
from prophet.plot import plot_cross_validation_metric
plot_cross_validation_metric(df_cv,
                             metric='rmse',
                             rolling_window=0.1,
                             point_color='black');
```

The `plot_cross_validation_metric` generates a plot displaying the **RMSE** values across various forecast horizons evaluated during the cross-validation process. It includes a smoothed line representing the trend, which is calculated as an aggregate (average) over a rolling window (the default is 10% of the data). A larger rolling window (e.g., `rolling_window=0.2`) will result in a smoother line.

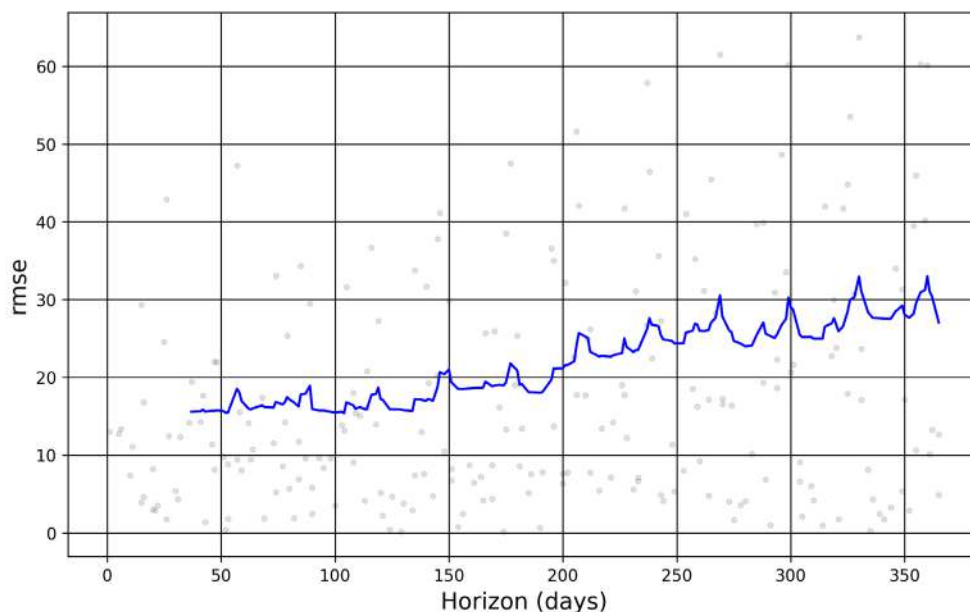


Figure 10.9: Plot showing the model's RMSE over different forecast horizons

The dots represent individual RMSE values for each forecast horizon across all cross-validation cutoffs. The plot indicates that the model performs better at shorter forecast horizons (e.g., up to ~50 days), as evidenced by relatively low RMSE values. However, as the forecast horizons increase (e.g., beyond ~200 days), the RMSE generally increases, indicating reduced reliability for long-term forecasts. Generally, we expect the model to perform better with lower errors for short-term forecasts (1–3 months).

Incorporating holidays and regressors

In this section, you will use a weekly sales dataset to build Prophet models while incorporating holidays and additional regressors. The dataset contains the following columns:

- **Holiday_Flag:** Indicates whether a week corresponds to a special holiday (1) or not (0)
- **Temperature:** Average temperature for the week
- **Fuel_Price:** Average fuel price during the week
- **CPI:** The consumer price index (an economic indicator)
- **Unemployment:** Unemployment rate
- **Weekly_Sales:** Sales for the week (target variable)
- **Date:** The date corresponding to each week

1. Start by loading the `weekly_sales.csv` file and renaming the `Date` and `Weekly_Sales` columns to match Prophet's requirements:

```
retail = Path('../datasets/Ch10/weekly_sales.csv')
retail_df = pd.read_csv(retail, parse_dates=['Date'])
retail_df.rename(columns={'Date': 'ds', 'Weekly_Sales': 'y'},
                 inplace=True)
```

2. Plot the data to gain some intuition about any visible seasonal patterns:

```
retail_df.plot(x='ds', y='y', ylabel='Weekly Sale');
```

This should produce a line plot showing noticeable repeating patterns, such as spikes in weekly sales around the end of November (Thanksgiving) and late December:

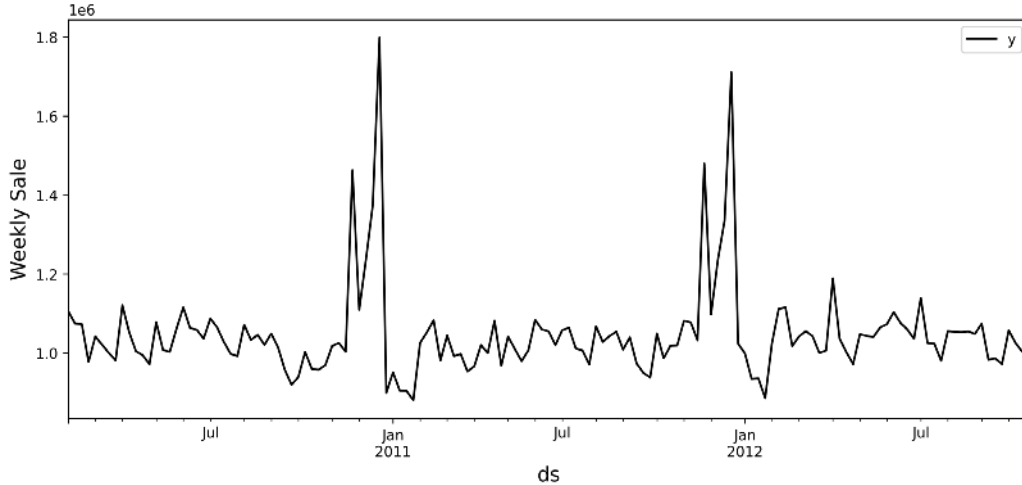


Figure 10.10: Time series plot showing weekly sales patterns over time

3. Split the data into train and test sets using the `split_data` function:

```
# 12 weeks test and remaining 131 weeks for training
test_split = 12
retail_train, retail_test = split_data(retail_df, test_split)
```

4. Create the first Prophet model (a baseline model) without incorporating holidays or regressors:

```
model_retail_v1 = Prophet(seasonality_mode='additive').fit(retail_train)
retail_future = model_retail_v1.make_future_dataframe(
    len(retail_test), freq='W-FRI'
)
retail_forecast_v1 = model_retail_v1.predict(retail_future)
```

5. Visualize the forecast, model components, and perform cross-validation to evaluate the model and capture the scores, as shown in the figure that follows:

```
model_retail_v1.plot(retail_forecast_v1);
```

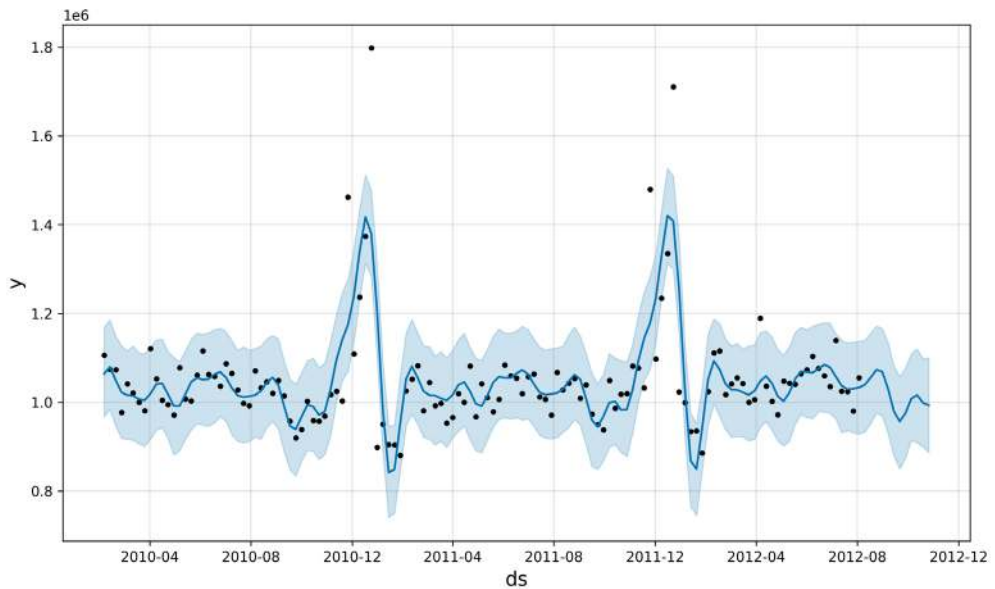


Figure 10.11: Forecast plot using a baseline Prophet model

6. Plot the identified components, as shown in the figure that follows:

```
model_retail_v1.plot_components(retail_forecast_v1);
```

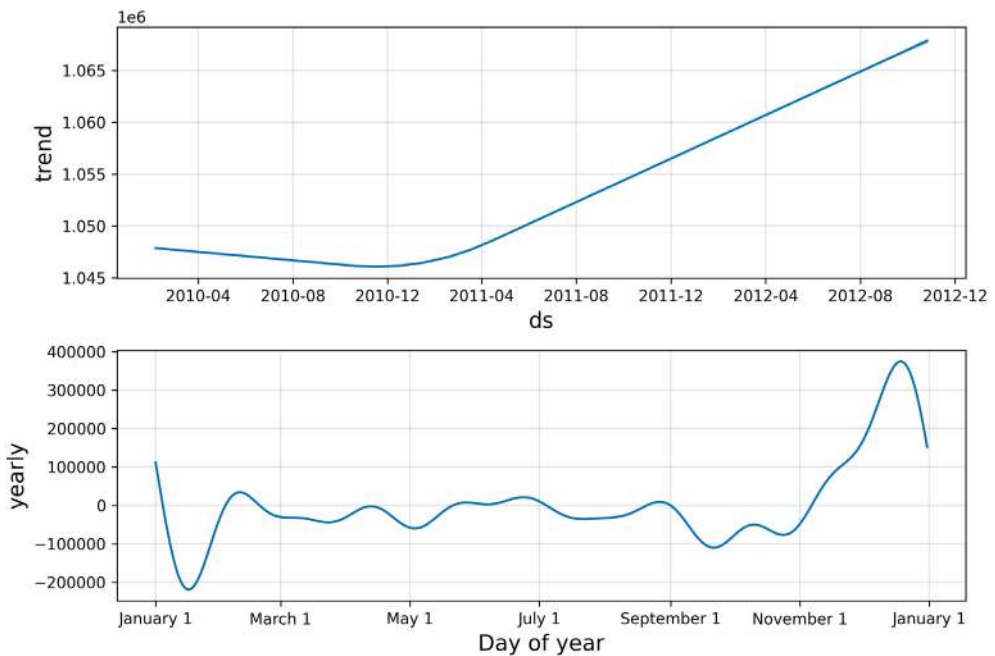


Figure 10.12: Identified components showing trend and yearly seasonality

- Evaluate performance using cross-validation. Set the initial training period to 2 years (730 days) to ensure sufficient data for learning trends and seasonality patterns, with a month (30 days) step forward, and a forecast horizon of approximately 2 weeks (12 days):

```
retail_cv_v1 = cross_validation(model_retail_v1,
                               initial='730 days',
                               period='30 days',
                               horizon='12 days')
```

- Capture the mean performance metrics across all horizons for easier comparison:

```
retail_perf_v1 = performance_metrics(retail_cv_v1)
retail_perf_v1.mean()[1:-1]

>>
mse      986288946.425682
rmse      25589.66785
mae      25589.66785
mape       0.02515
mdape     0.02515
smape     0.025007
```

Now, build the second model that incorporates holidays and regressors such as Temperature, Fuel Price, CPI, and Unemployment rate. These regressors capture external factors that might influence weekly sales.

- Start by creating a holidays DataFrame based on Holiday_Flag:

```
from prophet.make_holidays import make_holidays_df

holidays = pd.DataFrame({
    'holiday': 'custom holiday', # Name of the holiday
    'ds': retail_train[retail_train['Holiday_Flag'] == 1]['ds'],
})
```

Prophet also includes a built-in collection of holidays for each country. The `make_holidays_df` function generates built-in national holiday dates for a specified country (e.g., the U.S.) based on the years in your dataset:

```
years = retail_df['ds'].dt.year.unique().tolist()
us_holidays = make_holidays_df(year_list=years, country='US')
```

- Merge the custom holidays from the dataset (Holiday_Flag) with the built-in holidays to create a comprehensive holiday DataFrame:

```
holidays = pd.concat([holidays, us_holidays])
```

3. Create the second model, `model_retail_v2`, to incorporate holidays and the additional regressors:

```
model_retail_v2 = Prophet(seasonality_mode='additive', holidays=holidays)

model_retail_v2.add_regressor('Temperature')
model_retail_v2.add_regressor('Fuel_Price')
model_retail_v2.add_regressor('CPI')
model_retail_v2.add_regressor('Unemployment')
```

4. Fit the model and make predictions, as shown in the figure that follows:

```
model_retail_v2.fit(retail_train)

retail_future_v2 = retail_future.merge(retail_df[['ds',
                                                'Temperature',
                                                'Fuel_Price',
                                                'CPI',
                                                'Unemployment']],
                                      on='ds', how='left')

retail_forecast_v2 = model_retail_v2.predict(retail_future_v2)

model_retail_v2.plot(retail_forecast_v2);
```

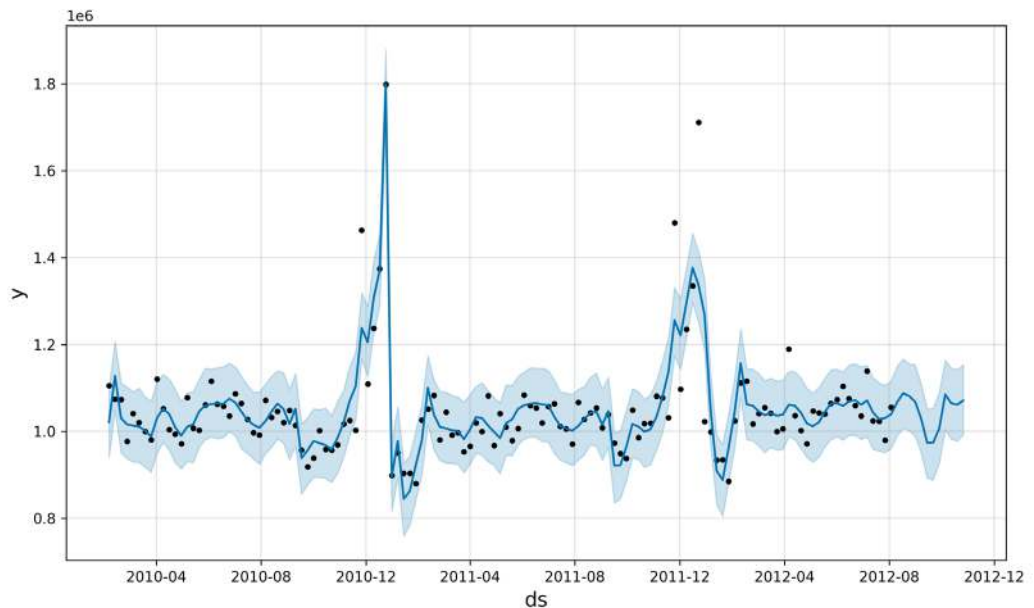


Figure 10.13: Forecast of weekly sales using Prophet with holidays and regressors

5. Plot the identified components, as shown in the figure that follows:

```
model_retail_v2.plot_components(retail_forecast_v2);
```

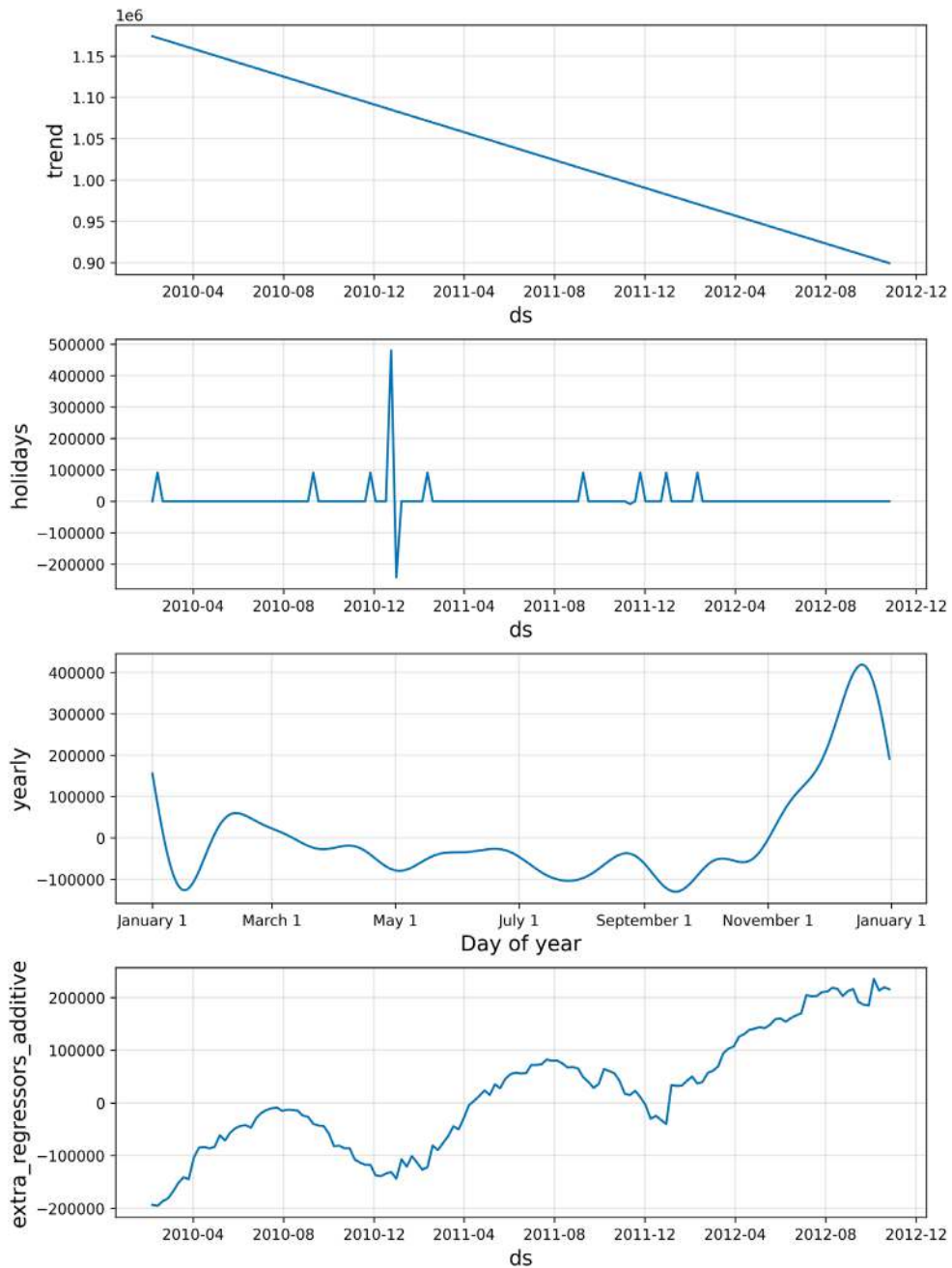


Figure 10.14: Decomposition of components: trend, holidays, yearly seasonality, and extra regressors

Figure 10.14 shows the decomposed components of the model after incorporating holidays and regressors:

- **Trend component:** Shows a gradual decline over time, reflecting the underlying long-term behavior of weekly sales after removing holiday effects and external factors. Previously (baseline model), holiday-related spikes and regressors inflated the trend, but these are now captured separately in other components.
 - **Yearly seasonality:** Shows a clear seasonal pattern with noticeable peaks around November and December, corresponding to holiday periods such as Thanksgiving and Christmas.
 - **Holiday component:** Shows spikes and dips corresponding to specific holiday effects.
 - **Extra regressors component:** Captures the combined impact of external factors such as economic indicators (CPI, fuel price, and unemployment) and environmental conditions (temperature). It shows an increasing trend over time with greater variability, highlighting how these factors influence weekly sales.
6. Finally, evaluate the model using cross-validation and capture the mean performance metrics across all horizons for easier comparison:

```
retail_cv_v2 = cross_validation(model_retail_v2,
                               initial='730 days',
                               period='30 days',
                               horizon='12 days')
retail_perf_v2 = performance_metrics(retail_cv_v2)
retail_perf_v2.mean()[1:-1]
>>
mse      1004511335.583567
rmse      25087.65871
mae      25087.65871
mape      0.024849
mdape     0.024849
smape     0.024532
```

7. The enhanced model (`model_retail_v2`) shows a slight improvement in mean performance metrics compared to the first baseline model (`model_retail_v1`). To visualize this comparison, plot the forecasts from both models against actual weekly sales data, as shown in the figure that follows:

```
ax = retail_test.plot(x='ds',
                     y='y',
                     style='--',
                     label='actual',
                     alpha=0.15,
                     title='Forecasts vs Actual',
```

```

        ylabel='Weekly Sales')
    retail_forecast_v2.tail(12).plot(x='ds', y='yhat', ax=ax,
                                     label='model_retail_v2',
                                     style='-')
    retail_forecast_v1.tail(12).plot(x='ds', y='yhat', ax=ax,
                                     label='model_retail_v1',
                                     style='-.');

```

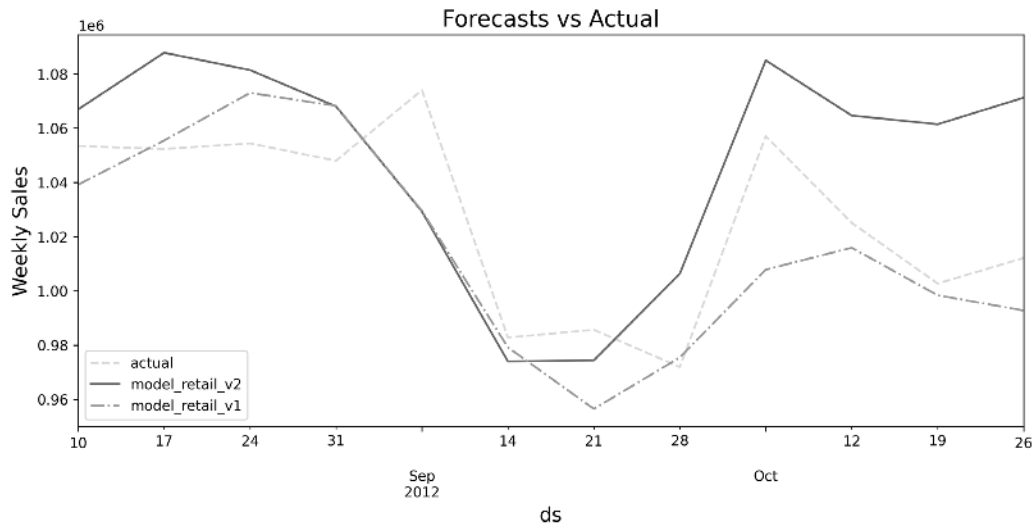


Figure 10.15: Comparing forecasts from the baseline model (*model_retail_v1*) and enhanced model (*model_retail_v2*) against actual weekly sales

In Figure 10.15, you can observe that the second model, *model_retail_v2*, captures holiday-related spikes more effectively than the baseline model, *model_retail_v1*.

Overall, incorporating holidays and additional regressors can improve the model's ability to capture complex patterns in weekly sales and can improve interpretability by isolating holiday effects and external factors.

See also

- Prophet supports both Python and R. For more information on the Python API, please visit the following documentation: https://facebook.github.io/prophet/docs/quick_start.html#python-api.
- If you are interested in reading the original paper behind the Prophet algorithm, which is publicly available, go to <https://peerj.com/preprints/3190/>.
- Cross-validation can also be used for fine-tuning the hyperparameters of the model. You can learn more about this here: <https://facebook.github.io/prophet/docs/diagnostics.html#hyperparameter-tuning>.

Forecasting univariate time series data with Theta

The **Theta model** was introduced by *Assimakopoulos and Nikolopoulos (2000)* and gained popularity after performing well in the M3 forecasting competition, making it a great choice for both trend and seasonal forecasting. Its ease of use and performance make the Theta model a practical option when you're looking for a straightforward method for short- to medium-term forecasting, especially when working with time series that exhibit smooth patterns.

In this recipe, you will explore the **Theta model**, a simple yet powerful approach for forecasting **univariate** time series, particularly in handling trends. The Theta model works by decomposing a time series into two components or lines (referred to as **Theta lines**):

- The long-term trend component captures the overall direction of the series and is modeled using linear regression techniques.
- The short-term component captures local fluctuations or variations in the data and is modeled using **Simple Exponential Smoothing (SES)**. SES adapts dynamically to local changes, making it ideal for short-term dynamics.

The two components are forecasted separately and then combined with equal weights to produce the final forecast. What makes the Theta model particularly powerful is how it balances the stability of long-term trends with the adaptiveness of exponential smoothing for short-term dynamics.

You will use the `ThetaModel` implementation from the `statsmodels` library and apply it to the *Milk Production* dataset. We introduced and extensively used the `statsmodels` library in *Chapter 9*.

Getting ready

You can find the Jupyter notebooks and necessary datasets in this book's GitHub repository. Please refer to the *Technical requirements* section of this chapter for more information.

You will be using the `statsmodels` library. To install using `conda`, you can use the following:

```
conda install -c conda-forge statsmodels
```

To install with `pip`, you can use this:

```
pip install statsmodels
```

As of this writing, the latest version of `statsmodels` is 0.14.5. You can use a more recent version if available.

How to do it...

You will start by loading the *Milk Production* dataset and splitting it into train and test sets, as you did in the previous recipe, *Forecasting univariate time series data using Prophet*:

1. Start by importing the necessary libraries:

```
from statsmodels.tsa.forecasting.theta import ThetaModel
```

2. Load the `milk_production.csv` file and split it into *train* and *test* sets. Hold out the last 12 months as the test set using the `test_split()` function:

```
milk_file = Path('.././datasets/Ch10/milk_production.csv')
milk = pd.read_csv(milk_file,
                  parse_dates=['month'],
                  index_col='month')

test_split = 12
train, test = split_data(milk, test_split)

print(f'Train: {train.shape}')
print(f'Test: {test.shape}')
>>
Train: (156, 1)
Test: (12, 1)
```

3. Create an instance of the Theta model with default parameters:

```
model = ThetaModel(train, deseasonalize=True)
```

The `deseasonalize=True` parameter instructs the model to check for a seasonal component in the time series. If seasonality is detected, the model will remove the seasonal pattern, fit the Theta model on the **deseasonalized** data, and later reintroduce the seasonal component for the final forecast.

4. Fit the model on the training dataset and display the summary, as shown in the figure that follows:

```
results = model.fit()
print(results.summary())
```

```

                                ThetaModel Results
=====
Dep. Variable:                production    No. Observations:                156
Method:                      OLS/SES      Deseasonalized:                  True
Date:                        Mon, 03 Nov 2025    Deseas. Method:                Multiplicative
Time:                        23:26:31          Period:                          12
Sample:                      01-01-1962
                                - 12-01-1974

    Parameter Estimates
    =====
              Parameters
    -----
b0          1.820835909069458
alpha 0.9997934075172108
    =====
```

Figure 10.16: Summary of the fitted Theta model showing estimated parameters for trend and SES components

The fitted model is summarized in the output, which shows details such as whether the seasonality is modeled as **multiplicative** or **additive**. By default, the method used for seasonal decomposition is set to `method='auto'`, allowing the model to automatically select the best approach for **decomposition**. The method used for seasonal decomposition was **multiplicative**.

5. You can view specific parameters by accessing the `params` attribute:

```
results.params
>>
b0      1.820836
alpha   0.999793
dtype: float64
```

The `alpha` parameter is the smoothing parameter used by the **SES**. It determines the weight given to the most recent observation versus past observations in the time series. An `alpha` value closer to 1 gives more weight to recent observations, while values closer to 0 place more emphasis on past (historical) patterns. An `alpha` value of 0.999793 suggests that the model is putting heavy weight on the most recent observations in the **SES** component. The `beta` (`b0`) parameter, estimated using **Ordinary Least Squares (OLS)**, represents the slope of the **trend** component.

6. Generate a forecast using the `forecast` method for the next 12 months. The `forecast` method takes two parameters: the `steps` parameter defines the number of time steps ahead to forecast, and the `theta` parameter is used to influence how the two lines (**Trend** and **SES**) are combined to produce the forecast. In other words, the `theta` parameter controls how much emphasis is placed on different components (Theta lines) of the time series:

```
forecast = results.forecast(steps=n, theta=2)
forecast_df = pd.Series(forecast.tolist(), index=test.index)
```

By default, `theta=2` achieves a balanced combination between the trend and short-term component (**SES**). Generally, the default, `theta=2`, is a good choice for many time series because it captures both long-term trends and short-term variations effectively.

7. Visualize the forecast against the actual data:

```
idx = test.index
plt.plot(idx, test['production'], label='Actual')
plt.plot(idx, forecast_df,
         label='Theta Forecast',
         linestyle='--')
plt.title('Milk Production: Actual vs Theta Forecast')
plt.xlabel('Month')
plt.ylabel('Production')
plt.legend()
plt.show()
```

This should produce the following plot:

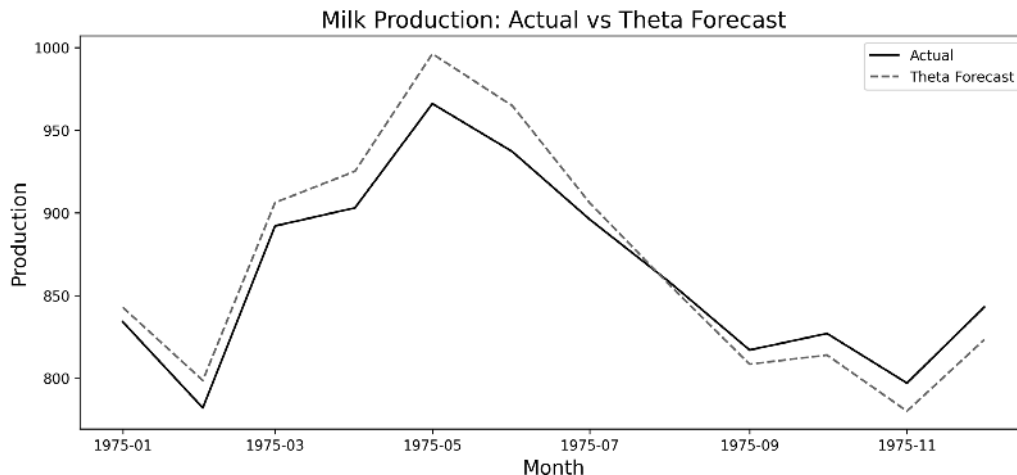


Figure 10.17: Actual versus forecasted milk production using the Theta model

This plot compares the actual values of milk production with the forecasted values generated by the Theta model, providing a visual assessment of the model's performance.

How it works...

In this recipe, you used the Theta model from the `statsmodels` library to forecast univariate time series data. The Theta model works by decomposing the time series into two components, known as Theta lines. These lines capture different aspects of the data's behavior: the long-term direction (trend) and the short-term fluctuations. These two are then combined to create a balanced forecast, with the balance controlled by the `theta` parameters.

Here is how things work behind the scenes:

- **Deseasonalization:** Before applying the Theta decomposition, the model first checks for seasonality using the **Autocorrelation Function (ACF)** test at seasonal lags to detect regular patterns (e.g., lag 12 for monthly data). If seasonality is detected, the model automatically chooses between multiplicative or additive decomposition based on characteristics of the data to remove the seasonal components before fitting the Theta model. The removal of seasonal components allows the Theta model to focus on trend and short-term dynamics.
- **Decomposition into Theta lines:** Once any seasonal patterns are removed, the model creates two specialized components (Theta lines). The first Theta line, the Trend component, is estimated using a **linear regression (OLS)**. The line captures the overall direction of the data. The second Theta line, the short-term component, captures local patterns or short-term fluctuations using SES.
- **Combining for forecasting:** The `theta` parameter controls the balance on how these two components are combined in the forecast (how they are weighted).
- **Reseasonalization:** After generating the combined forecast, the model reintroduces the seasonal pattern that was removed earlier either additively or multiplicatively.

To capture the estimated uncertainty around the forecast, you can generate **confidence intervals** using the `prediction_intervals` method.

Overall, the Theta model is an effective and simple forecasting model that combines linear regression (for estimating trends) and exponential smoothing (for capturing short-term variations), making it great at handling trends and being adaptable to seasonality.

There's more...

The Theta model provides several useful tools for understanding and visualizing your forecasts through the **ThetaModelResults** object, which gives access to three essential methods:

- `forecast_components` provides the individual components (trend, SES, and seasonal) used in the forecast
- `prediction_intervals` allows you to compute confidence intervals for the forecast
- `plot_predict` is a convenient method to visualize the forecast along with its confidence intervals

The `forecast_components` method allows you to inspect the different components (trend, SES, and seasonal) of the forecast:

```
components = results.forecast_components(steps=n)
print(components)
>>
```

	trend	ses	seasonal
157	1.821212	862.63851	0.975930
158	3.642048	862.63851	0.923796
159	5.462884	862.63851	1.047136
160	7.283720	862.63851	1.067902
161	9.104556	862.63851	1.148714
162	10.925392	862.63851	1.111344
163	12.746228	862.63851	1.042552
164	14.567064	862.63851	0.984282
165	16.387899	862.63851	0.928328
166	18.208735	862.63851	0.933702
167	20.029571	862.63851	0.893855
168	21.850407	862.63851	0.942458

Each component reflects a part of the overall forecast:

- The trend component shows a steady increase, reflecting long-term growth in milk production
- The SES component remains constant because it captures the current level of the series and reflects the smoothing level
- The seasonal component shows recurring patterns in milk production

You can calculate prediction intervals to capture the uncertainty in the forecast. For instance, to get the lower and upper bounds of the confidence interval, you can use the `prediction_intervals` method, as shown:

```
intervals = results.prediction_intervals(steps=n, theta=2)
print(intervals)
>>
```

	lower	upper
157	754.502241	931.025494
158	673.763431	923.405012
159	753.286267	1059.033515
160	748.579355	1101.625867
161	798.795497	1193.513498
162	748.562726	1180.954631
163	672.471536	1139.508173
164	606.607354	1105.890521
165	543.633580	1073.203349
166	534.840716	1093.056267
167	487.294532	1072.755940
168	517.549714	1129.044213

This will return the lower and upper bounds of the intervals based on the `theta=2` model. By default, a 95% confidence interval ($\alpha=0.05$) is calculated. You can adjust this through the `alpha` parameter.

The `plot_predict` method provides a convenient way to visualize your forecast along with the confidence intervals. Here is an example:

```
results.plot_predict(steps=n, theta=2, figsize=(10,4), alpha=0.05);
```

This will generate a plot similar to *Figure 10.18*, showing forecasted values and their confidence interval (95% confidence interval):

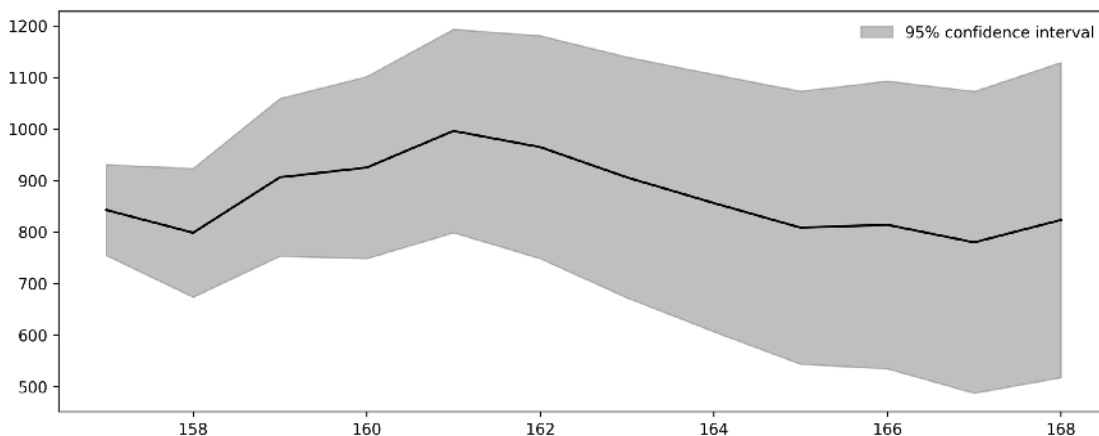


Figure 10.18: Forecast plot generated by `plot_predict` showing forecast and confidence intervals

This plot in *Figure 10.18* provides additional insight into the forecast:

- The solid line shows the expected future values
- The shaded area represents the confidence interval, showing where we expect the actual values to fall
- The wider the shaded area gets, the more uncertainty exists in our predictions further into the future

You can adjust the confidence level by adjusting the `alpha` parameter in `plot_predict` (the default is `alpha=0.05`, which corresponds to a 95% confidence interval).

See also

- To learn more about the **ThetaModel** implementation in `statsmodels`, you can check the official documentation here: <https://www.statsmodels.org/dev/examples/notebooks/generated/theta-model.html>.

Forecasting multivariate time series data using VAR

In this recipe, you will explore the VAR model for working with multivariate time series. In *Chapter 9*, we discussed AR, MA, ARIMA, and SARIMA as examples of univariate one-directional models. VAR, on the other hand, captures bi-directional dynamics among multiple time series.

VAR vs. AR models



You can think of VAR of order p , or $\text{VAR}(p)$, as a generalization of the univariate $\text{AR}(p)$ model made for working with multiple time series. While $\text{AR}(p)$ captures how a single variable depends on its own p past values, $\text{VAR}(p)$ captures how multiple variables depend on each other's p past values. These multiple time series are represented as a vector, hence the name vector autoregressive. For example, a VAR model with one lag, written as $\text{VAR}(1)$, describes how each variable in the system depends on its own previous value and the previous values of all other variables in the system.

The VAR model extends univariate autoregression to multivariate systems by capturing relationships among multiple time series variables. By leveraging these interdependencies, VAR can enhance forecasting accuracy.

There are other forms of multivariate time series models, including **Vector Moving Average (VMA)**, **Vector Autoregressive Moving Average (VARMA)**, and **Vector Autoregressive Integrated Moving Average (VARIMA)**, that generalize other univariate models. In practice, you will find that VAR is used the most due to its simplicity. VAR models are very popular in economics, but you will find them used in other areas, such as social sciences, natural sciences, and engineering.

The premise behind multivariate time series is that you can add more power to your forecast when leveraging multiple time series (or input variables) instead of a single time series (single variable). Simply put, VAR is used when you have two or more time series that have (or are assumed to have) an influence on each other's behavior.

These are normally referred to as **endogenous** variables, and relationship is bi-directional (e.g., spending affects income, and income affects spending), meaning that each variable can influence and be influenced by the others. However, there are often variables that influence the system but are **not** influenced by it. We refer to these as **exogenous** variables. They act as external inputs, driving changes in the main system without being affected by the feedback loop.

Exogenous vs. endogenous variables

When researching VAR models, you will encounter references to **endogenous** and **exogenous** variables. At a high level, these two types of variables are opposites, and in Python's `statsmodels` library, they are referred to as `endog` (endogenous) and `exog` (exogenous), respectively.

Endogenous variables are those influenced by other variables within the system. In other words, we expect that a change in one endogenous variable is expected to affect others within the same system. For example, GDP and inflation are often modeled as endogenous variables in an economic system because they influence each other; GDP growth can drive inflation through increased demand, while inflation can impact GDP by affecting purchasing power or investment decisions. To determine whether there is a predictive relationship between multiple endogenous variables, you can use the **Granger causality test**, which examines whether past values of one variable can predict another variable's future values beyond what that variable's own past values predict. In `statsmodels`, this can be implemented using the `grangercausalitytests` function.

On the other hand, **exogenous** variables originate outside the system. While they exert an influence on the variables within the system, they are not influenced by them. Instead, they act as external drivers. For example, weather conditions might be considered exogenous when modeling mall traffic because they influence foot traffic but are not influenced by it. Weather originates outside the system being modeled and acts as an external driver.

A VAR model, like an AR model, assumes the **stationarity** of the time series variables. This means that each endogenous variable (time series) needs to be stationary.

To illustrate how VAR works and the mathematical equation behind it, let's start with a simple VAR(1) model with two endogenous variables, referred to as $(y_{1,t}, y_{2,t})$. Recall from *Chapter 9* that an AR(1) model would take the following form:

$$AR(1) = y_t = \alpha + \theta_1 y_{t-1} + \epsilon_t$$

Generally, an AR(p) model is a linear model of past values of itself, and the (p) parameter tells us how far back we should go. Now, assume you have **two** AR(1) models for two different time series data. This will look as follows:

$$y_{(1),t} = \alpha_1 + \theta_{11} y_{(1),t-1} + \epsilon_{(1),t}$$

$$y_{(2),t} = \alpha_2 + \theta_{21} y_{(2),t-1} + \epsilon_{(2),t}$$

However, these are two separate models that do not show any relationship or influence on each other. If we create a **linear combination** of the two models (the past values of itself and the past values of the other time series), we would get the following formula:

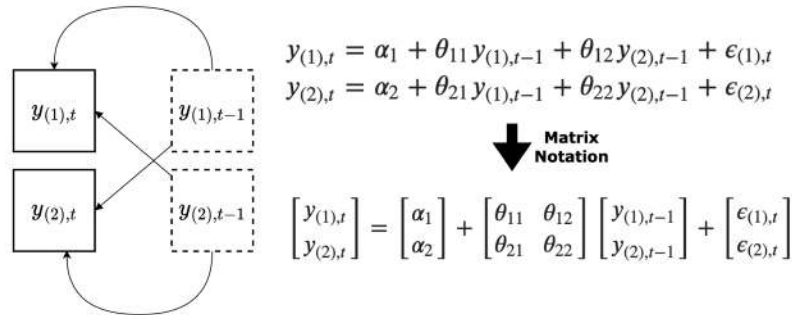


Figure 10.19: Formula for a VAR model with lag one or VAR(1) for two variables

The equation in Figure 10.19 represents a VAR(1) model with two variables, $y_{(1),t}$ and $y_{(2),t}$. The equation may seem complex at first glance, but like an AR model, it is fundamentally a linear function of past lags. In other words, in a VAR(1) model, you will have a linear function of lag (1) for each variable. In a VAR(1) model, each variable depends not only on its own past values but also on the past values of other variables in the system.

For instance, in Figure 10.19, notice how $y_{(1),t}$ depends not just on its own lagged (past) value ($y_{(1),t-1}$) like in AR(1), but also on the past of y_2 ($y_{(2),t-1}$). Similarly, $y_{(2),t}$, depends on both its own lagged (past) value ($y_{(2),t-1}$) and y_1 's past ($y_{(1),t-1}$). These cross-terms, $\theta_{12}y_{(2),t-1}$ and $\theta_{21}y_{(1),t-1}$, are what makes VAR models special as they capture how variables influence each other over time, enabling us to model interdependence effectively.

When fitting a VAR model, as you shall see in this recipe, the OLS method is used to estimate the parameters for each equation in the system. Each equation represents how one variable depends on the past values of all variables in the system (including itself). This means we're essentially solving a system of linear equations, where the coefficients capture both self-dependence and cross-dependence among variables.

Getting ready

You can find the Jupyter notebooks and necessary datasets in this book's GitHub repository. Please refer to the *Technical requirements* section of this chapter for more information.

For this recipe, you will use the `pandas_datareader` library to download data from **Federal Reserve Economic Data (FRED)**. The files are also available for you to download from the GitHub repo of this book.

To install using `pip`, use this:

```
pip install pandas-datareader
```

How to do it...

In this recipe, you will use the `FredReader()` function from the `pandas_datareader` library to pull two different time series datasets. As mentioned on FRED's website, the first symbol, `FEDFUNDS`, is the federal funds effective rate, which *"is the interest rate at which depository institutions trade federal funds (balances held at Federal Reserve Banks) with each other overnight."* Simply put, the federal funds effective rate influences the cost of borrowing. It is *"the target interest rate set by the Federal Open Market Committee (FOMC) for what banks can charge other institutions for lending excess cash from their reserve balances."* The second symbol is `unrate`, for the unemployment rate, which is the percentage of the total labor force that is unemployed but actively seeking employment or willing to work.

Citations



Board of Governors of the Federal Reserve System (US), Federal Funds Effective Rate [FEDFUNDS], retrieved from FRED, Federal Reserve Bank of St. Louis: <https://fred.stlouisfed.org/series/FEDFUNDS>, October 6, 2024.

U.S. Bureau of Labor Statistics, Unemployment Rate [UNRATE], retrieved from FRED, Federal Reserve Bank of St. Louis: <https://fred.stlouisfed.org/series/UNRATE>, October 6, 2024.

Follow these steps:

1. Start by loading the necessary libraries and pulling the data. Note that both `FEDFUNDS` and `unrate` are reported monthly:

```
import pandas_datareader.data as web
from statsmodels.tsa.api import VAR, adfuller, kpss
from statsmodels.graphics.tsaplots import (plot_acf,
                                           plot_pacf,
                                           plot_ccf)
from statsmodels.tsa.stattools import grangercausalitytests
```

2. Pull the data using `FredReader`, which wraps over the FRED API and returns a pandas DataFrame. For the `FEDFUNDS` and `unrate` symbols, you will pull close to 35 years' worth of data (421 months):

```
import pandas_datareader.data as web
start = "01-01-1990"
end = "01-01-2025"
economic_df = web.FredReader(
    symbols=["FEDFUNDS", "unrate"],
    start=start,
    end=end
).read()
```

```
file = '../datasets/Ch10/economic_df.pickle'
economic_df.to_pickle(file)
```

Store the DataFrame as a pickle object, as shown in the last line of the preceding code. This way, you do not have to make an API call to rerun the example. You can read the `economic_df.pickle` file using `economic_df = pd.read_pickle(file)`.

3. Inspect the data and make sure there are no missing values:

```
economic_df.isna().sum()
>>
FEDFUNDS    0
unrate      0
dtype: int64
```

Change the index frequency of the DataFrame to month start ('MS') to reflect how the data is being stored:

```
economic_df.index.freq = 'MS'
```

4. Plot the datasets for visual inspection and understanding:

```
economic_df.plot(subplots=True, sharey=True);
```

5. With `subplots=True`, the output will contain two subplots, one for each column:

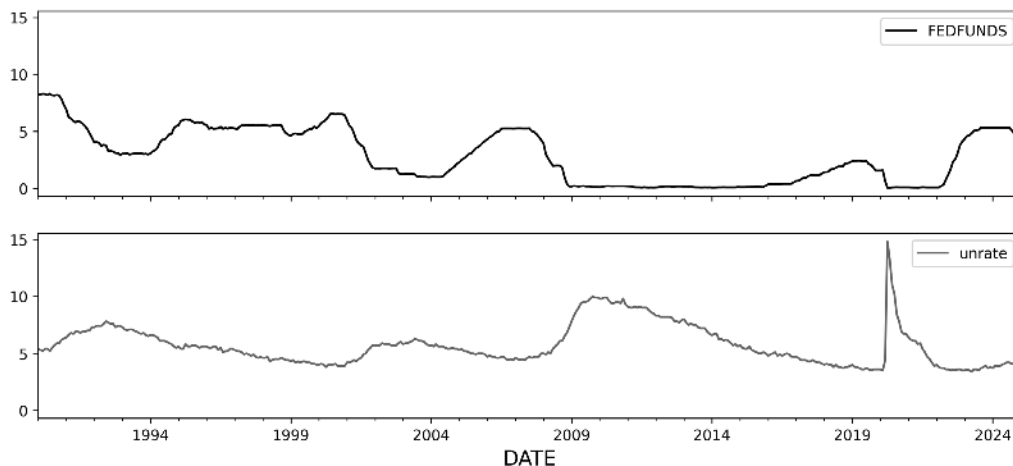


Figure 10.20: Plotting both *FEDFUNDS* and *unrate*

There is some sort of inverse relationship between *FEDFUNDS* and *unrate* – as *FEDFUNDS* increases, *unrate* decreases. There is an interesting anomalous behavior starting in 2020 due to the COVID-19 pandemic.

6. We can further check the correlation between the variables:

```
correlation_matrix = economic_df.corr()
correlation_matrix
>>
```

	FEDFUNDS	unrate
FEDFUNDS	1.000000	-0.435171
unrate	-0.435171	1.000000

The correlation between FEDFUNDS and unrate is -0.435, indicating a moderate negative relationship (inverse correlation). This suggests that as the federal funds rate increases, the unemployment rate tends to decrease, and vice versa. However, correlation does not imply causation, so further analysis is needed to understand the underlying dynamics.

Further, you can perform a **Cross-Correlation Function (CCF)** between FEDFUNDS and unrate to see the correlation at different lags. The CCF helps identify lagged relationships or temporal dependencies between the two time series. The outcome mainly tells us whether one series leads or lags the other. It is not a formal test of causality, which you will investigate later:

```
lags = np.arange(-12, 13)
plot_ccf(economic_df['FEDFUNDS'], economic_df['unrate'], lags=lags)
plt.grid(True)
```

This produces the plot in *Figure 10.21*, with the lag on the x-axis and the estimated correlational coefficient on the y-axis. The CCF plot below shows the correlation between FEDFUNDS and unrate at different time lags, highlighting potential lead-lag relationships.

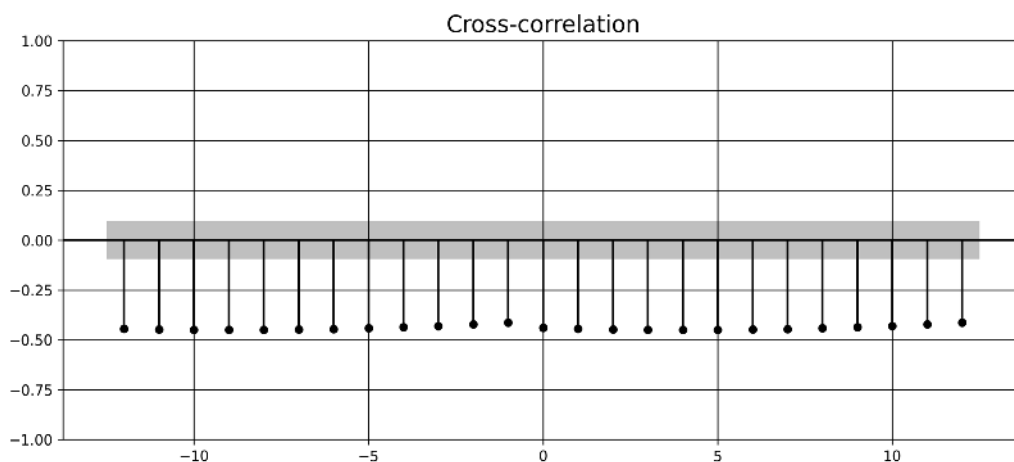


Figure 10.21: CCF plot for 12 months ahead and 12 months behind for better interpretability

The plot shows spikes that extend beyond the shaded confidence interval, which indicates a significant negative correlation. This suggests that changes in FEDFUNDS might be associated with opposite changes in unrate within the same frame.

An important assumption in VAR is **stationarity**. Both variables (the two endogenous time series) need to be stationary.

7. Use the `check_stationarity` function to evaluate each endogenous variable (column):

```
for i in economic_df:
    adf = check_stationarity(economic_df[i])
    print(f'{i} adf: {adf}')

>>
FEDFUNDS adf: Stationary
unrate adf: Stationary
```

Overall, both time series are shown to be stationary.

8. Plot both **ACF** and **PACF** plots to gain an intuition over each variable and which process they belong to – for example, an AR or MA process:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
for col in economic_df.columns:
    fig, ax = plt.subplots(1,2, figsize=(18,4))
    plot_acf(economic_df[col], zero=False,
             lags=30, ax=ax[0], title=f'ACF - {col}')
    plot_pacf(economic_df[col], zero=False,
             lags=30, ax=ax[1], title=f'PACF - {col}');
```

This should produce an ACF and PACF for FEDFUNDS and unrate:

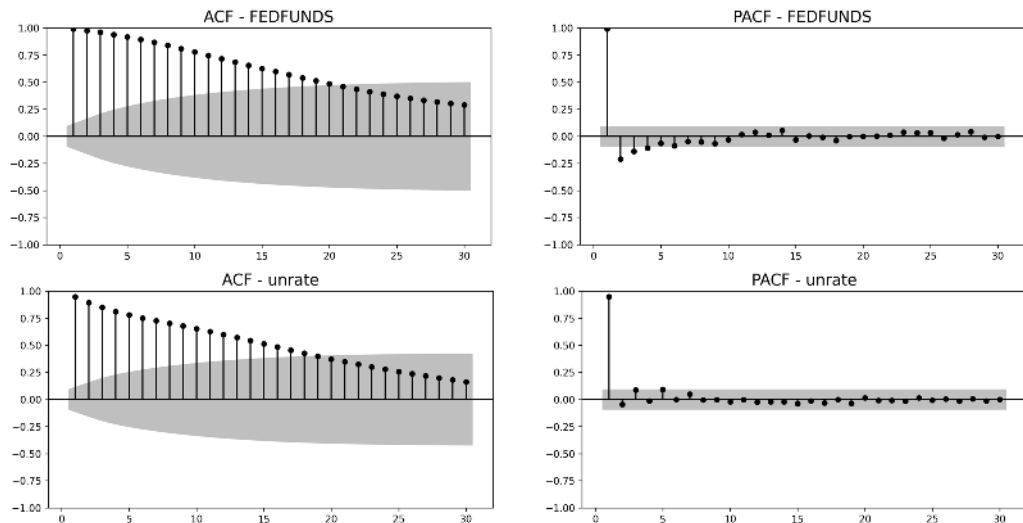


Figure 10.22: ACF and PACF plots for FEDFUNDS and unrate

The ACF and PACF plots for FEDFUNDS and unrates indicate that we are dealing with an autoregressive (AR) process. The ACF plots show a slow, gradual decay, while the PACF plots show a sharp cutoff after lag 1. The PACF for FEDFUNDS shows slightly significant (above or below the shaded area) lags at 2 and 3, while the PACF for unrates shows some significance at lag 24. We can ignore those for now.

9. Split the data into train and test sets:

```
# hold 12 months for testing
train, test = split_data(economic_df, test_split=12)
print(f'Train: {len(train)}, Test: {len(test)}')
>>
Train: 409, Test: 12
```

You can scale the data (**standardization**), even though it is not necessary for VAR, and the two time series are not far off in terms of scale. In this step, you will perform scaling for *illustrative purposes* to show how it can be done and how you inversely transform the results for interpretation later on.



There is often a debate on whether the variables need to be scaled (standardized) when implementing VAR. However, the VAR algorithm does not inherently require the variables to be scaled, as it is **scale-invariant**. In practice, it is common to leave variables in their original units to preserve the interpretability of the coefficients and residuals in meaningful terms (e.g., percentage points). However, if the variables differ significantly in scale, then standardization can be applied to make the coefficients easier to compare and detect **outliers** more effectively.

If you choose to standardize (for example, using `StandardScaler` from **scikit-learn**), it is important to note that the results will be in terms of **standard deviations**. You can use the `inverse_transform` method to revert the data back to its original units when interpreting the results. This is useful when you need to explain the findings in the context of the original variable scale.

10. Scale the data using `StandardScaler` by fitting the train set using the `fit` method. Then, apply the scaling transformation to both the train and test sets using the `transform` method. The transformation will return a NumPy ndarray, which you will then return as pandas DataFrames. This will make it easier for you to plot and examine the DataFrames further:

```
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()
scale.fit(train)
train_sc = pd.DataFrame(scale.transform(train),
                        index=train.index,
                        columns=train.columns)
test_sc = pd.DataFrame(scale.transform(test),
                      index=test.index,
                      columns=test.columns)
```

But how can you determine the optimal lag order, p , for your VAR model? Luckily, the VAR implementation in `statsmodels` will pick the best VAR order. You only need to define the maximum number of lags (threshold); the model will determine the best p values that minimize each of the four information criteria scores: **AIC**, **BIC**, **FPE**, and **HQIC**. The `select_order` method will compute the scores at each lag order, while the `summary` method will display the scores for each lag. The results will help when you train (fit) the model to specify which information criteria the algorithm should use:

```
model = VAR(endog=train_sc)
res = model.select_order(maxlags=10)
res.summary()
```

This should show the results for all 10 lags. The lowest scores are marked with *:

```
VAR Order Selection (* highlights the minimums)
=====
      AIC      BIC      FPE      HQIC
-----
0    -0.3054   -0.2854    0.7369   -0.2975
1     -7.408    -7.348   0.0006064   -7.384
2     -7.941    -7.841   0.0003559   -7.901
3     -8.026    -7.886*   0.0003269   -7.971*
4     -8.029    -7.850   0.0003257   -7.958
5     -8.022    -7.802   0.0003281   -7.935
6     -8.039    -7.779   0.0003227   -7.936
7     -8.046*    -7.746   0.0003205*   -7.927
8     -8.028    -7.688   0.0003261   -7.894
9     -8.017    -7.638   0.0003297   -7.867
10    -8.004    -7.584   0.0003343   -7.837
-----
```

11. The `res` object is an instance of the `LagOrderResults` class. You can print the selected lag number for each information criterion using the `selected_orders` attribute, which returns a dictionary of the optimal lag orders for **AIC**, **BIC**, **FPE**, and **HQIC**:

```
print(res.selected_orders)
>>
{'aic': 7, 'bic': 3, 'hqic': 3, 'fpe': 7}
```

At lag 7, both *AIC* and *FPE* scores were the lowest. On the other hand, both *BIC* and *HQIC* scores were the lowest at lag 3, suggesting a more parsimonious model with fewer lags.

Generally, *BIC* and *HQIC* tend to favor more **parsimonious** models (i.e., models with fewer lags) because they impose a higher penalty on model complexity. In contrast, *AIC* and *FPE* tend to be more lenient and may suggest higher lag orders since they are less strict in penalizing for complexity.

In this case, lag 3 seems like a safer choice, helping to aim for simplicity and avoiding overfitting. However, choosing lag 7 would result in a more complex model that might capture more dynamics in the data, but it comes with a higher risk of overfitting.

12. To train the model, you must use the *BIC* score (or another criterion of your choice). You experiment with a different information criterion by updating the `ic` parameter. The `maxlags` parameter is optional – if you leave it blank, the model will automatically determine the optimal lag order based on the selected information criterion, which, in this case, is `'bic'`:

```
results = model.fit(ic='bic')
```

This will automatically select the lag order that minimizes the BIC score (i.e., lag 3). You can experiment with other criteria, such as AIC, by setting `ic='aic'`. If you prefer to manually specify the maximum number of lags, you can use the `maxlags` parameter. Keep in mind that if both `maxlags` and `ic` are used, then `ic` will take precedence.

Running `results.summary()` will print a detailed summary, including the regression results for each autoregressive process.

The summary includes information on the number of equations (corresponding to the number of variables), the **information criteria** (AIC, BIC, HQIC, and FPE), and other details such as **log likelihood**. These metrics help assess the overall fit and complexity of the model:

```
Summary of Regression Results
=====
Model:                                VAR
Method:                               OLS
Date:      Tue, 11, Feb, 2025
Time:      13:01:03
-----
No. of Equations:      2.00000      BIC:                                -7.90565
Nobs:                  406.000      HQIC:                               -7.98912
Log likelihood:        494.713      FPE:                                0.000321089
AIC:                   -8.04380      Det(Omega_mle):          0.000310297
-----
```

At the end of the summary, there is a **correlation matrix** of residuals. The matrix shows the correlations between the residuals (errors) from each equation in the VAR model. Ideally, you want these off-diagonal values to be as close to zero as possible. A low correlation suggests that the model has captured most of the relationship between the variables, and there is little leftover correlation in the residuals:

```
Correlation matrix of residuals
      FEDFUNDS      unrte
FEDFUNDS      1.000000 -0.11377
unrate       -0.11377  1.000000
```

The correlation between the residuals for FEDFUNDS and unrte is -0.113677 , which is relatively small, indicating that the model has done a good job capturing the relationship between the two variables.

13. Store the VAR lag order using the `k_ar` attribute so that we can use it later when we use the forecast method:

```
lag_order = results.k_ar
lag_order
>> 3
```

This shows that the optimal lag order selected was 3.

14. Finally, you can forecast using either the `forecast` or `forecast_interval` method. The latter will return the forecast, as well as the upper and lower **confidence intervals**. Both methods will require past values and the number of steps ahead to forecast. The prior values (`past_y`) will be used as the initial values for the forecast:

```
past_y = train_sc[-lag_order:].values
n = test_sc.shape[0]
forecast, lower, upper = results.forecast_interval(past_y, steps=n)
```

15. Since you applied `StandardScaler` on the dataset, the forecast values are scaled. You will need to apply `inverse_transform` to convert them back to the original scale of the data. You can also convert the forecasts and confidence interval into pandas DataFrames for convenience:

```
forecast_df = pd.DataFrame(scale.inverse_transform(forecast),
                           index=test_sc.index,
                           columns=test_sc.columns)
lower_df = pd.DataFrame(scale.inverse_transform(lower),
                        index=test_sc.index,
                        columns=test_sc.columns)
upper_df = pd.DataFrame(scale.inverse_transform(upper),
                        index=test_sc.index,
                        columns=test_sc.columns)
```

16. Plot the actual versus forecasted unrte with the confidence intervals:

```
idx = test.index
plt.figure(figsize=(10, 6))
plt.plot(idx, test['unrate'], label='Actual unrte')
plt.plot(idx, forecast_df['unrate'], label='Forecasted unrte',
         linestyle='dashed')
plt.fill_between(idx, lower_df['unrate'], upper_df['unrate'],
                 alpha=0.2, label='Confidence Interval')
plt.title('Actual vs Forecasted unrte with Confidence Intervals')
plt.legend()
plt.show()
```

This should plot the actual test data, the forecast (middle dashed line) for unrate, along with the upper and lower confidence intervals:

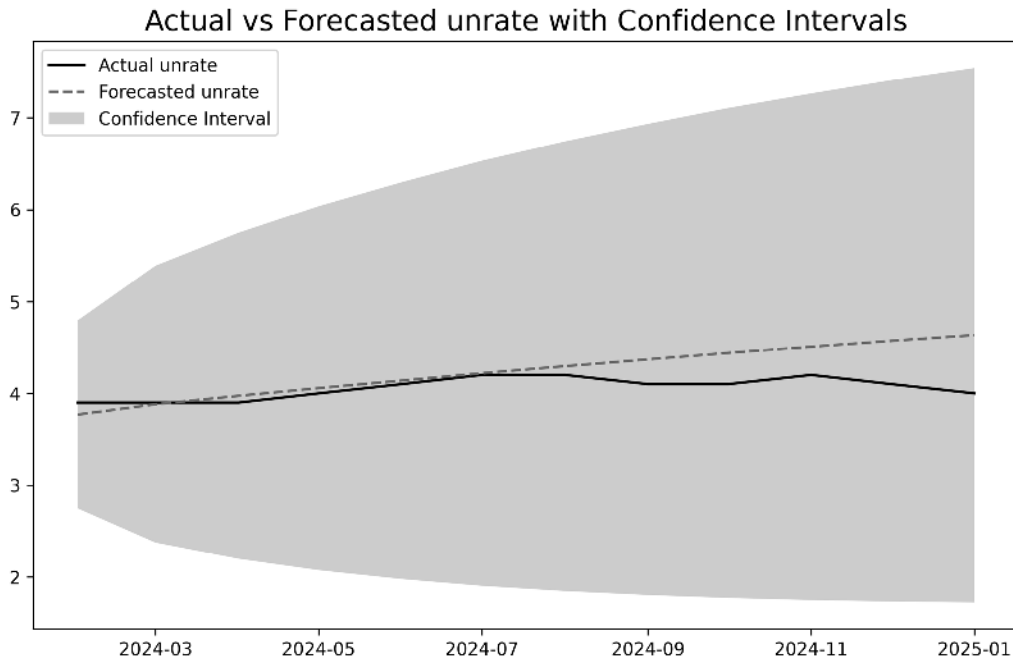


Figure 10.23: Plotting the forecast with confidence intervals for unrate

In a VAR model, all variables are forecasted simultaneously because the model assumes that past values of all variables in the system contribute to the future values of each variable. So, even though you may only be interested in predicting unrate from FEDFUNDS, the model forecasts the future values of FEDFUNDS based on unrate. You can visualize the forecast for FEDFUNDS in a similar way as you did for unrate, as shown in the figure that follows:

```
idx = test.index
plt.figure(figsize=(10, 6))
plt.plot(idx, test['FEDFUNDS'], label='Actual FEDFUNDS')
plt.plot(idx, forecast_df['FEDFUNDS'], label='Forecasted FEDFUNDS',
         linestyle='dashed')
plt.fill_between(idx, lower_df['FEDFUNDS'], upper_df['FEDFUNDS'],
                 alpha=0.2, label='Confidence Interval')
plt.title('Actual vs Forecasted FEDFUNDS with Confidence Intervals')
plt.legend()
plt.show()
```

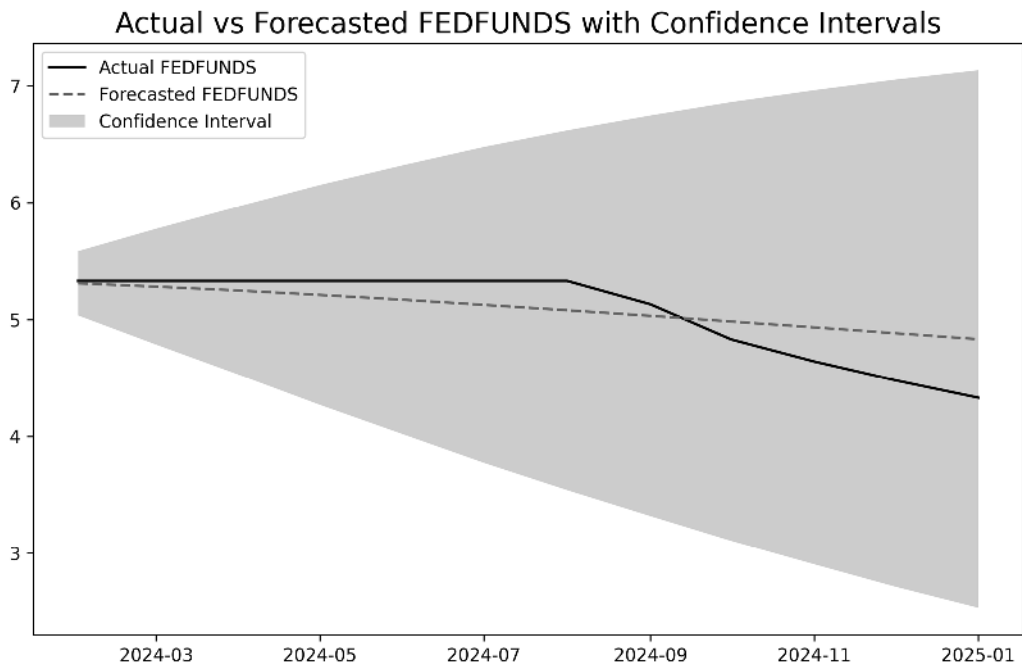


Figure 10.24: Plotting the forecast with confidence intervals for FEDFUNDS



Recall that the training data was set up until the end of 2023, which includes the significant economic disruption caused by COVID-19 in 2020. Thus, the model's predictions may not perform as well as expected due to the large and sudden shifts in the data.

You may need to adjust the train-test split to accommodate this fact and perform different experiments with the model. When dealing with major anomalies such as COVID-19, it is essential to assess whether the event is a **one-time anomaly** whose effects will fade or whether it represents a lasting change that requires special handling in your model.

This is where **domain knowledge** becomes crucial: understanding the economic context will help you decide whether to model such events or treat them as outliers that shouldn't influence future predictions.

How it works...

VAR models are very useful, especially in econometrics. When evaluating the performance of a VAR model, it is common to report results from several important analyses, such as **Granger causality tests**, residual (or error) analysis, and **impulse response** analysis. These are critical for understanding the interactions between variables. You will explore these in the next recipe, *Evaluating VAR models*.

In *Figure 10.19*, a VAR(1) model for two variables results in two equations. Each equation models the current value of one variable as a function of its own lagged values and the lagged values of the other variable. The matrix notation shows four coefficients and two constants. Each equation has two coefficients for the lagged values of both variables (e.g., θ_{11} and θ_{12} for $y_{(1),t}$ and θ_{21} and θ_{22} for $y_{(2),t}$). The constants α_1 and α_2 represent the intercepts for each equation.

In this recipe, the model selected was a VAR(3) model for two variables. The key difference between VAR(1) and VAR(3) is the increased complexity due to additional lagged terms. In the VAR(3) model, you will have 12 coefficients (3 lags per variable) and 2 constants to solve for. Each equation is estimated using OLS, as shown in the `results.summary()` output.

The two functions represented in *Figure 10.19* for VAR(1) show how each variable is influenced not only by its own past values but also by the past values of the second (endogenous) variable. This is a key difference between a VAR model and an AR model (which only considers a variable's own lags). A VAR model captures the dynamic interaction between multiple variables over time.

In the next recipe, now that the model is fitted, you will spend time plotting VAR-specific outputs – for example, the **impulse responses (IRs)** and the **forecast error variance decomposition (FEVD)** – to better understand these interactions and how the variables influence each other.

In this recipe, we focused on using the model for forecasting; next, we will focus on understanding the **causal relationships**, using tools such as Granger causality tests, and evaluating the overall performance of the model through additional diagnostics.



Vector ARIMA (VARIMA) models extend VAR to handle non-stationary data by incorporating **differencing**. We didn't use it here because both time series were stationary, so differencing wasn't needed. Consider VARIMA when dealing with multiple non-stationary variables that require differencing to achieve stationarity.

There's more...

Since we are focusing on comparing forecasting results, it would be interesting to see whether our VAR(3) model, with two endogenous variables, performs better than a simpler univariate AR(3) model. Comparing these models helps assess whether incorporating additional variables (e.g., FEDFUNDS) improves the forecasting accuracy of `unrate`.

Fit an AR(3) or ARIMA(3,0,0) model to the `unrate` time series, using the same lag values for consistency. Since the `unrate` series is stationary, there is no need for differencing. Recall from the previous activity that `lag_order` is equal to 3:

```
from statsmodels.tsa.arima.model import ARIMA
model = ARIMA(train['unrate'], order=(lag_order,0,0)).fit()
```

You can review the ARIMA model's summary using `model.summary()`. After fitting the model, you can evaluate the residuals by plotting diagnostic charts to check for any issues with the model's fit:

```
fig = model.plot_diagnostics(figsize=(12,6));
fig.tight_layout()
plt.show()
```

This should produce four diagnostic subplots:

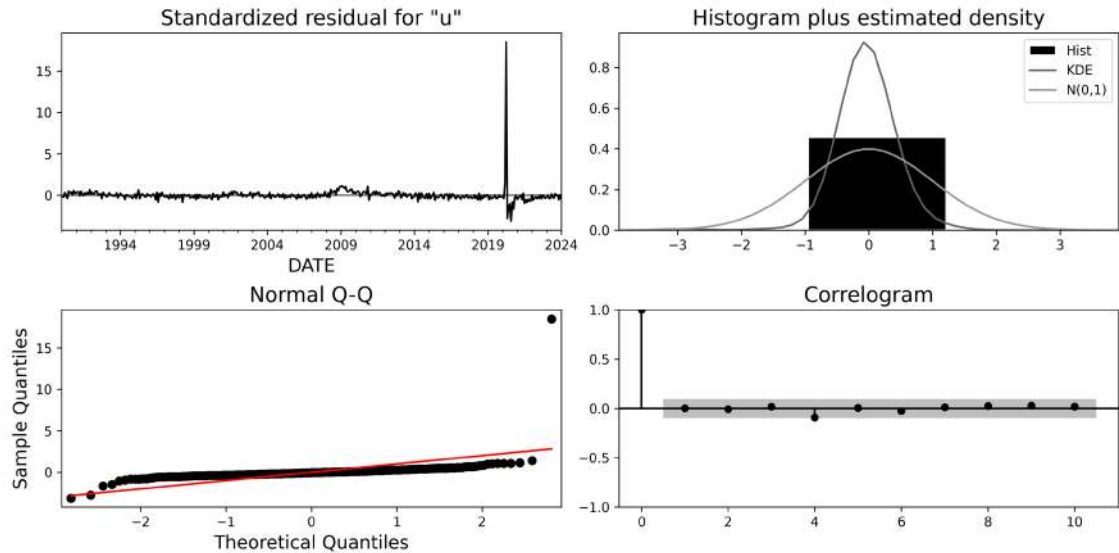


Figure 10.25: Output of AR(3) or ARIMA(3, 0, 0) model diagnostic plots

The standardized residual plot shows a significant spike around 2020, likely due to the economic impact of COVID-19. There are also some deviations from normality based on the Q-Q plots, suggesting the model may not fully capture the tail behavior of the data. Overall, the AR model captured the necessary information based on the autocorrelation plot.

Next, forecast future steps using the AR(3) model and compare it to the VAR(3) model, which we already fitted and applied `inverse_transform` to:

```
# Forecast from AR(3) model
ar_forecast = pd.Series(model.forecast(n), index=test.index)

# Plot the forecast for AR(3) against the actual data
idx = test.index
plt.figure(figsize=(10, 4))
plt.plot(idx, test['unrate'], label='Actual unrate')
plt.plot(idx, ar_forecast, label='Forecasted unrate', linestyle='dashed')
plt.title('AR(3) model - Actual vs Forecasted unrate')
plt.legend()
plt.show()
```


This should produce the following plot:

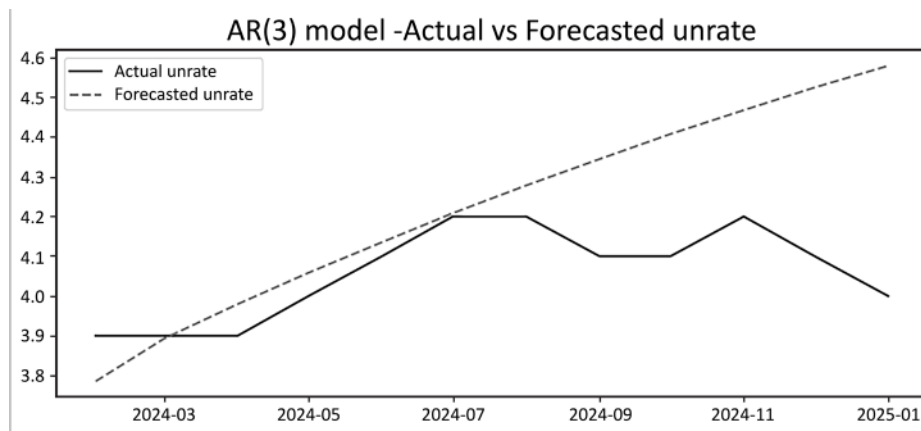


Figure 10.26: AR(3) forecast versus actual comparison against test

The same plot can be done for the VAR(3) model, which includes the impact of FEDFUNDS on unrte:

```
# plotting VAR(3) same code as before but without confidence intervals
idx = test.index
plt.figure(figsize=(10, 4))
plt.plot(idx, test['unrate'], label='Actual unrte')
plt.plot(idx, forecast_df['unrate'],
         label='Forecasted unrte', linestyle='dashed')
plt.title('VAR(3) model - Actual vs Forecasted unrte')
plt.legend()
plt.show()
```

This should produce the following plot:

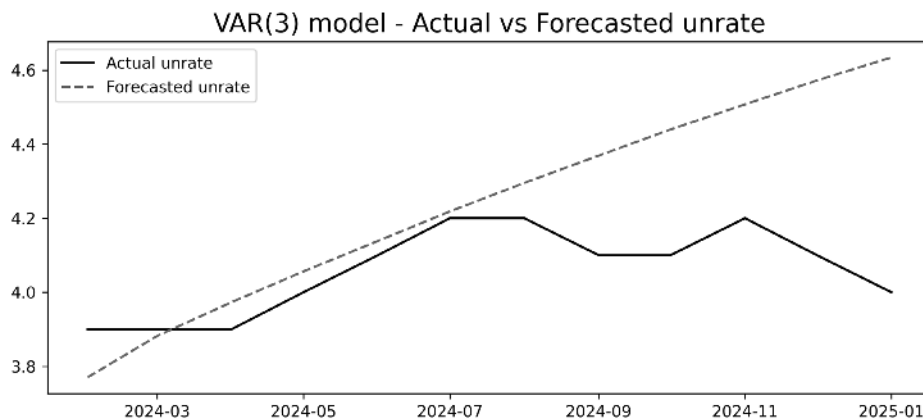


Figure 10.27: VAR(3) forecast versus actual comparison against test

Finally, let's calculate the RMSE scores for both models (VAR and AR) to see which one performs better on the test data:

```
from statsmodels.tools.eval_measures import rmse
rmse_var = rmse(test['FEDFUNDS'], forecast_df['unrate'])
print('VAR(3) RMSE = ', rmse_var)

rmse_ar = rmse(test['FEDFUNDS'], ar_forecast)
print('AR(3) RMSE = ', rmse_ar)
>>
VAR(3) RMSE = 1.0234912944493626
AR(3) RMSE = 1.0235946950213852
```

The AR(3) model has a similar RMSE compared to the VAR(3) model, indicating that the simpler AR(3) model, which only considers the past values of `unrate`, actually performs just as good as the more complex VAR model. The VAR(3) model, which incorporates both `unrate` and `FEDFUNDS`, doesn't significantly improve the forecast for `unrate`.

Given the results, an AR(3) model might be preferred due to its lower complexity and comparable forecasting accuracy. However, when there are stronger interactions between the variables, a VAR model can provide additional insights and more accurate forecasts.

See also

- To learn more about the VAR class in statsmodels, please visit the official documentation at https://www.statsmodels.org/dev/generated/statsmodels.tsa.vector_ar.var_model.VAR.html.

Evaluating VAR models

After fitting a VAR model, the next step is to evaluate how well the model captures the interactions and dynamic relationships between the different endogenous variables (multiple time series). These evaluations provide insights into causality, variable influence, and how shocks to one variable propagate through the system.

In this recipe, you will continue where you left off from the previous recipe, *Forecasting multivariate time series data using VAR*, by exploring various diagnostic tools to deepen our understanding of the VAR model's performance. Specifically, you will do the following:

- Test for **Granger causality** to determine whether changes in one variable help predict changes in another
- Analyze the residual ACF plots to check whether residuals are uncorrelated (white noise)
- Use the **Impulse Response Function (IRF)** to measure how shocks to one variable affect others over time
- Perform **FEVD** to quantify each variable's contribution to forecast error variance (relative importance to each other)

These evaluation techniques will help you understand both the causal relationships and the **interdependencies** between the variables in your system, ensuring that your model effectively captures the underlying dynamics correctly.

Getting ready

The following steps will focus on diagnosing the VAR(3) model that was created from the previous recipe. If you have not performed those steps, you can run the code from the accompanying Jupyter notebook to follow along.

How to do it...

You will start with Granger causality tests to assess whether past values of one time series (variable) can be used to improve the predictions of another beyond its own past values. In this case, you want to find out whether past values of FEDFUNDS provide significant predictive power for unrte. Based on the previous recipe, you have already selected a lag order of 3 using the BIC criterion. However, you can adjust the lag order (e.g., test for higher lags) depending on the nature of your data or specific research question:

1. **Granger causality tests** are implemented in statsmodels with the `grangercausalitytests()` function, which performs four **statistical** tests across each past lag up to a maximum lag threshold. You can control this threshold using the `maxlag` parameter. Granger causality tests are used to determine whether past values from one variable influence the other variable.

The null hypothesis in the Granger causality tests states that the second variable (in this case, FEDFUNDS) does not Granger-cause the first variable (in this case, unrte). In other words, it assumes that past values of FEDFUNDS do not provide statistically significant predictive power (so significant influence or effect).



Note that Granger causality does not imply true causality; it only indicates predictive relationships between time series based on historical data.

2. To test whether FEDFUNDS influences unrte, you will need to switch the order of the columns before applying the test. The order matters because the first variable represents the one being influenced (or *caused*), while the second variable is the potential influencer:

```
granger = grangercausalitytests(
    x=economic_df[['unrate', 'FEDFUNDS']],
    maxlag=3
)
```

The test is set for a maximum of 3 lags for consistency with our VAR(3) model, where the lag order was determined using the BIC criterion earlier. For each lag, the output shows the following:

- **Test statistics:** F-values for F-tests or chi-square for chi-square tests

- **p values:** Used to assess statistical significance
- **Degrees of freedom:** Includes *de_denom* and *de_num* for F-tests, and *df* for chi-square tests

Focus on the p value to decide whether to reject or accept the null hypothesis:

```
Granger Causality
number of lags (no zero) 1
ssr based F test:      F=0.5935 , p=0.4415 , df_denom=417, df_num=1
ssr based chi2 test:   chi2=0.5977 , p=0.4394 , df=1
likelihood ratio test: chi2=0.5973 , p=0.4396 , df=1
parameter F test:      F=0.5935 , p=0.4415 , df_denom=417, df_num=1

Granger Causality
number of lags (no zero) 2
ssr based F test:      F=21.5087 , p=0.0000 , df_denom=414, df_num=2
ssr based chi2 test:   chi2=43.5370 , p=0.0000 , df=2
likelihood ratio test: chi2=41.4205 , p=0.0000 , df=2
parameter F test:      F=21.5087 , p=0.0000 , df_denom=414, df_num=2

Granger Causality
number of lags (no zero) 3
ssr based F test:      F=21.7341 , p=0.0000 , df_denom=411, df_num=3
ssr based chi2 test:   chi2=66.3128 , p=0.0000 , df=3
likelihood ratio test: chi2=61.5503 , p=0.0000 , df=3
parameter F test:      F=21.7341 , p=0.0000 , df_denom=411, df_num=3
```

The `grangercausalitytests` function performs four tests for each lag:

- **SSR-based F test:** Uses the sum of squared residuals to compute the F-statistic
- **SSR-based chi2 test:** Similar to the F test, but uses the chi-square distribution
- **Likelihood ratio test:** Compares the likelihood of two models to assess whether adding lagged terms improves the model
- **Parameter F test:** Evaluates the significance of individual parameters in the model

Since the p values for lags 2 and 3 are less than 0.05, this suggests that FEDFUNDS does **Granger-cause** `unrate` at these lags. In other words, past values of FEDFUNDS provide significant predictive power for `unrate` when considering a lag of 2 or 3 months.

3. Next, you will explore the residual plots to evaluate how well the VAR(3) model has captured the dynamic relationship between variables (FEDFUND and `unrate`). Recall from the previous recipe that the `results` object was created using `results = model.fit(ic='bic')`. The object is of the `VARResultsWrapper` type, which provides access to various methods and attributes. Start with the ACF plot of the residuals using the `plot_acorr` method:

```
results.plot_acorr(resid=True);
```

This should produce four plots (2x2 grid – two plots for each variable) showing both **autocorrelation** and **cross-correlation** between the residuals. Recall from *Figure 10.19* that for two variables, you will have two functions, and this translates into a 2x2 grid. If you had three variables, you would have a 3x3 grid instead of a 2x2 grid:

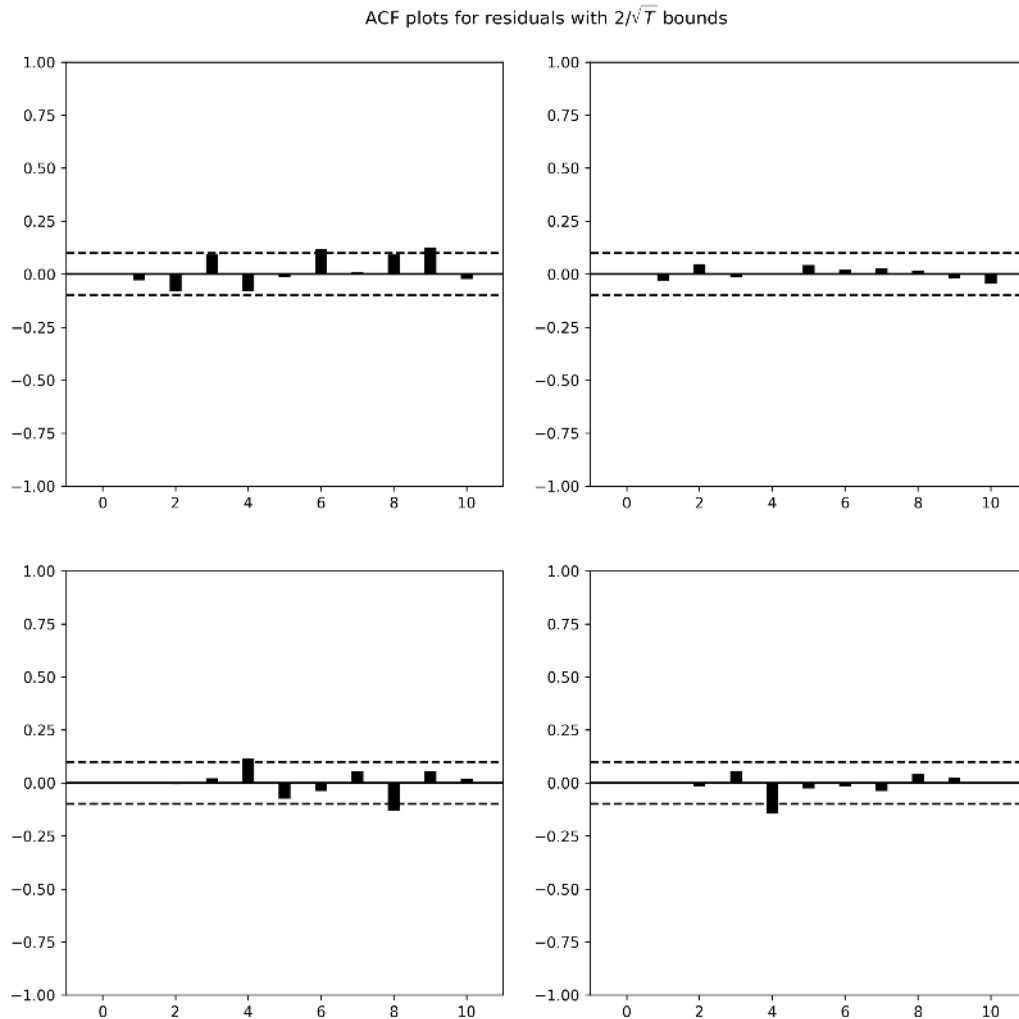


Figure 10.28: Residual autocorrelation and cross-correlation plots

Unfortunately, the plots in *Figure 10.28* do not have proper labels. The plots are arranged in a 2x2 grid as follows:

Diagonal plots represent autocorrelations:

- **Top-left** plot: Autocorrelation of residuals for FEDUNDS
- **Bottom-right** plot: Autocorrelation of residuals for unrate

Off-diagonal plots represent cross-correlation between the residuals of the two variables:

- **Bottom-left** plot: Cross-correlation between residuals of `unrate` (influencer) and `FEDFUNDS` (influenced)
- **Top-right** plot: Cross-correlation between residuals of `FEDFUNDS` (influencer) and `unrate` (influenced)

These plots help evaluate whether the VAR(3) model has captured the relationships between `FEDFUNDS` and `unrate` effectively. The dashed horizontal lines represent the confidence bounds. When the bars stay within these bounds, it indicates no significant correlation at that lag. In *Figure 10.28*, we observe no significant autocorrelation in either variable's residuals and no significant cross-correlation between the variables' residuals. This confirms that VAR(3) has effectively captured the dynamic relationships between the variables (`FEDFUNDS` and `unrate`), with no systematic patterns in the residuals that the model failed to capture and account for.

4. If you want a more detailed look at the residuals for each variable separately, you can do so by extracting the residuals using the `resid` attribute. This would return a DataFrame of the residuals for each variable. You can use the standard `plot_acf` function to create your ACF plots:

```
for col in results.resid.columns:
    fig, ax = plt.subplots(1,1, figsize=(10,2))
    plot_acf(results.resid[col], zero=False,
             lags=10, ax=ax, title=f'ACF - {col}')
```

This should produce two ACF plots – one for `FEDFUNDS` and another for `unrate`. This will confirm the same finding – that there is no significant autocorrelation in the residuals.

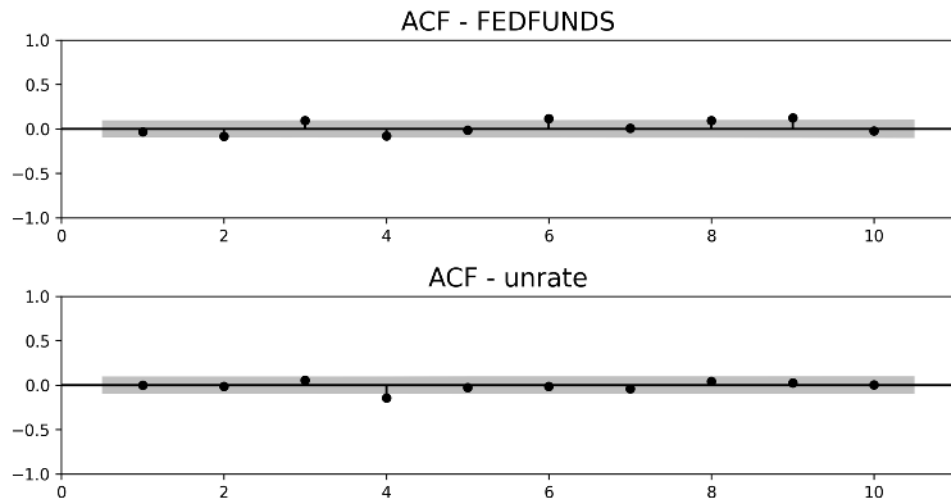


Figure 10.29: ACF plots for `FEDFUNDS` and `unrate`

5. Next, you will analyze the **IRF** to visualize how shocks to one variable propagate through the system and affect other variables (or itself) over time. This analysis is crucial for assessing the dynamic interactions between the variables in a VAR model.

6. You will analyze the impulse response to shocks in the system using the `irf` method:

```
irf_output = results.irf()
```

The `irf_output` object is of the `IRAnalysis` type and has access to a `plot()` function:

```
irf_output.plot();
```

The `irf_output.plot()` function generates four subplots (2x2) showing the impulse responses. Each plot shows the effects of one-unit shock in the impulse variables on the response variables over a specified number of lags. The default number of lags is 10 (`results.irf(periods=10)`):

- **Top-left plot:** Response of FEDFUNDS to a shock in FEDFUNDS
- **Top-right plot:** Response of FEDFUNDS to a shock in unrate
- **Bottom-left plot:** Response to unrate to a shock in FEDFUNDS
- **Bottom-right plot:** Response of unrate to a shock in unrate

The dashed lines represent the 95% confidence interval:

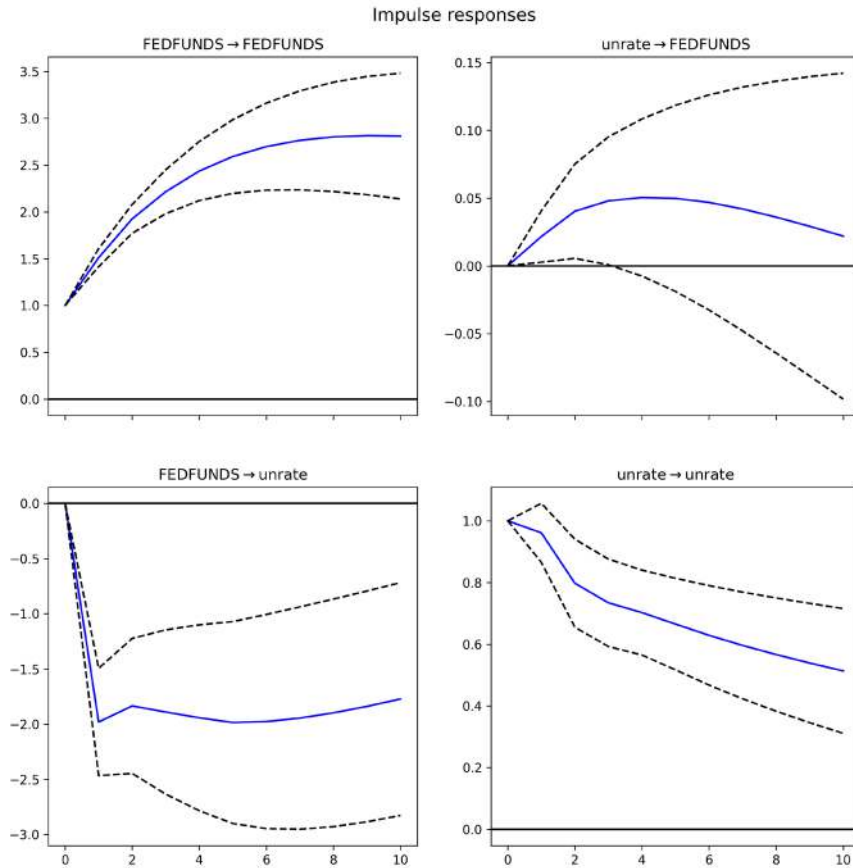


Figure 10.30: Impulse response showing the effect of a one-unit change in one variable on another

The IRF computes the dynamic impulse responses along with approximated standard errors, showing how one-unit shock (or impulse) in one variable propagates through the system and influences other variables over time (lags). In a **VAR model**, all the variables influence each other, and the IRF traces the effect of a change in one variable and its influence on the other variables over time.

For example, the FEDFUNDS → unrate plot (bottom left in *Figure 10.30*) shows how unrate responds to a one-unit increase in FEDFUNDS across the 10 lags. Immediately, there is a noticeable sharp negative response in unrate, indicating that an increase in the federal funds rate lowers the unemployment rate. The effect persists but gradually diminishes over time, which is consistent with how monetary policy impacts employment in the economy. We see this effect from *lag 2* to *lag 3*, where there is a period of stabilization. The response levels off a bit; this is where the delayed effect becomes visible – as the initial drop occurs quickly, the total adjustment takes longer as unrate remains below the starting point. After *lag 3*, we see a slight upward movement in unrate, and this gradual adjustment suggests that unrate starts to recover slowly but does not yet fully return to its pre-shock level for several periods.

On the other hand, the unrate → FEDFUNDS plot (top right in *Figure 10.30*) shows a **relatively small and slightly positive response**. This suggests that an increase in unemployment (unrate) leads to a slight rise in FEDFUNDS, but the effect diminishes over time.

7. If you only want to see the response of one variable to another (instead of all subplots), you can specify the impulse and response variables:

```
fig = irf_output.plot(impulse='FEDFUNDS',
                     response='unrate',
                     figsize=(5, 5))

fig.tight_layout();
```

8. Analyze the **cumulative impulse responses** to understand the total accumulated effect of shocks over time. This is important for understanding the long-term effect of shocks and how their effect builds over time. Use the `plot_cum_effects` function to plot the cumulative response effect:

```
irf.plot_cum_effects();
```

The function generates four subplots (2x2) showing the cumulative impulse responses:

- **Top-left plot:** Cumulative response of FEDFUNDS to a shock in FEDFUNDS
- **Top-right plot:** Cumulative response of FEDFUNDS to a shock in unrate
- **Bottom-left plot:** Cumulative response of unrate to a shock in FEDFUNDS
- **Bottom-right plot:** Cumulative response of unrate to a shock in unrate

The dashed lines represent the 95% confidence interval:

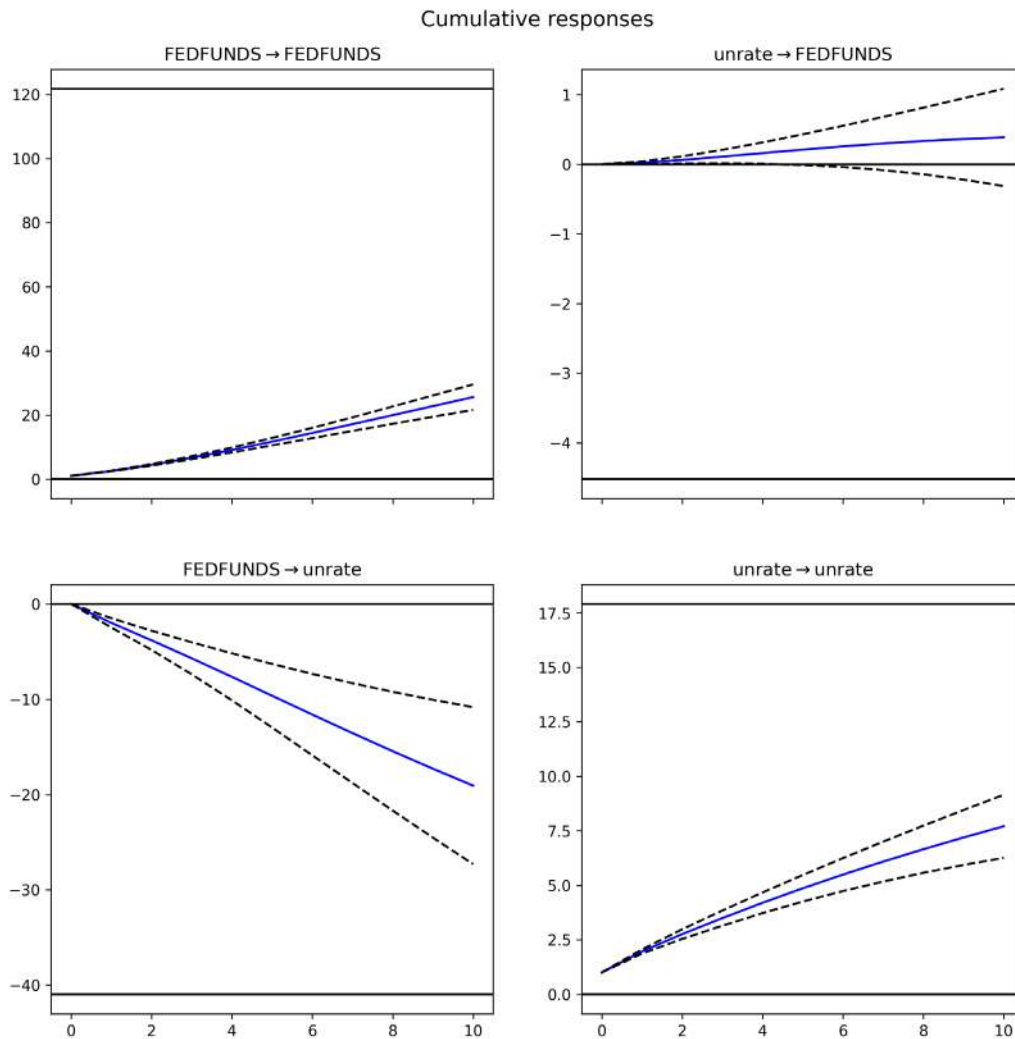


Figure 10.31: Plot of the cumulative impulse response function

The cumulative response plot shows how the effect of a shock builds up over time, which is useful for understanding the sustained effects of policy changes or external shocks. For example, in the FEDFUNDS → unrate plot (bottom left in Figure 10.31), you can see that the cumulative effect of a one-unit increase in FEDFUNDS leads to a sustained decrease in unrate over time.

9. Next, you will compute the **FEVD**, which quantifies how much of the forecast error variance of each variable in the system is attributed to shocks from itself versus shocks from other variables. In other words, you want to understand the contribution of shocks from each variable (key drivers in the system).

You can compute the FEVD using the `fevd()` method:

```
# default is periods=10
fv = results.fevd()
```

10. You can explore the FEVD results using either the `fv.summary()` or `fv.plot()` method, both of which provide similar information:

```
fv.plot();
```

This will produce two FEVD plots (*Figure 10.32*), one for each variable, showing how forecast error variance evolves over time:

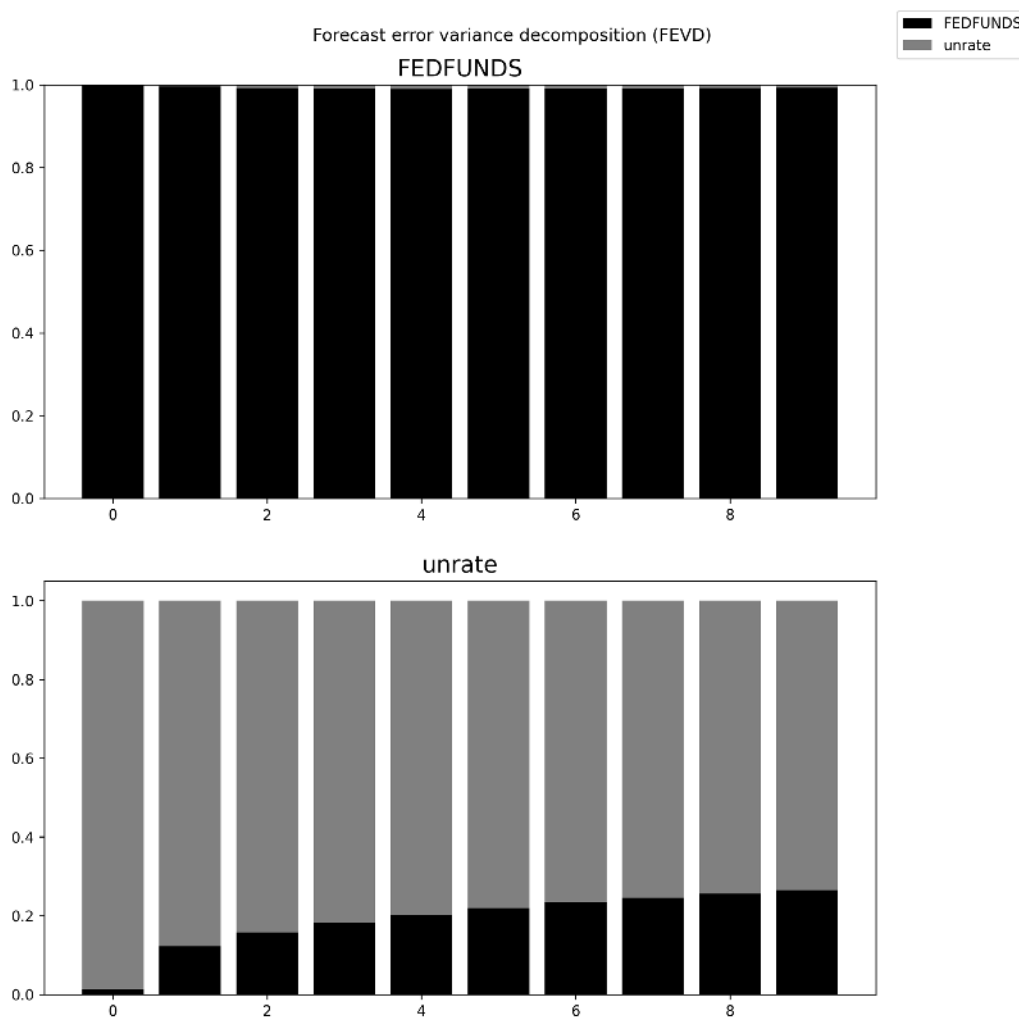


Figure 10.32: FEVD plot for the *FEDFUNDS* and *unrate* variables

The x axis represents the number of periods (lags) from 0 to 9, and the y axis represents the percentage (0 to 100%) contribution of each shock to the forecast error variance. Visually, the contribution of each shock is represented as a portion of the total bar for every period.

The FEDFUNDS plot (top) shows that the forecast error variance for FEDFUNDS is entirely explained by its own shocks across all lags. This means that shocks to `unrate` have no significant impact on the FEDFUNDS forecast error variance.

On the other hand, the `unrate` plot (bottom) starts off by showing that the forecast error variance of `unrate` is mostly explained by its own shocks. However, starting from *lag 4*, the contribution from FEDFUNDS begins to grow. By *lag 9*, roughly around 25% of the `unrate` variance can be attributed to shocks from FEDFUNDS. This suggests that, over longer time horizons, FEDFUNDS becomes a more significant driver of the `unrate` forecast error variance. This demonstrates the delayed but significant impact of monetary policy on employment

FEVD provides valuable insights into how variables influence each other and helps prioritize which variables to monitor or control in dynamic systems.

How it works...

The VAR implementation from `statsmodels` provides a comprehensive set of tools to help you understand the dynamic relationships between different time series (endogenous variables). These tools serve different but complementary purposes:

- **Granger causality tests:** Tests whether past values of one variable help predict another, providing statistical evidence for temporal relationships
- **IRFs:** Trace the effect of a shock in one variable on others over time
- **FEVD:** Quantifies how much of a variable's forecast error variance is explained by shocks from itself or other variables

Each of these tools gives you a different perspective on how variables in the system interact over time, whether through causal links or how shocks to one variable influence others.

In addition to analyzing relationships, diagnostic tools such as residual autocorrelation plots can help assess model fit and determine whether any adjustments (such as model order, differencing, or additional variables) are necessary.

By combining these tools, you can gain a comprehensive understanding of both short-term and long-term interactions between variables while ensuring that your model is well-specified for forecasting or further analysis.

There's more...

In the earlier *Forecasting multivariate time series data using VAR* recipe, you manually created a forecast plot. However, the `results` object (an instance of the `VARResults` class) offers a convenient way to generate and plot your forecasts quickly using the `plot_forecast` method. This function automatically generates a forecast along with confidence intervals for each variable in your system:

```
# Number of future steps to forecast
n = len(test)
# Generate forecast plots with confidence intervals
results.plot_forecast(steps=n, plot_stderr=True);
```

Here, n is the number of future steps. This should produce two subplots – a forecast for each variable in your system – as shown in *Figure 10.33*:

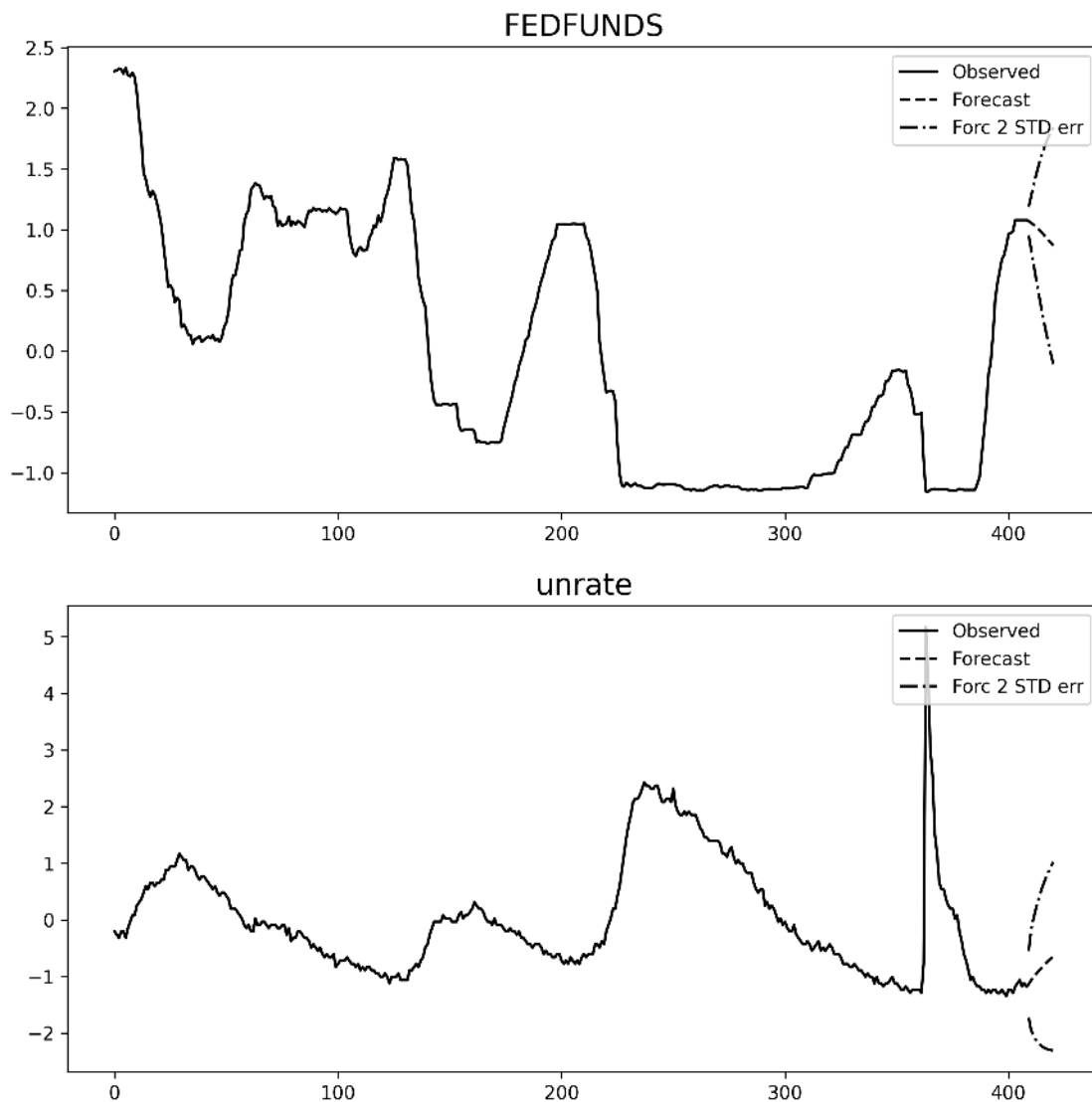


Figure 10.33: Forecast plots for FEDFUNDS and unrate

See also

- To learn more about the `VARResults` class and all the available methods and attributes, visit the official documentation at https://www.statsmodels.org/dev/generated/statsmodels.tsa.vector_ar.var_model.VARResults.html.

Forecasting volatility in financial time series data with GARCH

When working with financial time series data, one of the critical tasks is forecasting **volatility**, which quantifies uncertainty in future returns. Volatility represents the spread of the probability distribution of returns over a specific period, often calculated as the **variance** or its square root, the **standard deviation**. It serves as a proxy for quantifying **risk** or **uncertainty**. In other words, volatility measures the dispersion of financial asset returns around their expected value. Higher volatility indicates greater risk, helping investors gauge the level of return they can expect and the likelihood of deviation from those expectations.

While our previous models, such as ARIMA, SARIMA, Theta, and Prophet, excel at forecasting observed variables based on their past values, they rely on a key assumption: constant variance over time (**homoskedasticity**). However, in financial markets, variance often fluctuates over time (**heteroskedasticity**), making traditional time series models insufficient for capturing these dynamics. This limitation led to the development of specialized models focused on volatility modeling for forecasting and analyzing changes in variance over time.

In this recipe, you will explore a different kind of forecasting: modeling and predicting fluctuations in variance, also known as **volatility forecasting**. Volatility is an important measure of risk, especially in financial markets, where understanding and predicting uncertainty is as important as forecasting future prices. Many trading strategies, such as **mean reversion**, utilize volatility as a key factor to make informed decisions.

To model volatility, you will be introduced to a family of algorithms based on **Autoregressive Conditional Heteroskedasticity (ARCH)**. ARCH models capture time-varying variance by modeling it as a function of past squared residuals or error terms (*innovations*). An extension of ARCH is the **Generalized Autoregressive Conditional Heteroskedasticity (GARCH)** model, which enhances ARCH by adding a moving average component to account for past variances. This extension provides a more flexible and persistent structure for volatility modeling.

GARCH models are widely used in econometrics and financial risk assessment, helping institutions evaluate market risks, investment opportunities, and portfolio stability. They are particularly effective in identifying volatility clustering, a phenomenon where high-volatility and low-volatility periods tend to occur in clusters rather than being randomly distributed.

Before diving into the implementation, let's break down the components of a general ARCH model:

- **Autoregressive (AR):** As we explored in *Chapter 9*, an AR process means that the current value of a variable is influenced by its past values. In ARCH models, this means that current volatility (*variance*) is influenced by past squared residuals or error terms (*innovations*).
- **Heteroskedasticity:** This refers to the property of having different magnitudes or variability at different time points – essentially, the variance changes over time rather than remaining constant.

- **Conditional:** Since the variance (volatility) is not fixed, it depends on past information. In other words, the variance at a given time point is conditionally dependent on past values from the series, meaning volatility is updated dynamically as new information becomes available. This makes the model more responsive to changing market conditions.

To specify a GARCH model, we need to determine its order, which defines how many past terms the model will consider. In this recipe, you will create a GARCH(p, q) model, in which the following applies:

- p represents the number of **lagged** variances (GARCH term)
- q represents the number of **lagged** squared residuals (ARCH term)

You will implement this using the arch Python library and its `arch_model` function. The function's parameters can be broken down into three main components based on the GARCH model's assumptions:

- **dist:** Specifies the assumed distribution of residuals (innovations). The default is 'normal'. Other supported error distributions include 'studentst' or 't', 'skewstudent' or 'skewt', and 'generalized error' or 'ged'.
- **mean:** Controls the model for the conditional mean. The default is 'Constant', which assumes a constant mean. Other acceptable values include 'Zero', 'LS', 'AR', 'ARX', 'HAR', and 'HARX'.
- **vol:** Specifies the volatility model (conditional variance), such as 'GARCH', 'ARCH', 'EGARCH', 'FIGARCH', 'APARCH', and 'HARCH'. The default is 'GARCH'.

These parameters allow you to tailor the model to your specific financial data characteristics. For example, if your returns show heavy tails, you might want to change the 'dist' parameter from the default normal distribution to something like 't' (Student's t-distribution).

A note about model order notation

In most academic literature, the notation for GARCH models uses q to represent the ARCH order (the number of lagged squared residuals or innovations) and p represents the GARCH order (the number of lagged variances). However, the arch Python library reverses this convention.

In the arch package, p refers to the lag order of the squared residuals (the ARCH component) and q refers to the lag order of the lagged variances (the GARCH component).

This distinction is important to keep in mind when specifying and interpreting GARCH models using the arch library, as it differs from the conventional notation found in most textbooks and research papers. This reversal in notation means that when you see GARCH(p,q) in academic papers, you'll need to swap the values of p and q when implementing the model using the arch library. The model's underlying mathematics remains the same – only the parameter naming convention is different. For example, specifying p=2 and q=1 in the `arch_model` function would fit a GARCH(1, 2) model according to conventional notation, meaning 1 lagged variance (GARCH term) and 2 lagged residuals (ARCH term).



Getting ready

You can find the Jupyter notebooks and necessary datasets in this book's GitHub repository. Please refer to the *Technical requirements* section of this chapter for more information.

In this recipe, you will be using the `arch` library, which contains several volatility models, as well as financial econometrics tools. The library produces similar output to that from the `statsmodels` library. At the time of writing, the latest version is 7.2.0.

To install `arch` using `pip`, use the following command:

```
pip install arch
```

How to do it...

For this recipe, you will be using the Microsoft **daily closing price** dataset:

1. Start by loading the necessary libraries for this recipe:

```
from arch import arch_model
```

2. Load the `MSFT.csv` dataset:

```
msft = pd.read_csv('../datasets/Ch10/msft.csv',
                  index_col='date',
                  usecols=['date', 'close'],
                  parse_dates=True)
```

3. You will need to convert the daily stock price into daily **percentage** returns. This can be calculated as $R = \left(\frac{P_t - P_{t-1}}{P_{t-1}} \right) \times 100$, where P_t is the price at time t and P_{t-1} is the price from the previous day. In pandas, this can be done using the `pct_change()` function, followed by multiplying by 100 to express returns as percentages.

The `pct_change()` function takes a `periods` parameter that defaults to 1. If you want to calculate a 30-day return, then you will need to change that value to `pct_change(periods=30)`:

```
msft['returns'] = 100 * msft['close'].pct_change()
msft.dropna(inplace=True, how='any')
```

The `dropna` step is necessary because calculating returns will produce `NaN` for the first row.

4. Plot both the daily closing prices and daily returns to understand the data:

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 7))
msft['close'].plot(ax=ax1, title='MSFT Daily Closing Price')
msft['returns'].plot(ax=ax2, title='Daily Returns (%)')
ax2.axhline(y=0, color='k', linestyle='--', alpha=0.3)
plt.tight_layout()
```

This will produce the following plot:

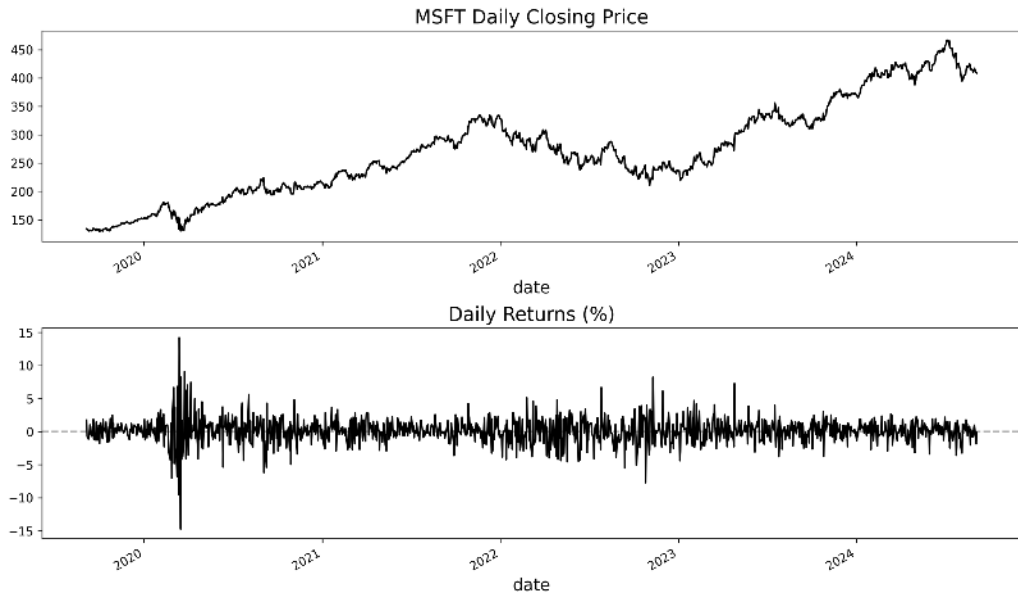


Figure 10.34: Microsoft daily closing price and daily returns

Notice the volatility clusters, such as the spike in early 2020 (likely due to COVID-19 market turbulence) and another significant period around late 2022.

5. Split the data into train and test sets. For short-term volatility forecasting, you will evaluate how well the model predicts the next few days' volatility. When splitting the data, you will leave the last five days for testing (hold out) using the `test_split()` function:

```
train, test = split_data(msft, test_split=5)
>>
train: 1253 , test: 5
```

Before fitting the **GARCH** model, we need to determine the best configuration (parameter values) for mean, dist (distribution), and vol (volatility model). **GARCH(1,1)** is a common starting point due to its simplicity and effectiveness in capturing volatility clusters.

6. In addition to observations made from Figure 10.34, you can validate the distribution of returns by performing statistical tests:

```
import scipy.stats as stats

skewness = stats.skew(msft.returns)
kurtosis = stats.kurtosis(msft.returns)
jarque_bera = stats.jarque_bera(msft.returns)
```



```

print(f"Skewness: {skewness:.4f}")
print(f"Kurtosis: {kurtosis:.4f}")
print(f"Jarque-Bera test statistic: {jarque_bera.statistic:.4f}")
print(f"Jarque-Bera p-value: {jarque_bera.pvalue:.4f}")
>>
Skewness: 0.0033
Kurtosis: 7.2709
Jarque-Bera test statistic: 2771.0549
Jarque-Bera p-value: 0.0000

```

The results indicate that the returns are not normally distributed and have significantly heavy tails (kurtosis > 3). The Jarque-Bera p value strongly rejects the null hypothesis of normality. Based on this, you will use the **Student's t-distribution** (`dist='studentst'`) for the residuals. From the plot in *Figure 10.34*, we can see the returns oscillate around a stable mean and appear roughly symmetrical around zero. Set `mean='Constant'`.

7. Configure and fit the GARCH(1,1) model:

```

model = arch_model(train,
                    p=1, q=1,
                    mean='Constant',
                    vol='GARCH',
                    dist='studentst')
results = model.fit(update_freq=5)
>>
Iteration:      5,   Func. Count:    43,   Neg. LLF: 2391.32450141722
Iteration:     10,   Func. Count:    73,   Neg. LLF: 2391.245393849382
Optimization terminated successfully   (Exit mode 0)
      Current function value: 2391.245393849383
      Iterations: 10
      Function evaluations: 73
      Gradient evaluations: 10

```

The fitting process was completed in 10 iterations. Using `update_freq=5` affects the frequency of progress updates, which can be useful for debugging purposes. The default is 1, which means it will print out an output on each iteration. By setting it to 5, we get an output every 5 iterations.

8. To print the model's summary, use the `summary()` method:

```

=====
Constant Mean - GARCH Model Results
=====
Dep. Variable:          returns    R-squared:          0.000
Mean Model:            Constant Mean  Adj. R-squared:      0.000
Vol Model:            GARCH          Log-Likelihood:    -2391.25
Distribution:          Standardized Student's t    AIC:              4792.49
Method:              Maximum Likelihood    BIC:              4818.16
Date:                Mon, Nov 03 2025    No. Observations:  1253
Time:                23:32:35            Df Residuals:      1252
                                          Df Model:          1
                                          Mean Model
=====
              coef    std err          t      P>|t|    95.0% Conf. Int.
-----
mu           0.1579  3.932e-02     4.017  5.895e-05 [8.088e-02, 0.235]
Volatility Model
=====
              coef    std err          t      P>|t|    95.0% Conf. Int.
-----
omega        0.0797  3.585e-02     2.225  2.611e-02 [9.483e-03, 0.150]
alpha[1]     0.1094  2.440e-02     4.486  7.274e-06 [6.162e-02, 0.157]
beta[1]      0.8702  2.936e-02    29.639  4.680e-193 [ 0.813, 0.928]
Distribution
=====
              coef    std err          t      P>|t|    95.0% Conf. Int.
-----
...
nu           7.4433   1.355         5.493  3.941e-08 [ 4.788, 10.099]
=====

Covariance estimator: robust

```

Figure 10.35: Summary of the GARCH(1,1) model

The *omega*, *alpha*, and *beta* parameters (the ω , α , and β symbols) of the GARCH(1,1) model are estimated using the **maximum likelihood** method. The *p* value for the coefficients indicates they are *statistically significant*.

9. You can access several of the components that you see in the summary table by calling the appropriate attribute from the results object – for example, `results.pvalues`, `results.tvalues`, `results.std_err`, or `results.params`:

```

print(results.params)
>>
mu           0.157941
omega        0.079740
alpha[1]     0.109448
beta[1]      0.870215
nu           7.443287
Name: params, dtype: float64

```

Let's look at the model parameters:

- The mean (μ) of 0.157941 represents the constant average daily return of approximately 0.16%.
- The alpha (0.109448) captures the immediate impact of market shocks on volatility.
- The beta (0.870215) measures how persistent past volatility is in affecting current volatility.

Their sum ($\alpha + \beta = 0.979663$), being very close to 1, indicates high volatility persistence in Microsoft's stock. This means that when periods of high volatility occur (such as during market turbulence), the elevated volatility tends to persist for extended periods before gradually returning to normal levels. In fact, with this level of persistence, it takes about 34 trading days for the impact of a volatility shock to decrease by half (**half_life**):

```
half_life = np.log(0.5) / np.log(0.979663)
print(f"{half_life:.2f} days")
>>
33.74 days
```

10. Evaluate the model's performance by plotting the **standardized residuals** and **conditional volatility** using the plot method:

```
results.plot();
```

This should produce the following plot:

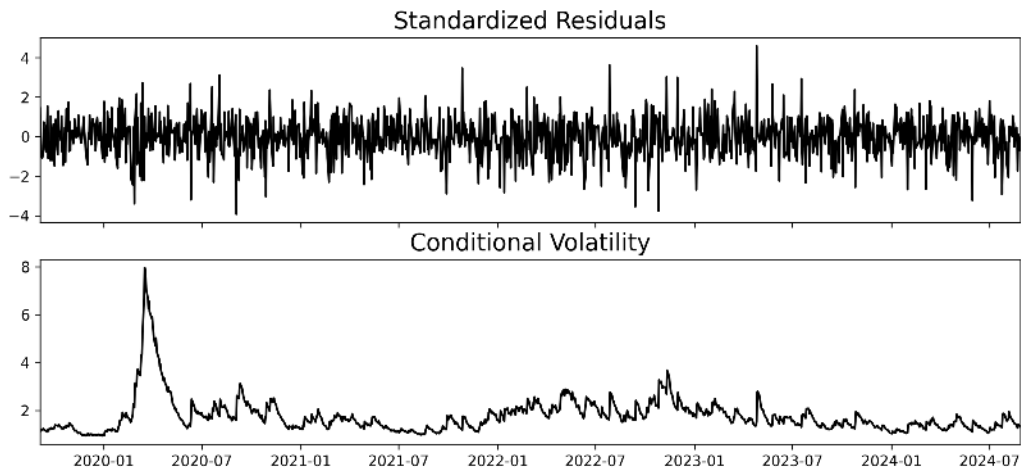


Figure 10.36: Model diagnostics for the GARCH(1,1) model

If the model is well fitted, then the **Standardized Residuals** plot should look like white noise, indicating that no patterns remain in the residuals. The plot seems to show randomness without obvious patterns, indicating a good fit. The **Conditional Volatility** plot shows the time-varying volatility that the model has estimated, reflecting how volatility changes over time. You can see periods of high and low volatility reflecting market conditions.

11. Plot a histogram of the standardized residuals. You can obtain this using the `std_resid` attribute:

```
results.std_resid.hist(bins=20)
plt.title('Standardized Residuals')
```

This will produce the following plot:

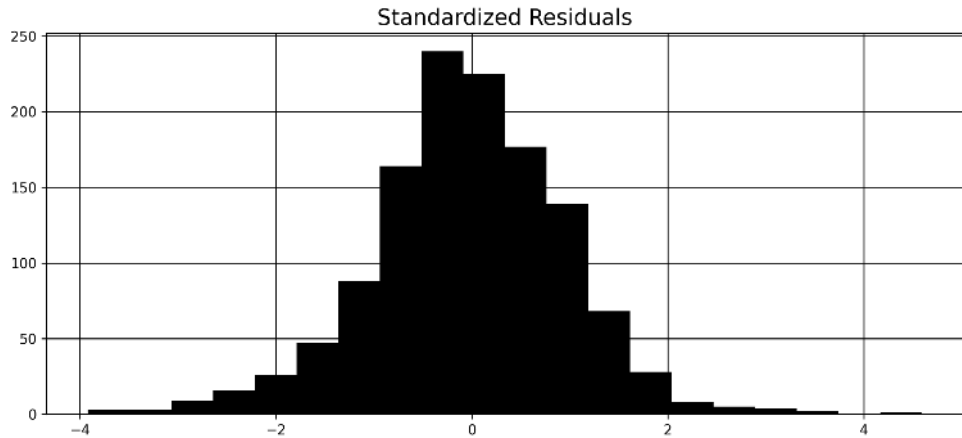


Figure 10.37: Histogram plot for the standardized residuals of the GARCH model

The histogram suggests that the standardized residuals are roughly normal, though you can use additional statistical tests, such as the **Jarque-Bera** test, for a more formal assessment of normality.

12. You can plot the ACF using `plot_acf` on the standardized residuals:

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(results.std_resid);
```

This will produce the following:

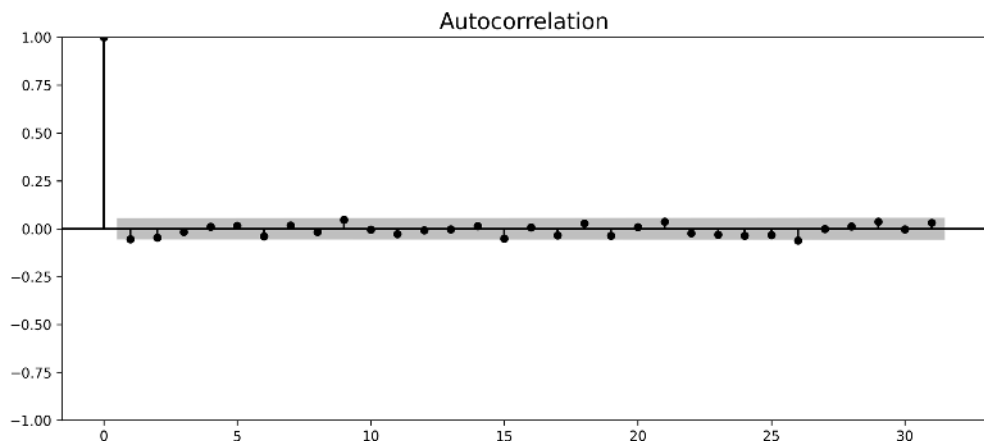


Figure 10.38: Autocorrelation plot of standardized residuals

The ACF plot shows no significant autocorrelation.

13. Additionally, you can test for **autocorrelation** using the **Ljung-Box test**, which tests the null hypothesis of no autocorrelation. Use `acorr_ljungbox` for the first 10 lags. A p value greater than 0.05 indicates that the null hypothesis cannot be rejected, suggesting no significant autocorrelation at that lag:

```
from statsmodels.stats.diagnostic import acorr_ljungbox
acorr_ljungbox(results.std_resid,
               lags=10,
               return_df=True)['lb_pvalue']

>>
1      0.053509
2      0.040470
3      0.080895
4      0.141873
5      0.205075
6      0.161524
7      0.212449
8      0.267047
9      0.168310
10     0.228794
Name: lb_pvalue, dtype: float64
```

The p-values from the **Ljung-Box** test suggest that there is no significant autocorrelation in the standardized residuals at most lags. Since most p values are above 0.05, we fail to reject the null hypothesis of no autocorrelation, indicating a good model fit.

14. To make a prediction, use the `forecast()` method to generate the forecast. The default is to produce a one-step ahead forecast. To get `n` number of steps ahead, you will need to update the `horizon` parameter:

```
msft_forecast = results.forecast(horizon=test.shape[0])
```

15. You access several of the forecast components. For example, to access and display the forecasted variance, you can use the `variance` property from the `msft_forecast` object:

```
forecast = msft_forecast.variance
print(forecast)

>>
               h.1      h.2      h.3      h.4      h.5
date
2024-08-27  1.609903  1.656904  1.702949  1.748058  1.79225
```

The `msft_forecast.variance` gives the predicted conditional variance.

16. To get the **volatility**, which is often more relevant in finance, take the square root of the variance (the standard deviation):

```
print(np.sqrt(forecast))
>>
           h.1      h.2      h.3      h.4      h.5
date
2024-08-27  1.268819  1.287208  1.304971  1.322141  1.338749
```

The forecast shows increasing volatility over the 5-day horizon, starting at approximately 1.27% and rising to 1.34%. This increasing pattern is typical in volatility forecasts as uncertainty tends to grow with the forecast horizon. The mean return remains constant at 0.158% per day (μ), which is expected given our model specification:

```
print(msft_forecast.mean)
>>
           h.1      h.2      h.3      h.4      h.5
date
2024-08-27  0.157941  0.157941  0.157941  0.157941  0.157941
```

17. Lastly, you can plot the volatility forecast:

```
volatility_forecast = np.sqrt(msft_forecast.variance)
forecast_series = pd.Series(volatility_forecast.values.ravel(),
                             index=test.index,
                             name='Forecasted Volatility')

plt.figure(figsize=(10, 6))
forecast_series.plot(marker='o')
plt.title('5-Day Ahead Volatility Forecast')
plt.ylabel('Volatility (Standard Deviation)')
plt.grid(True, alpha=0.3)
```

This should produce the following plot:

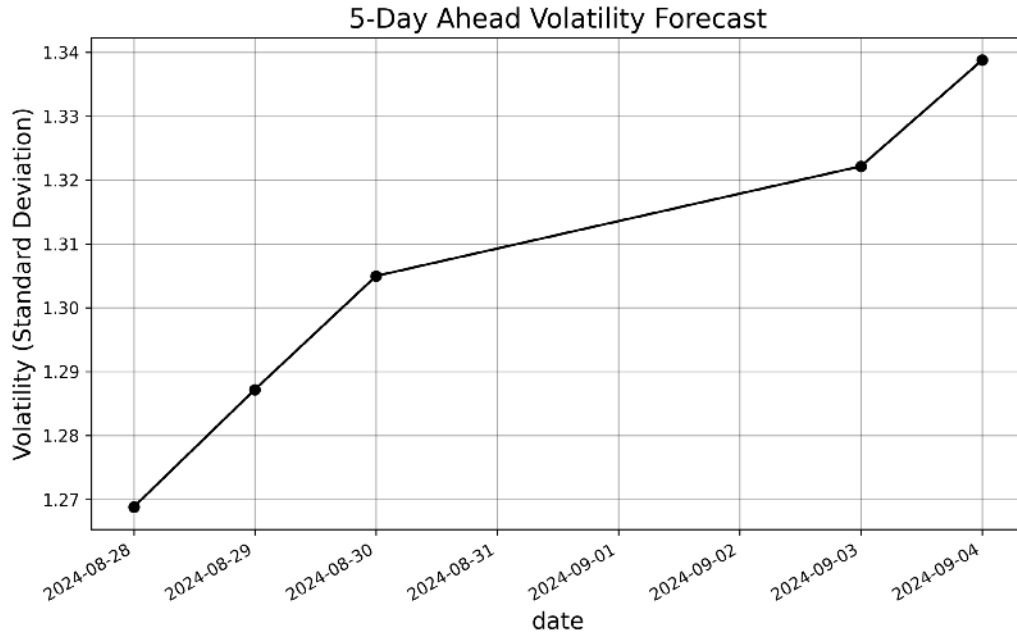


Figure 10.39: Five-day ahead volatility forecast

How it works...

The GARCH(p,q) model can be expressed mathematically as follows:

$$\sigma_t^2 = \omega + \sum_{i=1}^q \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^p \beta_j \sigma_{t-j}^2$$

Let's break this down:

- σ_t^2 represents the conditional variance at time t
- ϵ_{t-1}^2 represents past squared residuals (innovations)
- σ_{t-1}^2 represents past variances

Omega, alpha, and beta (ω , α , and β) are the parameters here. The p order is commonly referred to as the GARCH term (number of lagged variances), and q is referred to as the ARCH order (number of lagged squared residuals).

The parameters for the GARCH model represent the following:

- **Omega (ω):** The constant or baseline variance, representing the long-term average volatility
- **Alpha (α):** The coefficient for the lagged squared residuals (*ARCH term*), which measures the impact of recent shocks on volatility
- **Beta (β):** The coefficient for the lagged variances (*GARCH term*), representing the persistence of volatility over time

The sum of alpha and beta ($\alpha + \beta$) measures volatility persistence. When this sum is close to 1, as in our Microsoft stock example (0.979663), it indicates that high volatility periods tend to cluster together.

The GARCH(1,1) model you implemented can be written as follows:

$$\sigma_t^2 = \omega + \alpha \epsilon_{t-1}^2 + \beta \sigma_{t-1}^2$$

The model captures the impact of recent shocks ($\alpha \epsilon_{t-1}^2$) and the persistence of past volatility ($\beta \sigma_{t-1}^2$).



Innovations vs. errors in time series

In time series literature, **innovations** refer to unexpected and unpredictable new information, shocks, or errors that cannot be forecasted using past information. Put simply, you can think of **innovations** as forecast errors or the surprise at each time step that cannot be explained by the model.

While machine learning typically uses the term *prediction errors*, in time series models such as ARCH/GARCH, we use the term **innovations** to emphasize that these represent new, unanticipated market information that drives changes in volatility.

There's more...

When implementing GARCH models, the choice of mean model and error distribution can significantly impact performance. While we used a 'Constant' mean and Student's t-distribution in our recipe, it's often valuable to evaluate different combinations to find the optimal model for your data.

Here's how you can systematically evaluate different model configurations:

```
from itertools import product
from arch import arch_model

# Define the distributions and mean models to evaluate
# Error distributions
distributions = ['normal', 'studentst', 'skewstudent', 'ged']
# Mean specifications
mean_models = ['Constant', 'Zero', 'AR']

# Use product to create all combinations
for mean, dist in product(mean_models, distributions):
    try:
        model = arch_model(msft['returns'], mean=mean, dist=dist,
                           vol='GARCH', p=1, q=1)
        result = model.fit(disps='off')
        results[(mean, dist)] = {
            'aic': result.aic,
            'bic': result.bic,
```



```

        'llf': result.loglikelihood
    }
except:
    continue

```

In this example, we will sort based on the AIC score to find our best model:

```

# Get best model based on AIC
best_config = min(results.items(), key=lambda x: x[1]['aic'])

print("Best Model Configuration:")
print(f"Mean Model: {best_config[0][0]}")
print(f"Distribution: {best_config[0][1]}")
print(f"AIC: {best_config[1]['aic']:.2f}")
print(f"BIC: {best_config[1]['bic']:.2f}")
>>
Best Model Configuration:
Mean Model: Constant
Distribution: studentst
AIC: 4807.97
BIC: 4833.65

```

The results confirm our initial model choice:

- **Mean Model:** Constant, indicating stable average returns
- **Distribution:** Student's t, appropriate for capturing the heavy tails we observed

This systematic evaluation reinforces that our original model specification in the recipe was indeed the most appropriate for Microsoft's return data.

See also

- To learn more about the arch library, you can visit the official documentation at <https://arch.readthedocs.io/en/latest/univariate/introduction.html>.

Automated univariate forecasting using the StatsForecast library

The StatsForecast library by Nixtla, introduced in early 2022, has gained quite a popularity since its initial release. Nixtla offers several specialized forecasting libraries, including **MLForecast**, which enables machine learning-based forecasting using standard machine learning models; **NeuralForecast**, which implements deep learning approaches for time series prediction; **HierarchicalForecast**, which handles multi-level time series structures with various reconciliation methods; and **TimeGPT**, which provides the proprietary GPT-based forecasting capabilities.

The StatsForecast library offers a collection of popular univariate time series forecasting models, such as ARIMA, Theta, GARCH, ARCH, and exponential smoothing, with automatic variants. According to Nixtla's website, their StatsForecast contains the “*fastest and most accurate* implementations of AutoArima, AutoETS, AutoCES, MSTL, and Theta in Python.”

In this recipe, you will focus on their automatic forecasting tools such as **AutoARIMA**, **AutoETS**, and **AutoTheta**.

Getting ready

You can find the Jupyter notebooks and necessary datasets in this book's GitHub repository. Please refer to the *Technical requirements* section of this chapter for more information. We'll be using StatsForecast version 2.0.2, released in July 2025.

To install StatsForecast using pip, you can run the following:

```
pip install statsforecast
```

How to do it...

You will continue working with the *Milk Production* dataset introduced in the previous recipes, *Forecasting univariate time series data using Prophet* and *Forecasting univariate time series data with Theta*. We use the same dataset for consistency because it contains both *trend and seasonality*, making it a suitable choice for comparing different time series forecasting methods:

1. Start by importing the necessary libraries:

```
from statsforecast import StatsForecast
from statsforecast.models import (AutoARIMA, AutoETS, AutoTheta)
```

2. Load `milk_productions.csv` and split it into *train* and *test* sets:

```
milk_file = Path('../..../datasets/Ch10/milk_production.csv')
milk = pd.read_csv(milk_file, parse_dates=['month'])
milk.columns = ['ds', 'y'] # Rename for Statsforecast compatibility
milk['unique_id'] = 'milk' # Single series, so set to 'milk'
```

Let's break down what we did here. The milk production dataset columns (date and production) were renamed to 'ds' and 'y', respectively. This is because, similar to the Prophet library, StatsForecast expects default column names:

- 'ds' for the time column (the `time_col` parameter)
- 'y' for the target column (the `target_col` parameter)

Additionally, StatsForecast requires a third column, 'unique_id' (the `id_col` parameter), which identifies different time series in a long-format dataset. Even though we only have one time series, we still provide the column by setting 'unique_id' to 'milk' for all rows.

- Next, split the dataset into a train/test split. You will hold out the last 12 months as the test set using the `test_split()` function:

```
test_split = 12
train, test = split_data(milk, test_split)
>>
train: 156 , test: 12
```

- Create a Python list containing the time series models to compare: **AutoARIMA**, **AutoETS**, and **AutoTheta**. These are **automated** versions of popular statistical models that handle trend and seasonality:

```
s1 = 12
models = [AutoARIMA(season_length=s1),
          AutoETS(season_length=s1),
          AutoTheta(season_length=s1)]
```

- Fit multiple models at once using the `StatsForecast` class:

```
sf = StatsForecast(models = models, freq = 'MS')
sf.fit(train)
```



Using the default column names is for convenience, but you can always specify your own column names by updating the `id_col`, `time_col`, and `target_col` parameters when initializing the `StatsForecast` class. For example, use the following if your `DataFrame` has the `id`, `month`, and `production` columns:

```
sf.fit(train,
       id_col = 'id',
       time_col = 'month',
       target_col = 'production')
```

The `StatsForecast` class allows you to fit multiple models on the training data with a single call. You set the season length (`season_length`) to 12 months, since the milk production dataset is monthly and exhibits yearly seasonality. The frequency (`freq = 'MS'`) is set to represent the start of each month for the monthly milk production dataset.

- After fitting the models, you can make your forecast using either the `forecast` or `predict` method. The `forecast` method is designed for efficiency and produces predictions on the fly without the need to store parameters. The `predict` method is useful if you want to inspect your model and store model parameters.
- You will use the `predict` method to inspect model parameters:

```
n = test.shape[0]
forecast_df = sf.predict(h=n, level=[80,95])
```

StatsForecast allows you to request multiple **confidence levels** for **prediction intervals**. In this example, both 80% and 95% confidence intervals (CIs) will be returned for each model.

Inspect the `forecast_df` DataFrame. You will notice that for each model (AutoARIMA, AutoETS, and AutoTheta), we get 5 columns: the predicted value, and the upper and lower bounds for each confidence level (80% and 95%).

8. You can inspect model parameters after fitting. To access the fitted parameters, use the `fitted_` attribute from the `StatsForecast` object. This will give you access to each model's internal configuration:

```
sf.fitted_
>>
array([[AutoARIMA, AutoETS, AutoTheta]], dtype=object)
```

9. You can retrieve the parameters for each specific model by indexing into the `fitted_` array.

Here are some examples:

```
# AutoARIMA model
# AutoARIMA model parameters (p,d,q)(P,D,Q)m
# where p=2, d=0, q=0, P=1, D=0, Q=1, m=12
sf.fitted_[0][0].model_['arma']
>>
(2, 0, 0, 1, 12, 0, 1)
```

This returns the **SARIMA** model parameters indicating a **SARIMA(2,0,0)(1,0,1)12** model, where (2,0,0) represents the non-seasonal part with AR order=2, differencing=0, and MA order=0. The (1, 0,1)12 represents the seasonal part with seasonal AR=1, seasonal differencing=0, seasonal MA=1, and seasonal periods=12 months.

```
# AutoETS model
sf.fitted_[0][1].model_['method']
>>
'ETS(A,A,A)'
```

This returns the **ETS** model components. **ETS(A, A, A)** means the model consists of an **Error (A)** for additive errors, **Trend (A)** for additive trend component, and **Seasonal (A)** for additive seasonal component.

```
# AutoTheta model
sf.fitted_[0][2].model_.get('par')
>>
{'initial_smoothed': 301.76331057730624,
 'alpha': np.float64(0.9899999999999973),
 'theta': np.float64(10.539311991193415)}
```

This returned the initial smoothed value, and **alpha** and **theta** parameters for the AutoTheta model.

10. To view all the available parameters for each model, you can loop through the `fitted_` attribute and print them:

```
model_list = [str(models) for models in sf.models]
for i, m in enumerate(model_list):
    print(f'Model {i} = {m} ')
    print(list(sf.fitted_[0][i].model_.keys()))

>>
Model 0 = AutoARIMA
['coef', 'sigma2', 'var_coef', 'mask', 'loglik', 'aic', 'arma',
 'residuals', 'code', 'n_cond', 'nobs', 'model', 'bic', 'aicc', 'ic',
 'xreg', 'x', 'lambda']
Model 1 = AutoETS
['loglik', 'aic', 'bic', 'aicc', 'mse', 'amse', 'fit', 'residuals',
 'components', 'm', 'nstate', 'fitted', 'states', 'par', 'sigma2', 'n_
params', 'method', 'actual_residuals']

Model 2 = AutoTheta
['mse', 'amse', 'fit', 'residuals', 'm', 'states', 'par', 'n',
 'modeltype', 'mean_y', 'decompose', 'decomposition_type', 'seas_
forecast', 'fitted']
```

11. Inspect the first few predictions from **AutoARIMA** along with the confidence intervals (95%):

```
arima_cols = ['ds', 'AutoARIMA', 'AutoARIMA-lo-95', 'AutoARIMA-hi-95']
print(forecast_df[arima_cols].head())

>>
      ds  AutoARIMA  AutoARIMA-lo-95  AutoARIMA-hi-95
0 1975-01-01  838.410156      823.862915      852.957397
1 1975-02-01  793.659363      775.495972      811.822815
2 1975-03-01  898.408508      877.277954      919.539062
3 1975-04-01  912.460083      888.765808      936.154419
4 1975-05-01  975.550781      949.575928     1001.525635
```

12. Once you have your forecast, you can visualize it by plotting the actual and predicted values:

```
sf.plot(test, forecast_df)
```

This should produce a plot comparing **actual** values from the test set against the predicted values from each model.

13. You can also plot each model's forecast individually, including its confidence intervals. This can be achieved using the `plot` method and updating the `models` and `level` parameters. The `models` parameter takes a list of models, and the `level` parameter also takes a list of confidence intervals. In this example, you can loop through the models and plot each one separately:

```
for model in sf.models:
    print(model)
    fig = sf.plot(test, forecast_df,
                  models=[str(model)], level=[80,95])
    display(fig)
```

This will generate plots for AutoARIMA, AutoETS, and AutoTheta, each with its associated 80% and 95% confidence intervals.

The following is an example output comparing all three models:

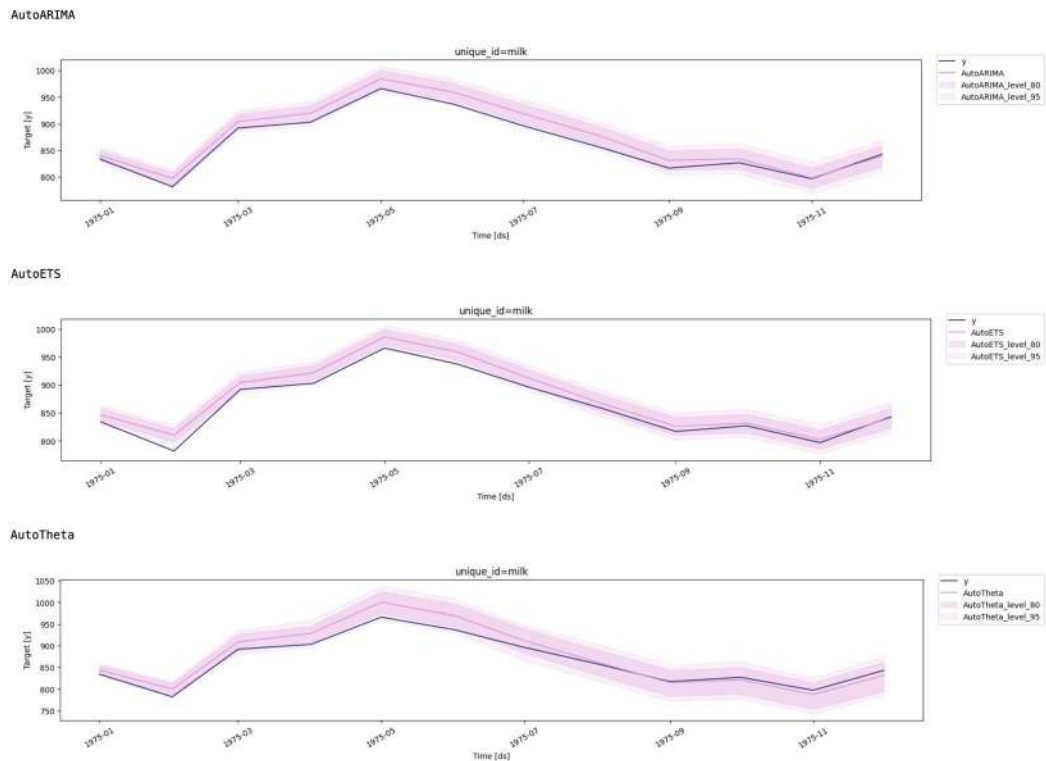


Figure 10.40: Actual versus predicted for AutoARIMA, AutoETA, and AutoTheta with 80% and 95% CIs

14. StatsForecast provides a method for **cross-validation** to evaluate the performance of the models. You can use the `cross_validation()` function to assess how well each model generalized to unseen data, with `h=12` for forecasting 12 steps ahead, `step_size=12` for monthly data, and `n_windows=4` for 4 validation windows:

```
crossvalidation_df = sf.cross_validation(
    df=milk, h=12, step_size=12, n_windows=4
)
crossvalidation_df.head()
```

This returns a DataFrame with forecasts for each model across the validation windows:

	unique_id	ds	cutoff	y	AutoARIMA	AutoETS	AutoTheta
0	milk	1972-01-01	1971-12-01	826.0	835.644897	816.135010	833.166870
1	milk	1972-02-01	1971-12-01	799.0	792.584961	782.166321	789.943298
2	milk	1972-03-01	1971-12-01	890.0	891.021484	870.334045	897.904968
3	milk	1972-04-01	1971-12-01	900.0	909.428711	887.983398	919.605774
4	milk	1972-05-01	1971-12-01	961.0	972.341736	951.904297	993.479309

Figure 10.41: Cross-validation results for each model

The `crossvalidation_df` contains the cross-validation results for each model.

15. The `utilsforecast` library (installed with StatsForecast) provides an `evaluate()` function as well as many commonly used evaluation metrics, such as **MAE**, **MSE**, **RMSE**, and **MAPE**:

```
from utilsforecast.losses import mse, mape, rmse, mae
from utilsforecast.evaluation import evaluate

metrics = [mae, mse, rmse, mape]

evaluation = evaluate(
    crossvalidation_df,
    metrics=metrics,
    models=model_list,
)
print(evaluation.drop(columns='unique_id'))

>>
metric  AutoARIMA  AutoETS  AutoTheta
0  mae    13.109066   15.322071   15.726689
1  mse   247.590637  334.270325  344.069336
2  rmse   15.735013   18.283062   18.549107
3  mape    0.015649    0.018218    0.018136
```

16. Finally, you can pick the best model based on the evaluation metrics by finding the model that achieves the lowest score for each metric:

```
evaluation['Best_Model'] = evaluation[[
    'AutoARIMA', 'AutoETS', 'AutoTheta'
]].idxmin(axis=1)
print(evaluation)

>>
metric  AutoARIMA  AutoETS  AutoTheta  Best_Model
0   mae    15.087765  15.322071  15.726682  AutoARIMA
1   mse   338.023010  334.270325  344.069244  AutoETS
2  rmse    18.385403  18.283062  18.549103  AutoETS
3  mape     0.018029   0.018218   0.018136  AutoARIMA
```

The results show an interesting pattern where different models excel at different metrics:

- **AutoARIMA** performs best in terms of MAE and MAPE
- **AutoETS** leads in MSE and RMSE
- **AutoTheta**, while competitive, doesn't lead in any metric, but remains close to the other models and is a simpler model

This highlights an important aspect of time series forecasting: different metrics may favor different models based on their error characteristics. Here is an example:

- MAE and MAPE (where AutoARIMA leads) are more robust to outliers and give equal weight to all errors
- MSE and RMSE (where AutoETS leads) penalize larger errors more heavily due to the squared term

Do note that when selecting the “best” model, you should consider which metric aligns best with your business objectives:

- If your goal is to minimize the impact of large errors, focus on MSE/RMSE
- If you need more interpretable error measures or want to treat all errors equally, consider MAE/MAPE

The differences between the models are relatively small, suggesting that all three approaches are viable for this dataset.

How it works...

One of the great features in **StatsForecast** is that you can train multiple models on different time series with just a few lines of code. You can also generate **probabilistic forecasts** and request different **confidence interval** levels to produce both **point forecasts** and **prediction intervals**, which provide insights into forecast uncertainty.

The **cross-validation** implementation in StatsForecast is also fast and efficient. In time series analysis, cross-validation involves splitting the data into multiple training and testing sets using a **rolling window**. Each model is trained on earlier data up to a certain cutoff, and then tested on future data. This approach helps evaluate the model's ability to generalize to unseen data better than the traditional one-time train/test split.

In the recipe, you provide the entire milk production dataset to the `cross_validation()` function:

```
sf.cross_validation(df=milk, h=12, step_size=12, n_windows=4)
```

Let's break it down:

- `h=12`: This sets the forecast horizon to 12 months. This means that for each cross-validation window, the model will predict 12 months into the future. Think of this as your test set size.
- `step_size=12`: This controls how far the window moves forward after each iteration. With `step_size=12`, each new training window starts 12 months after the previous one. This ensures that each test period contains completely new, unseen data.
- `n_windows=4`: This creates four different train-test splits. If you are familiar with k-fold cross-validation (k-Fold CV) in traditional machine learning, you can think of `n_windows` as analogous to the number of folds in k-fold CV. Each window evaluates the model on a different portion of the time series, providing a more robust evaluation of model performance. The key difference is that in time series, we maintain the temporal order – newer data is always used for testing, while older data is used for training.

This **rolling-window** approach allows the model to learn from different segments of the time series, ensuring that it captures both trends and seasonal variations from different parts of the data.

The diagram in *Figure 10.42* summarizes the cross-validation approach visually:

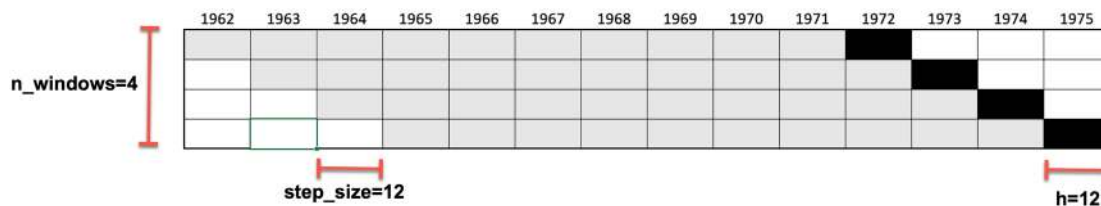


Figure 10.42: Cross-validation with `h=12`, `step_size=12`, and `n_windows=4`

The diagram in *Figure 10.42* shows how the training and testing windows roll forward by 12 months after each iteration, creating 4 distinct evaluation periods, ensuring that the model is evaluated on multiple distinct test sets:

- Window 1: Uses data from 1962–1971 to predict 1972 (`h=12` months)
- Window 2: Shifts forward, using 1963–1972 to predict 1973
- Window 3: Moves to 1964–1973 to predict 1974
- Window 4: Finally uses 1965–1974 to predict 1975

This method allows us to assess the model's ability to predict future values across various time periods, providing a more comprehensive performance evaluation.

There's more...

In the *Forecasting multivariate time series data using VAR* recipe, we discussed **endogenous** versus **exogenous** variables. In short, **endogenous variables** are those that influence and are influenced by other variables within the system (in the current example, the milk production is influenced by its own past values and historical patterns), while **exogenous variables** are external factors (independent factors) that influence the system but are not influenced by it.

In our milk production example, if we were considering multiple dairy-related variables (such as milk production and milk prices), these would be endogenous variables since they likely influence each other. However, the **consumer price index (CPI)** is an exogenous variable because while it may affect milk production through economic conditions, milk production doesn't significantly influence overall CPI levels.

Incorporating exogenous variables exposes the model to additional information, which potentially may improve its ability to capture real-world influences and enhance its forecasting accuracy. For example, adding feed prices as an exogenous variable in the milk production model will allow the model to account for fluctuations in feed costs, which may influence production levels. Similarly, using the CPI could help explain the broader economic conditions that might affect dairy farm operations.

In this section, you will explore how **StatsForecast** makes it easy to incorporate exogenous variables by using the `cpi.csv` dataset (essentially, it is the Milk Production dataset with an additional CPI column).

Start by loading the `cpi.csv` dataset and renaming the columns for convenience. In this case, the target variable (`y`) represents milk production, and the exogenous variable (`X`) is the **CPI**:



Even though the CPI dataset contains only one exogenous variable, your data could include additional exogenous columns. The model is capable of considering multiple exogenous variables at once. For instance, you could include both **feed prices** and **temperature** alongside CPI to give the model a broader set of factors to consider, potentially enhancing forecast accuracy.

```
milk_file = Path('../datasets/Ch10/cpi.csv')
milk = pd.read_csv(milk_file, parse_dates=['month'])
milk.columns = ['ds', 'y', 'X']
milk['unique_id'] = 'milk'
print(milk.head())
>>
```

	ds	y	X	unique_id
0	1962-01-01	589	30.04	milk
1	1962-02-01	561	30.11	milk

```
2 1962-03-01 640 30.17 milk
3 1962-04-01 656 30.21 milk
4 1962-05-01 727 30.24 milk
```

Split the data into a train and a test set, where we hold out the last 12 months as the test set. Additionally, separate the exogenous variable (X) from the target variable (y) for use in forecasting:

```
train, test = split_data(milk, 12)
X_test = test.drop(columns=['y']) # Exogenous variables for testing
Y_test = test.drop(columns=['X']) # Target variable for testing
```

For this activity, you will explore the **AutoARIMA model**. Set `season_length` to 12 months (since we expect annual seasonality), and fit the model on the training data (`df=train`). Forecast 12 months ahead (`h=12`), providing `X_test` as the exogenous variable (`X_df=X_test`):

```
models = [AutoARIMA(season_length = 12)]
sf = StatsForecast(models=models, freq='MS', n_jobs=-1)
# Model with exogenous variables
sf.fit(train)

forecast = sf.forecast(df=train, X_df=X_test, h=12, level=[95])
```

Future values of exogenous variables must be provided for the forecast period (`X_test`).



When using exogenous variables in forecasting, ensure that you have future values or reliable forecasts of these variables for the entire forecast horizon.

Once the forecast is generated, print and plot it to visualize the predicted values and their confidence intervals (95% CI), as shown in the figure that follows:

```
print(forecast.head())
>>
  unique_id      ds  AutoARIMA  AutoARIMA-lo-95  AutoARIMA-hi-95
0      milk 1975-01-01  838.484092      823.918154      853.050030
1      milk 1975-02-01  794.040735      775.835364      812.246106
2      milk 1975-03-01  899.036197      877.806395      920.265999
3      milk 1975-04-01  913.368435      889.494317      937.242552
4      milk 1975-05-01  976.887260      950.633832     1003.140688

# visualize the forecast
sf.plot(test, forecast)
```

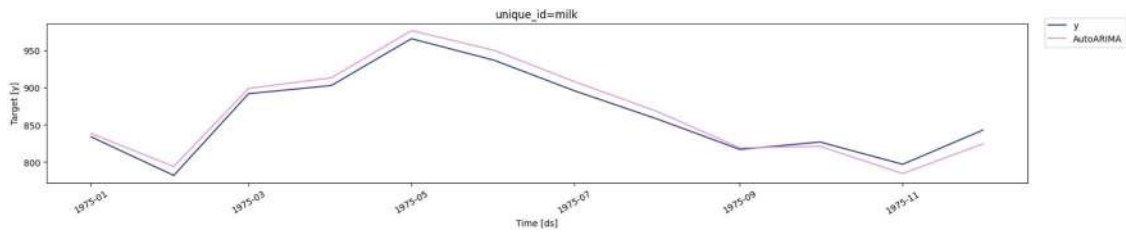


Figure 10.43: Actual versus predicted for AutoARIMA with exogenous variable and 95% CI

Next, evaluate the model's accuracy by calculating the MAE. This compares the actual values (Y_{test}) to the forecasted values from the model (AutoARIMA), including the exogenous variables:

```
res = Y_test.merge(forecast, how='left', on=['unique_id', 'ds'])
mae = abs(res['y']-res['AutoARIMA']).mean()
print('The MAE with exogenous regressors is '+str(round(mae,2)))
>>
The MAE with exogenous regressors is 9.95
```

For comparison, train and forecast a univariate model (without exogenous variables) and compute the MAE again:

```
# model without exogenous variables (univariate)
forecast_u = sf.forecast(df=train[['unique_id', 'ds', 'y']], h=12)
res_u = Y_test.merge(forecast_u, how='left', on=['unique_id', 'ds'])
mae_u = abs(res_u['y']-res_u['AutoARIMA']).mean()
print('The MAE without exogenous regressors is '+str(round(mae_u,2)))
>>
The MAE without exogenous regressors is 13.56

# Improvement percentage
improvement = ((mae_u - mae) / mae_u) * 100
print(f'Improvement with exogenous variables: {improvement:.1f}%')
>>
Improvement with exogenous variables: 26.6%
```

As can be observed, the model with **exogenous** regressors has a lower MAE compared to the **univariate** model. In this case, including CPI as an exogenous variable improved the MAE from 13.56 to 9.95, representing a significant 26.6% improvement in the forecast.

The results could trigger further investigation and experimentation, such as the following:

- Incorporating additional exogenous variables (e.g., feed price and temperature) that might have stronger relationships with milk production
- Refining the model's configuration (e.g., adjusting `season_length` or exploring other models such as **AutoETS** or **AutoTheta**)

It's important to note that exogenous variables may not always lead to improvements in forecast accuracy, and they may not be suitable in every case. Only through experimentation and leveraging domain knowledge can you determine whether specific external factors are good candidates for your model. When using exogenous variables, also consider the following:

- The availability and quality of future values for these variables
- The stability of the relationship between variables over time
- The additional complexity introduced to the model

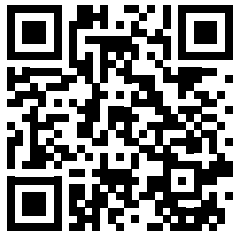
See also

- To learn more about StatsForecast, you can refer to the official documentation here: <https://nixtlaverse.nixtla.io/statsforecast/index.html>.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



11

Forecasting Using Supervised Machine Learning

In this chapter, you will explore how to apply different **Machine Learning (ML)** algorithms for time-series forecasting. ML algorithms can be categorized into **supervised**, **unsupervised**, and **reinforcement** learning. Here, we will focus on *supervised ML*, where the goal is to learn from past observations to predict future values.

The first recipe will guide you through the critical process of preparing a time series for supervised ML. Since time-series data is sequential, we need to transform it to make it suitable for ML models, which typically expect a **tabular format**.

Furthermore, you will work with two powerful ML libraries: `scikit-learn` and `sktime`. `Scikit-learn` (often called `sklearn`) is a popular ML library in Python that offers a wide range of algorithms for supervised and unsupervised learning and a plethora of tools for data preprocessing, as well as model evaluation and selection. Keep in mind that `scikit-learn` is a generic ML library and is not specific to time-series data. On the other hand, the `sktime` library is a specialized library that provides a unified interface for various time-series tasks, including forecasting, classification, and regression, with built-in support for time-series-specific cross-validation and evaluation metrics. The `sktime` library offers a wide range of forecasters, including deep learning models and ensemble methods.

In this chapter, we will cover the following recipes:

- Preparing time-series data for supervised learning
- One-step forecasting with `scikit-learn`
- Multi-step forecasting with `scikit-learn`
- Forecasting with `sktime`
- Hyperparameter tuning with `sktime`
- Forecasting with exogenous variables and ensemble learning in `sktime`
- Forecasting with `XGBoost`

Then, add the correct frequency for each DataFrame:

```
daily_temp.index.freq = 'D' # Daily frequency
energy.index.freq = 'MS' # Month start frequency
air.index.freq = 'M' # Month end frequency
```

You can plot the three DataFrames to gain an understanding of how they differ:

```
plt.rcParams["figure.figsize"] = [18, 3]
fig, (ax1, ax2, ax3) = plt.subplots(1, 3)

daily_temp.plot(ax=ax1, title='Avg Daily Weather Temperature in C')
energy.plot(ax=ax2, title='Monthly Energy Consumption')
air.plot(ax=ax3, title='Monthly Passengers')

plt.tight_layout()
plt.show()
```

This generates three plots comparing the temporal patterns across the datasets:

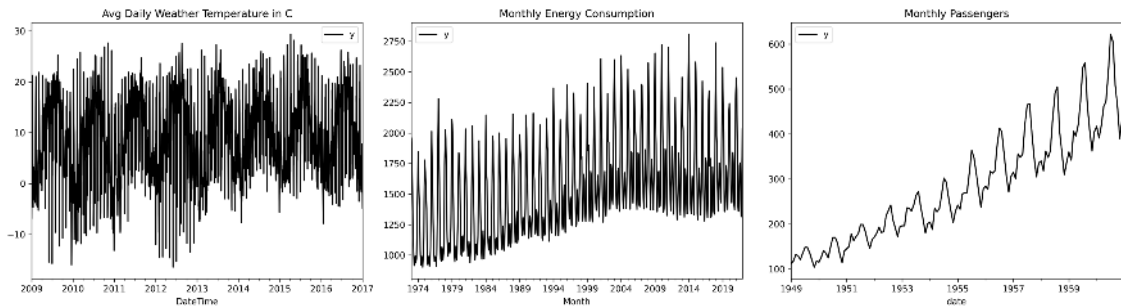


Figure 11.1: Visualizing the three datasets – daily temperature, energy consumption, and air passengers

When plotting the datasets, observe how each time series exhibits different characteristics. This initial insight will be helpful as you proceed with the chapter's recipes. In addition, you will realize how an algorithm's performance will vary when applied to different time-series data.

You can download the Jupyter notebooks and requisite datasets from the GitHub repository:

- **Jupyter notebook:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch11/Chapter_11.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch11>

Understanding supervised ML

In **supervised ML**, the data you use for training contains known outcomes from the past, called the **dependent variables** or **targets**. These are the variables you want your ML model to predict. The ML algorithm learns patterns from this data using the other variables, known as **independent variables** or **predictors**. These predictors help the model estimate the target.

For example, in the house pricing prediction problem, the target variable would be the price of the house. The predictors may include features such as the number of bedrooms and bathrooms, the total square footage, and the house's location. The ML model learns from this data to make predictions for house prices based on new and unseen data.

On the other hand, in **unsupervised ML**, there are no target variables or outcomes to train on. Unsupervised algorithms are good at finding patterns or structures in the data. Common examples include clustering (e.g., customer segmentation) or anomaly detection, such as flagging unusual transactions (see *Chapter 13*).

Generally, supervised ML has two main categories, that is, **classification** and **regression**:

- **Classification:** The goal is to predict which class (or category) a particular observation belongs to. This means predicting a **discrete value**, such as determining whether an email is spam or not, or whether a transaction is fraudulent or not. These are examples of **binary classification** problems (two possible classes). **Multi-class classification** involves predicting more than two classes, such as classifying images into categories such as cats, dogs, or birds. Some popular classification algorithms include **logistic regression**, **Random Forest**, **K-Nearest Neighbors (KNN)**, and **Support Vector Machine (SVM)**, to name a few. In time series, classification tasks might include detecting patterns such as trend reversals or identifying specific market conditions based on historical data patterns.
- **Regression:** The goal is to predict a continuous variable, such as predicting the price of a house or a person's height. Regression comes in several flavors:
 - **Simple regression:** You work with just one predictor (feature) to predict one target variable
 - **Multiple regression:** You use multiple predictors (features) to predict one target variable
 - **Multivariate multiple regression:** You leverage multiple predictors (features) to simultaneously predict multiple target variables

Some popular regression algorithms include **linear regression**, **Lasso regression**, and **elastic net regression**. These are considered **linear algorithms**, which assume a linear relationship between the predictors and the target. However, several classification algorithms can be adapted for regression tasks. For example, **Random Forest**, **KNN**, and **SVM** can be used to build regressor models that capture **non-linear relationships** between variables, producing more flexible and complex models.

In this chapter, regression is the primary focus since forecasting typically involves predicting continuous values.

Preparing time-series data for supervised learning

In supervised ML, it is essential to specify your **independent variables** (predictor variables) and the **dependent variables** (target variable). In scikit-learn, for example, you use the `fit(X,y)` method for fitting a model, where `X` is the matrix of independent variables and `y` is the vector of the target values.

In the context of time-series forecasting, the independent variables (`X`) are typically lagged values of the time series, and the dependent variable (`y`) is the current or future value you want to predict.

Generally, preparing time-series data for supervised learning shares similarities with what you have done in previous chapters, but there are specific steps that are unique to supervised learning. This recipe will walk you through these steps and teach you how to transform your time series manually. In later chapters, you will be introduced to other libraries that handle the transformations for you.

The following highlights the overall steps:

1. Inspect your time-series data for any missing data. Your time-series data may contain gaps that can affect model performance. If there are gaps, evaluate the impact and consider some of the **imputation** and **interpolation** techniques discussed in *Chapter 6*.
2. Check for stationarity assumptions. Before fitting the model, you need to be aware of any stationarity assumptions in the algorithm. Some models may require the time series to be stationary before fitting. If the model you are using has this requirement, consider applying transformations (e.g., **differencing**) such as those discussed in the *Detecting time-series stationarity* recipe in *Chapter 8*.
3. Transform your time series **into a tabular format** with independent and dependent variables. This is typically done using a sliding window technique, where you create a window of past observations (lags) to predict the current or future value.

For example, if you decide that the past five periods (lags) should be used to predict the current period (the sixth), then you will create a sliding window of five periods. This will transform a **univariate** time series into a **tabular format**. While a univariate time series initially has only one dependent variable and no independent variables, applying a five-period sliding window will transform it by producing five independent variables, which are lagged versions of the dependent variable. The final shape of your dataset will now contain a total of six columns—five independent variables (features) and one target variable.

This representation of multiple inputs (a sequence) to one output is referred to as a **one-step forecast**. This sets us up for one-step forecasting in this recipe, and we'll tackle multi-step forecasting in later recipes. The following two figures illustrate this concept. *Figure 11.2* shows the mechanical process of creating windows and *Figure 11.3* shows how this translates to an ML-ready dataset.

Note that the window size selection impacts both the number of features and the amount of training data available—larger windows create more features but reduce the number of training samples.

In addition to lagged values, other features (e.g., rolling statistics, seasonality indicators) can also be included as independent variables.

4. Split the data into training and test sets. When building your model, it is crucial to avoid overfitting by holding out a portion of the data for testing. For time-series data, when splitting the data into training and testing sets, we need to preserve the temporal order (e.g., use the first 80% of the data for training and the remaining 20% for testing). You can further split the data into *training*, *validation*, and *test* sets, as you will explore in *Chapter 12*.
5. Scale the data. Many ML algorithms benefit from data scaling, especially when features have different units or magnitudes. You can use scikit-learn's `StandardScaler` or `MinMaxScaler` to normalize your data. After making predictions, you will need to apply the inverse **transform** to restore the results to their original scale; for example, you can use the `inverse_transform` method from scikit-learn.

How to do it...

In this recipe, you will prepare a **univariate** time-series dataset for supervised learning using a sliding window to create independent variables from past observations. The following diagram (*Figure 11.2*) illustrates how a sliding window of five periods creates the dependent (target) variable at a specific time (t) and the five independent variables (x_1, x_2, x_3, x_4, x_5). For example, with the daily temperature data, this means a five-day sliding window.

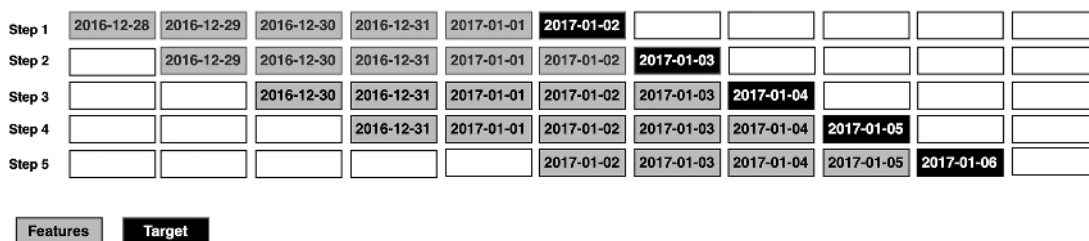


Figure 11.2: Example of a five-day sliding window for daily temperature data

Since you will be transforming all three DataFrames, you will create functions to simplify the process:

1. Start by copying the original DataFrames (*air passengers*, *energy consumption*, and *daily temperature*) and inspecting them for missing values. If missing data is found, then perform a simple forward fill:

```
air_copy = air.copy()
energy_copy = energy.copy()
daily_temp_copy = daily_temp.copy()
```

2. Load and Pass each DataFrame to the `handle_missing_data` function:

```
from utils import handle_missing_data
for df in [air_copy, energy_copy, daily_temp_copy]:
```

```

        handle_missing_data(df)
>>
    found 2 missing observations...

```

Only the `daily_temp_copy` DataFrame had two NaN (missing) values.

3. Load the `generate_lagged_features` function, which transforms the univariate time series into a new format where each row contains a set of lagged values (features) and the next time step as the target variable. The function reshapes the time series from its original long format into a wide, tabular format, where the number of columns is determined by the **sliding window** size (for features), plus one additional column for the target.

As shown in *Figure 11.3*, the original long-format time series is transformed into a format where each row contains multiple input columns (`x_1` to `x_n`) representing past observations, with a single target column (`y`) representing the next time step. This transformation is essential for preparing time-series data for supervised ML models. This technique is described in *Machine Learning Strategies for Time Series Forecasting*, Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg (https://doi.org/10.1007/978-3-642-36318-4_3).

You can load the helper function from `utils.py` as shown:

```

from utils import generate_lagged_features

```

The `generate_lagged_features` function transforms a time series based on a specified number of steps (the **sliding window size**).

4. For simplicity, you can apply this transformation to all three DataFrames using a sliding window size of five periods (`window=5`). For the weather dataset, each period represents one day, while for the air passengers and energy consumption datasets, one period corresponds to one month:

```

air_os = generate_lagged_features(air_copy, 5)
energy_os = generate_lagged_features(energy_copy, 5)
daily_temp_os = generate_lagged_features(daily_temp_copy, 5)

```

5. Inspect one of the datasets before and after transformation:

```

#before transformation
print(air_copy.head(6))
>>

```

	y
date	
1949-01-31	112
1949-02-28	118
1949-03-31	132
1949-04-30	129
1949-05-31	121

```
1949-06-30 135

# after transformation
print(air_os.head())
>>
           x_1  x_2  x_3  x_4  x_5    y
date
1949-06-30  112  118  132  129  121  135
1949-07-31  118  132  129  121  135  148
1949-08-31  132  129  121  135  148  148
1949-09-30  129  121  135  148  148  136
1949-10-31  121  135  148  148  136  119
```

6. You will need to split the transformed datasets into training and test sets. You could use the `train_test_split` function from `scikit-learn` with `shuffle=False`. Alternatively, you can load the `split_data` function to split the data, as shown:

```
from utils import split_data
```

The following code demonstrates how to use the `split_data` function:

```
train, test = split_data(air_os)
print(f'Train: {len(train)} Test: {len(test)}')
>>
Train: 126 Test: 13
```

You could accomplish the same results using `scikit-learn`'s `train_test_split`, as shown:

```
from sklearn.model_selection import train_test_split
train, test = train_test_split(air_os, test_size=0.10, shuffle=False)
```

7. Certain algorithms benefit from scaling the data. You can use the `StandardScaler` class from `scikit-learn`. In this recipe, you will use a custom class (`Standardize`) that you can load from `utils.py`. The class has three main methods:
- The `fit_transform` method will fit on the training set and then transform both the training and test sets. Scaling is fitted only on the training data to prevent data leakage.
 - The `inverse` method is used to return a `DataFrame` to its original scale.
 - The `inverse_y` method is used to inverse the target variable (or a specific column and not the entire `DataFrame`).

8. The following code shows how you can use the `Standardize` class:

```
from utils import Standardize
scaler = Standardize()
train_s, test_s = scaler.fit_transform(train, test)
train_original = scaler.inverse(train_s)
y_train_original = scaler.inverse_y(train_s['y'])
```



You will be leveraging these functions in other recipes of this chapter for data preparation.

How it works...

Preparing time-series data for supervised learning is summarized in *Figure 11.2* and *Figure 11.3*. For example, in a regression problem, you are essentially transforming a univariate time series into a multiple regression problem. This transformation involves creating lagged features (independent variables) from past observations to predict the current or future value (dependent variable). You will explore this concept further in the *One-step forecasting with scikit-learn* recipe.

The window size parameter can be adjusted to fit your needs. In the recipe, we used a split window of five periods, but you should experiment with different window sizes.

There's more...

In this recipe, using the `generate_lagged_features` function, you generated additional columns to represent independent variables (features) used in model training. This process of creating new columns from the original data is known as **feature engineering**. Feature engineering is crucial for improving your model's performance by providing the model with additional informative features.

The **sliding window technique** is one approach to creating features from past observations. However, there are other techniques that you can use to extract features from time-series data and further prepare it for supervised learning.

Window size selection

The choice of **window size** is a critical decision when preparing time-series data for supervised learning. The window size determines how much historical data is used to predict future values and can significantly impact model performance. Consider the following when selecting a window size:

- **Seasonal patterns:** A larger window size can help capture longer-term dependencies and seasonal patterns, but it will also reduce the number of training samples
- **Natural frequency:** The window size should align with the natural frequency of the time series (e.g., daily, weekly, or monthly)
- **Trade-offs:** Larger windows provide more context but may introduce noise, while smaller windows may miss important patterns

Experiment with different window sizes to find the optimal configuration for your dataset.

Date-based features

You can create date-based features from the date column (or index), such as the following:

- Day of the week
- Year
- Month
- Quarter
- Season
- End of the month
- Is leap

Here is an example of how you can engineer date-related features using pandas:

```
df = daily_temp.copy()
# adding lagged features
df_lf = generate_lagged_features(df, window=5)

# adding additional date related features
df_lf['day_of_week'] = df_lf.index.dayofweek
df_lf['days_in_month'] = df_lf.index.days_in_month
df_lf['month_end'] = df_lf.index.is_month_end.astype(int)
df_lf['is_leap'] = df_lf.index.is_leap_year.astype(int)
df_lf['month'] = df_lf.index.month
```

Statistical features

Rolling statistics capture the evolving behavior of your time series and can help identify changing trends, volatility, and extreme values:

```
# Weekly average
df_lf['rolling_mean'] = df_lf['y'].rolling(window=7).mean()
# Volatility measure
df_lf['rolling_std'] = df_lf['y'].rolling(window=7).std()
# Recent Lows
df_lf['rolling_min'] = df_lf['y'].rolling(window=7).min()
# Recent highs
df_lf['rolling_max'] = df_lf['y'].rolling(window=7).max()
```

Note that rolling statistics will introduce NaN values for the first few rows, so you may need to drop them or fill them later.

Cyclical encoding

For cyclical features such as day of the week or month, sinusoidal encoding preserves the circular nature of the data. This is particularly useful for features that repeat over a fixed period (e.g., days, months, or hours).

Here's how you can encode cyclical features:

```
# Encoding of cyclical features
df_lf['month_sin'] = np.sin(2 * np.pi * df_lf.index.month/12)
df_lf['month_cos'] = np.cos(2 * np.pi * df_lf.index.month/12)
```

The general formula for **cyclical encoding** is as follows:

```
feature_sin = np.sin(2 * np.pi * value / period)
feature_cos = np.cos(2 * np.pi * value / period)
```

Here, we have the following:

- **value** is our temporal value (day, month, hour, etc.)
- **period** is the full cycle length (7 for days of the week, 12 for months, 24 for hours, etc.)

Take the following example:

- **Daily data** may exhibit weekly patterns (use `day_of_week_sin` with `period=7`)
- **Monthly data** often has yearly seasonality (use `month_sin` with `period=12`)
- **Hourly data** might have daily patterns (use `hour_sin` with `period=24`)

Experimentation

Even though you will be using the `generate_lagged_features` function throughout the chapter, you should explore other feature engineering techniques. Experimenting with the different approaches and algorithms may help you better capture patterns in your time-series data, leading to improved model performance. For more ideas, refer to *Chapter 5*, where you will find detailed techniques for working with date and time data.

Best practices

When engineering features, keep the following points in mind:

- **Feature scaling:** Apply feature scaling after creating all features to ensure proper normalization.
- **Multicollinearity:** Remove or combine **highly correlated features** to prevent multicollinearity, which can reduce model interpretability and performance.
- **Efficient data types:** Use efficient data types (e.g., `float32` instead of `float64`) to reduce memory usage.
- **Handling missing values:** Be aware that certain feature engineering techniques (such as rolling statistics and lag features) will introduce **NaN** values at the beginning of your time series. Either drop these rows, fill them using appropriate methods (forward fill, mean, etc.), or ensure your model can handle missing values.

- **Feature selection:** For time series specifically, not all engineered features contribute equally to forecast accuracy.

See also

- To learn more about the `StandardScaler` class from `sklearn`, you can read the official documentation here: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>.
- To learn more about the `train_test_split` function from `sklearn`, you can read the official documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.
- To learn more about feature selection in `sktime`, you can read the following documentation: https://www.sktime.net/en/stable/api_reference/auto_generated/sktime.transformations.series.feature_selection.FeatureSelection.html.

One-step forecasting with scikit-learn

In *Chapter 9*, you were introduced to statistical models such as Autoregressive (AR) type models. These statistical models are considered **linear**, meaning they assume the target variable depends linearly on its past values at specific time lags. In this recipe, you will transition from traditional **statistical algorithms** to **ML algorithms**. While statistical models such as AR are powerful at capturing linear dependencies, ML models offer greater flexibility to model complex, non-linear relationships often presented in real-world datasets.

You will train various linear models, such as **linear regression**, **elastic net regression**, **ridge regression**, **Huber regression**, and **Lasso regression**. These models assume a linear relationship between input variables (features) and the target. Additionally, you will explore **non-linear regressors** available in the `scikit-learn` library, such as **Random Forest**, **Support Vector Regression (SVR)**, **Gradient Boosting**, and **KNN**. Non-linear models are particularly useful when the data exhibits complex patterns that linear models might not capture effectively.

In the previous recipe, you learned how to transform a univariate time series into a format suitable for ML by converting it into **tabular format** with independent variables (features) and a dependent variable. This transformation leverages the sliding window technique, where past steps (lags) are used to create lagged features for model training. Conceptually, you can think of this process as converting a univariate time series into a **multiple regression problem**, as illustrated in the following diagram (*Figure 11.3*):

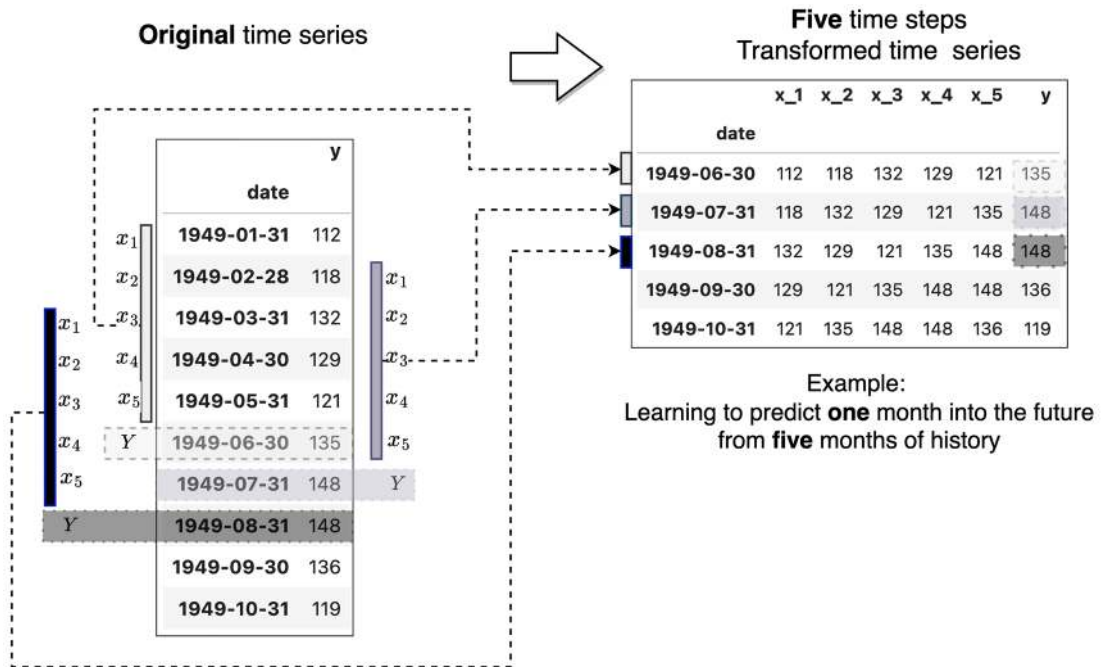


Figure 11.3: Transforming time series for supervised ML

For the representation in Figure 11.3, the multiple linear regression equation would be written as follows:

$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_5 X_5 + \epsilon$$

Here, we have the following:

- $\hat{y}$ is the estimated (*predicted*) value and is calculated as a weighted sum of the independent *lagged features*
- $\beta_1, \beta_2, \dots, \beta_5$ are the coefficients for each *lagged features*
- $X_1, X_2, \dots, X_5$ are the *lagged versions of the target variable created using the sliding window technique*
- β_0 is the intercept term
- ϵ is the residual or error term

You can also express this equation in **matrix notation** by adding an intercept term, $X_0 = 1$, as the first column in the design matrix, $\mathbf{X}$. This produces a simplified representation and allows for more efficient computation when dealing with multiple observations:

$$\hat{y} = \mathbf{X}\beta + \epsilon$$

This formula enables efficient computation of coefficients using linear algebra operations.

Here, we have the following:

- $\mathbf{X}$ represents the matrix of independent variables (features)
- $\boldsymbol{\beta}$ represents the vector of coefficients (including the intercept term)
- ϵ represents the residuals or errors

In **linear regression**, the goal is to minimize the **errors** or **loss**, which is the difference between the actual (observed) value, $\mathbf{Y}$, and the estimated (predicted) value, $\hat{\mathbf{Y}}$. More specifically, we take the **sum of squared errors**, also called the **Residual Sum of Squares (RSS)**, which serves as the cost function that we seek to minimize.

$$RSS = \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

Minimizing the RSS ensures that the model's predictions are as close as possible to the observed values. This gives us our **objective function** and can be written as follows:

$$\min \sum_{i=1}^n \left(Y_i - \sum_{j=1}^m \beta_j X_{ij} \right)^2$$

In practice, the objective function is often expressed as minimizing the **Mean Squared Error (MSE)**, which is the average of the squared differences between the actual and predicted values. Mathematically, this is the **RSS** divided by the number of observations, n .

$$MSE = \frac{RSS}{n}$$

The objective of model training is to find the set of coefficients that minimizes the loss function. Minimizing the MSE ensures that the model's predictions are as close as possible to the actual values.

Linear models assume a straight-line relationship between input features and the target. However, real-world time-series data often contains complex patterns that linear models may struggle to capture. To account for such complexities, it is useful to explore non-linear regressors, which can offer better performance when dealing with non-linear relationships.

Some common non-linear models to explore are the following:

- **Random Forest:** A tree-based **ensemble method** that builds multiple decision trees on random subsets of features and observations, then averages their predictions. This approach reduces overfitting and improves accuracy, making it well suited for capturing complex patterns in time-series data. In scikit-learn, you will use the `RandomForestRegressor` class.
- **SVR:** A regression variant of the **SVM** classification algorithm. SVR transforms data into a higher-dimensional space using a **kernel function** to handle non-linear relationships. The choice of the kernel (e.g., linear, polynomial, or **Radial Basis Function (RBF)** kernel) is crucial to determine how the input data is transformed. In scikit-learn, you will be using the `SVR` class.

- **Gradient Boosting:** An **ensemble method** that builds a series of **weak learners** (typically decision trees), where each learner improves upon the previous one by attempting to correct its errors. This iterative process improves model performance over time. In scikit-learn, you will be using the `GradientBoostingRegressor` class.
- **KNN:** A simple algorithm that predicts the target based on the average values of its nearest K data points. In scikit-learn, you will be using `KNeighborsRegressor`.
- **Extremely Randomized Trees (Extra Tree) regression:** This algorithm is similar to *Random Forest*, but with additional randomness introduced during the construction of the decision trees. In scikit-learn, you will be using the `ExtraTreesRegressor` class.

By incorporating these non-linear models, you can capture complex trends and seasonality in the data that linear models may miss due to their oversimplified assumptions. When deciding between linear and non-linear models, consider the nature of your time-series data. If the data exhibits clear linear trends and seasonality, linear models such as ridge or Lasso regression may suffice. However, for more complex patterns, non-linear models such as Random Forest and Gradient Boosting are often more effective.

In this recipe, you will compare the performance of these different models and select the one that works best for your time-series data.

How to do it...

In this recipe, you will continue using the three DataFrames (*air*, *energy*, and *daily_temp*) loaded in the *Technical requirements* section of this chapter. You will leverage the functions and classes loaded and used in the previous recipe, *Preparing time series data for supervised learning*:

1. Start by loading the necessary libraries and creating copies of the DataFrames to avoid overwriting the original data:

```
from sklearn.linear_model import (LinearRegression,
                                  ElasticNet,
                                  Ridge,
                                  Lasso,
                                  HuberRegressor)

from sklearn.ensemble import (RandomForestRegressor,
                              GradientBoostingRegressor,
                              ExtraTreesRegressor)

from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor

# Create copies of the DataFrames
air_cp = air.copy()
en_cp = energy.copy()
dw_cp = daily_temp.copy()
```

2. Use the `handle_missing_data` function to ensure there are no missing values in the datasets:

```
for df in [air_cp, en_cp, dw_cp]:
    handle_missing_data(df)
```

3. Convert the time-series DataFrames into a supervised learning problem using the `generate_lagged_features` function with a window size of 10:

```
air_ml = generate_lagged_features(air_cp, 10)
en_ml = generate_lagged_features(en_cp, 10)
dw_ml = generate_lagged_features(dw_cp, 10)
```

Feel free to change the window size.

4. Split and scale the data using the `preprocess` function:

```
def preprocess(df, split):
    train, test = split_data(df, test_split=split)
    scale = Standardize()
    train_s, test_s = scale.fit_transform(train, test)
    return train_s, test_s, scale
```

The `preprocess` function combines two essential steps: splitting the data into training and test sets using the `split_data` function, and scaling the data using the `Standardize` class. This ensures consistency and simplifies the workflow.

```
train_air, test_air, scale_air = preprocess(air_ml, split=0.10)
train_en, test_en, scale_en = preprocess(en_ml, split=0.10)
train_dw, test_dw, scale_dw = preprocess(dw_ml, split=0.10)
```

5. Define error metrics. Common error metrics for regression include **MSE** and **Root Mean Squared Error (RMSE)**. These metrics are *scale-dependent*, meaning that changes in the scale of the data will impact their values.

For time-series forecasting, a popular error metric is **Mean Absolute Percentage Error (MAPE)**. MAPE is often considered more intuitive to interpret because it is scale-independent and expressed as a percentage. However, MAPE has limitations and may not be suitable for certain problems. For instance, with the daily temperature data, MAPE puts a heavier penalty on negative errors (e.g., when temperatures are below 0 degrees Celsius). MAPE struggles when the values approach or equal 0 since you cannot divide by 0 (e.g., temperatures at 0 degrees Celsius can cause issues). Additionally, interpreting temperature as a percentage may not make sense in this context.

It's a good practice to capture different metrics; in this case, you will capture MAE, RMSE, MAPE, and MASE. The `sktime` library provides an implementation for MASE, which is not available in `scikit-learn`. The **Mean Absolute Scaled Error (MASE)** metric has been proposed as an alternative to address MAPE's limitations by scaling errors based on a naïve forecast benchmark. This makes MASE scale-independent and robust across a wider range of time-series data.

```

from sktime.performance_metrics.forecasting import(
    MeanAbsolutePercentageError,
    MeanSquaredError,
    MeanAbsoluteError,
    mean_absolute_scaled_error as mase
)

```

Create instances of these metrics to evaluate models based on their **Mean Absolute Error (MAE)**, **MSE**, **MAPE**, and **MASE** scores:

```

mae = MeanAbsoluteError()
mse = MeanSquaredError()
mape = MeanAbsolutePercentageError()

```

Note that you will be calculating RMSE as the square root of MSE using `np.sqrt(mse(actual, yhat))`.

6. Define the `train_models` function to train a model, make predictions, and evaluate performance. The function takes the training and test sets, fits the model on the training set, uses the scaler object to inverse the results into their original scale, and then evaluates the model against the test set using MAPE, MASE, MAE, and RMSE:

```

def train_model(train, test, regressor, reg_name, sc):
    X_train, y_train = train.drop(columns=['y']), train['y']
    X_test, y_test = test.drop(columns=['y']), test['y']
    print(f'training {reg_name} ...')

    regressor.fit(X_train, y_train)

    yhat = regressor.predict(X_test)
    actual = y_test.values
    yhat = sc.inverse_y(yhat)
    actual = sc.inverse_y(actual)

    rmse_test = np.sqrt(mse(actual, yhat))
    mae_test = mae(actual, yhat)
    mape_test = mape(actual, yhat)
    mase_test = mase(actual, yhat, y_train=sc.inverse_y(y_train))
    residuals = actual - yhat

    model_metadata = {
        'Model Name': reg_name,
        'Model': regressor,
        'RMSE': rmse_test,

```

```

    'MAPE': mape_test,
    'MASE': mase_test,
    'MAE': mae_test,
    'yhat': pd.Series(yhat, index=y_test.index),
    'resid': residuals,
    'actual': sc.inverse_y(y_test)}
    return model_metadata

```

The function returns a metadata dictionary that includes the model, predictions, residuals, and evaluation scores.

7. Create the regressors dictionary, which maps model names to their corresponding scikit-learn classes. This allows you to easily iterate through multiple models during training and evaluation. It also makes it easier to update the dictionary with additional regressors if needed:

```

regressors = {
    'Linear Regression': LinearRegression,
    'Elastic Net': lambda: ElasticNet(alpha=0.5),
    'Ridge Regression': lambda: Ridge(alpha=0.5),
    'Lasso Regression': lambda: Lasso(alpha=0.5),
    'Huber Regression': HuberRegressor,
    'Random Forest Regression': lambda: RandomForestRegressor(
        n_estimators=100),
    'Gradient Boosting Regression': lambda: GradientBoostingRegressor(
        n_estimators=100),
    'Extra Tree Regression': lambda: ExtraTreesRegressor(
        n_estimators=100),
    'Support Vector Regression (RBF)': lambda: SVR(kernel='rbf'),
    'Support Vector Regression (linear)': lambda: SVR(
        kernel='linear'),
    'K-Nearest Neighbor Regression': lambda: KNeighborsRegressor(
        n_neighbors=5)
}

```

The three regressors, **ridge**, **Lasso**, and **ElasticNet**, include a regularization (penalization) term added to the objective function to prevent overfitting. All three take an alpha (α) parameter, which determines the penalization factor for the coefficients. You can experiment with this hyperparameter; for now, we will use the value (0.5). For ensemble methods such as **Random Forest**, **Gradient Boosting**, and **Extra Tree regression**, the `n_estimators` parameter controls the number of trees in the model. The KNN has a `k` parameter, which determines the number of k-neighbors. We will go with 5 neighbors for now.

In a later recipe, you will learn how to optimize for these parameters using **hyperparameter tuning**.

8. Use the `train_different_models` function to iterate through the regressors dictionary, train each model by calling the `train_model` function, and store the results in a list. The `train_model` function fits and evaluates one model at a time.

Each item in the list is a dictionary containing the model's metadata, predictions, and evaluation metrics:

```
def train_different_models(train, test, regressors, sc):
    results = []
    for reg_name, regressor in regressors.items():
        reg = regressor()
        results.append(train_model(train, test, reg, reg_name, sc))
    return results
```

Pass the dictionary of regressors along with the scaler, training, and test sets to the `train_different_models` function and store the results:

```
air_results = train_different_models(train_air, test_air,
                                     regressors, scale_air)
en_results = train_different_models(train_en, test_en,
                                    regressors, scale_en)
dw_results = train_different_models(train_dw, test_dw,
                                    regressors, scale_dw)
air_results = train_different_models(train_air, test_air, regressors)
```

Here is an example output:

```
training Linear Regression ...
training Elastic Net ...
training Ridge Regression ...
training Lasso Regression ...
training Huber Regression ...
training Random Forest Regression ...
training Gradient Boosting Regression ...
training Extra Tree Regression ...
training Support Vector Regression (RBF) ...
training Support Vector Regression (linear) ...
training K-Nearest Neighbor Regression ...
```

9. Load the `evaluate_results` helper function, which takes your results and a specified metric and returns a DataFrame with the top models sorted based on the metric score you select.

```
from utils import evaluate_results
```


Use the `evaluate_results` function and print the top three models:

```
air_results_df = pd.DataFrame(air_results)
air_results_sorted = evaluate_results(air_results_df)
print(air_results_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Ridge Regression	39.358884	0.076121	1.390807
1	Linear Regression	39.820148	0.077452	1.411072
2	Support Vector Regression (linear)	43.608249	0.079431	1.448104

```
en_results_df = pd.DataFrame(en_results)
en_results_sorted = evaluate_results(en_results_df)
print(en_results_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Extra Tree Regression	105.745977	0.040870	0.327420
1	K-Nearest Neighbor Regression	109.006412	0.041790	0.334015
2	Support Vector Regression (RBF)	107.643507	0.045597	0.359839

```
dw_results_df = pd.DataFrame(dw_results)
dw_results_sorted = evaluate_results(dw_results_df)
print(dw_results_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Support Vector Regression (RBF)	4.654812	1.908226	0.891606
1	Extra Tree Regression	4.738185	2.051619	0.925450
2	Random Forest Regression	4.771619	1.667300	0.938026

You can run the `evaluate_results` function using a different scoring metric to sort by.

10. `train_different_models` returned a list of dictionaries (metadata per model) that contains the actual test results (`actual`), the forecast value (`yhat`), and the residuals (`resid`). You can use these to visualize the model's performance—a plot to compare actual versus forecast values, as well as an ACF plot for the residuals.

The `plot_results` function generates two plots for each model: a line plot comparing actual versus predicted values, and an autocorrelation plot of the residuals to help diagnose model performance and identify potential issues.

The `plot_results` function utilizes the `plot_acf` plotting function from `statsmodels` to evaluate the residuals.

```
from utils import plot_results
plot_results(air_results_df, 'Air Passengers')
plot_results(en_results_df, 'Energy Consumption')
plot_results(dw_results_df, 'Daily Temperature')
```

The preceding code should produce two plots for each of the 11 models (a total of 22 plots for each result set).

For each model, the function will output a line plot comparing out-of-sample data (test data) against the forecast (predicted values) and a residual autocorrelation plot.

For illustration, a single example is shown from the energy consumption dataset, where the **Extra Tree regression** model achieved the best performance.

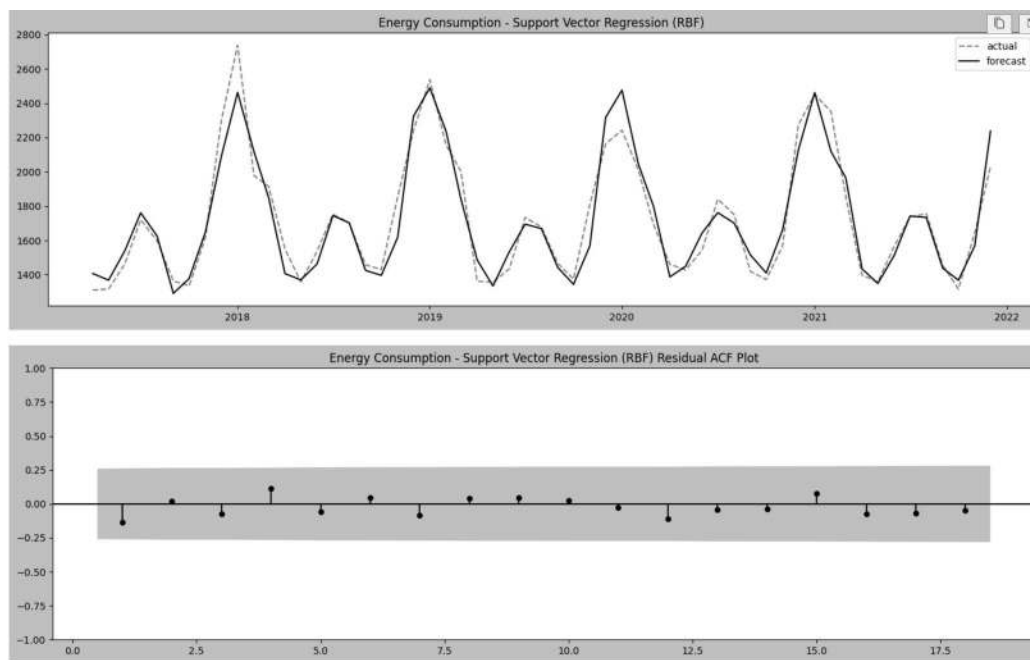


Figure 11.4: Extra Tree regression for the energy consumption dataset

The accompanying Jupyter notebook shows the plots for each model per dataset. Observe from the notebook how the models rank and behave differently on different time-series processes.

Later in this chapter, you will explore hyperparameter tuning, in the *Hyperparameter tuning with sktime* recipe.

How it works...

Once a time series is prepared for supervised ML, you can use any regression algorithm to train a model. This process is summarized in *Figure 11.3*. In this recipe, you used the `generate_lagged_features` function to transform the time series into a format suitable for ML. Specifically, you prepared the data so that a sequence of past observations (independent variables) is used to predict a **single** future value (**one-step forecast**). One-step forecasting in time series means predicting the very next value based on what happened before.

The visualization of results might seem to show multiple steps being forecast at once, but it's important to understand that each prediction point in the test set is generated individually using the past 10 periods (lags) as inputs. The plot displays these individual one-step forecasts together for evaluation purposes, but each forecast is made independently. This means for unseen data, you will need to provide the last 10 known periods to get back the next value—a one-step future.

One-step forecasting is a fundamental task in time-series forecasting, often used for short-term predictions where the immediate next value is critical, such as daily sales or temperature forecasts. In the next recipe, you will explore different methods for multi-step forecasting, where you actually predict multiple periods into the future.

The choice of error metrics is important when evaluating forecasting models. You implemented four complementary metrics, each with different strengths:

- **MAE**: Simply measures the average magnitude of errors without considering their direction
- **RMSE**: Gives higher weights to larger errors, making it useful when large errors are particularly undesirable
- **MAPE**: Provides errors as percentages, making it easier to interpret, but can be problematic with values near 0
- **MASE**: Compares your forecast to a naïve forecast, making it useful across different datasets

Different algorithms performed best on each dataset, which highlights an important point: there's no one-size-fits-all approach to time-series forecasting. For the air passengers data, **ridge regression** performed best, likely because the **regularization** helped manage the correlations between lag features. For energy consumption, **Extra Tree regression** excelled, probably due to its ability to capture **non-linear patterns**. The daily temperature dataset was best modeled by **SVR with an RBF kernel**, which can effectively handle the seasonal patterns in temperature data.

The ACF plots of the residuals help you determine whether your model has captured all the patterns in the data. Ideally, the residuals should look like random noise with no significant autocorrelation, indicating that the model has extracted all the useful information from the data.

There's more...

Recall that we stored the actual fitted model and thus we can inspect some model attributes further. In the case of the air passengers dataset, ridge regression and linear regression performed well, indicating that simple linear models are sufficient.

Feature selection and regularization

Further, the three regression models, **ElasticNet**, **Lasso**, and **ridge**, add a **regularization** (penalization) term to the objective function we want to minimize. In **Lasso regression**, the regularization term can reduce the coefficient of the least important features (independent variables) to 0, thus eliminating them. This added penalization term is called **L1 regularization**. In **ridge regression**, the regularization term is referred to as **L2 regularization** and can shrink the coefficients of the least important features, but *does not eliminate* them (no zero coefficients). **ElasticNet regression**, on the other hand, is a hybrid between the two by combining L1 and L2 regularization terms.

Regularization helps avoid overfitting during training and allows the model to generalize better. Additionally, regularization, particularly **L1 regularization**, can be used for **feature selection**.

Inspecting the coefficients can help us understand how each feature impacts the outcome variable—how a one-unit change in the corresponding independent variable causes a change in the target dependent variable while holding all variables constant.

You can inspect the coefficients to observe the effects, as shown in the following code block for the air passengers dataset:

```
cols = ['Model Name', 'Model']
air_models = air_results_df.iloc[0:4][cols]

for row in air_models.iterrows():
    print(row[1]['Model Name'])
    print(row[1]['Model'].coef_)
```

This should produce the following:

```
Linear Regression
[ 0.26681178  0.23043566 -0.42770452  0.24811951 -0.19850306  0.36627057
 -0.34839752  0.17465605 -0.4063289   1.0808261 ]

Elastic Net
[0.18648073 0.0263418  0.          0.          0.          0.
 0.          0.          0.10713549 0.34068171]

Ridge Regression
[ 0.33843604  0.11322012 -0.30232719  0.12859953 -0.06811208  0.2329643
 -0.22611879  0.03192433 -0.22325426  0.95924758]

Lasso Regression
[0.00797701 0.          0.          0.          0.          0.
 0.          0.          0.          0.44080144]
```

The air passengers data has 10 features. Observe how the **linear regression** model estimated the highest coefficient value (*weights*) for the last feature (lag 10). On the other hand, the **ElasticNet regression** model eliminated the middle features (lags 3 to 8) by estimating the coefficients at 0, indicating that they do not contribute to the model's prediction, while keeping the first two and last two features (lags). The **ridge regression** model produced similar results to the linear regression model, reducing the weights of the least significant features (thus shrinking their effect). Lastly, the **Lasso regression** model only deemed the last feature as the most significant contributor to the prediction, with a small contribution by the first feature (lag 1) and eliminating the rest with zero coefficients.

Recall that these features were engineered and represent lags or the past values of the dependent variable (y). The coefficients from the four models suggest that the 10th feature (the most recent lag or value) is significant in making a future prediction.

You can plot the coefficients for the ridge model as shown:

```
y_test = test_air['y']
X_test = test_air.drop(columns=['y'])

air_ridge = air_results_df.iloc[2]['Model'].coef_
title='Air Passenger Ridge Regression Coefficient Plot'
pd.DataFrame(air_ridge, X_test.columns).plot(kind='barh', title=title);
```

The output visualizes the coefficient magnitudes across all lag features of the ridge regression model, as shown in *Figure 11.5*.

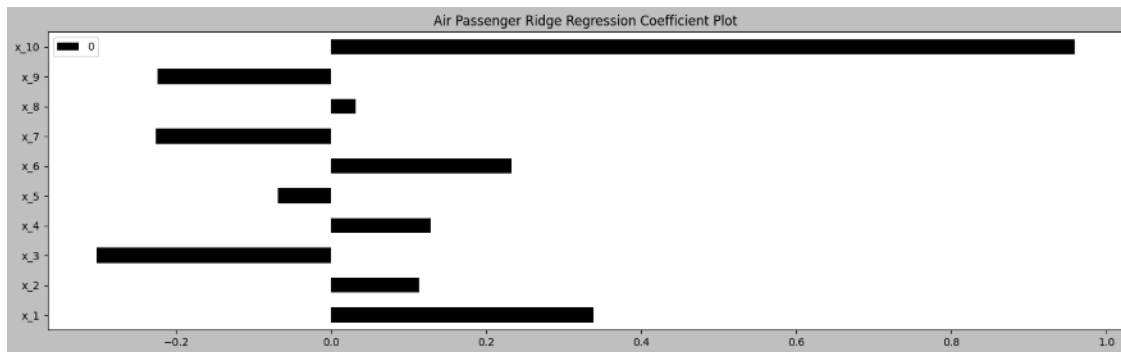


Figure 11.5: Plot of ridge regression model coefficients

Features with higher absolute coefficients have a stronger impact on the prediction. Positive coefficients indicate that an increase in the feature leads to an increase in the prediction, while negative coefficients indicate the opposite relationship.



Regularization not only helps with feature selection (in the case of Lasso) but also reduces overfitting by shrinking large coefficients that might cause the model to perform well on training data but poorly on unseen data.

Feature importance in ensemble methods

Several of the ensemble method models provide a `feature_importances_` attribute. Let's inspect the winning model for the energy consumption dataset, `ExtraTreeRegressor`:

```
en_features = en_results_df.iloc[7]['Model'].feature_importances_
title = 'Energy Consumption ExtraTreeRegressor Feature Importances'
pd.DataFrame(en_features, X_test.columns).plot(kind='barh', title=title);
```

The output visualizes the feature importance scores from the `ExtraTreeRegressor` model, as shown in *Figure 11.6*.

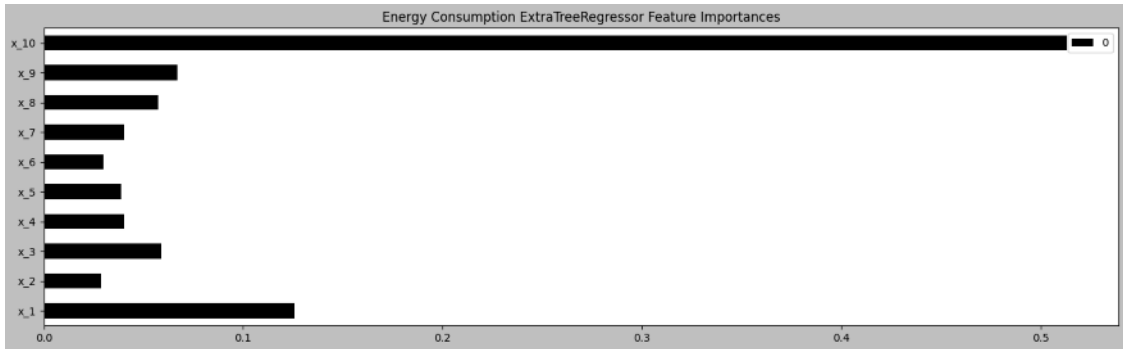


Figure 11.6: Plot of feature importance from `ExtraTreeRegressor`

Figure 11.6 reveals that the most recent lag (`x_10`) is by far the most important feature for the `ExtraTreeRegressor` model, with an importance score approximately five times higher than any other feature. Interestingly, both the most recent lag (`x_10`) and the oldest lag (`x_1`) show higher importance than the intermediate lags, suggesting that both immediate and longer-term patterns influence energy consumption forecasting.

Scikit-learn provides another option for model inspection and understanding feature importance through the `permutation_importance` function. Instead of just relying on model coefficients or internal metrics, the **permutation importance** is a handy technique to assess the importance of different features and their contribution to a model's prediction by randomly shuffling or **permutating** the values of each feature and re-evaluating the model performance on the dataset with the permuted feature. Then the importance of each feature is calculated. In other words, if shuffling a feature reduces the model's performance, it indicates that the feature is important for predictions. Let's apply this to the daily temperature dataset and the **Support Vector Regression (RBF)** model:

```
from sklearn.inspection import permutation_importance
svr_rbf = dw_results_df.iloc[8]['Model']

y_test = test_dw['y']
X_test = test_dw.drop(columns=['y'])

result = permutation_importance(svr_rbf, X_test, y_test,
```

```

n_repeats=10, random_state=42)
perm_importance = result.importances_mean
title='Daily Temperature SVR with RBF Kernel Permutation Importance'
pd.DataFrame(perm_importance, X_test.columns).plot(kind='barh', title=title);

```

The output shows the permutation importance for each lag feature, as displayed in *Figure 11.7*.

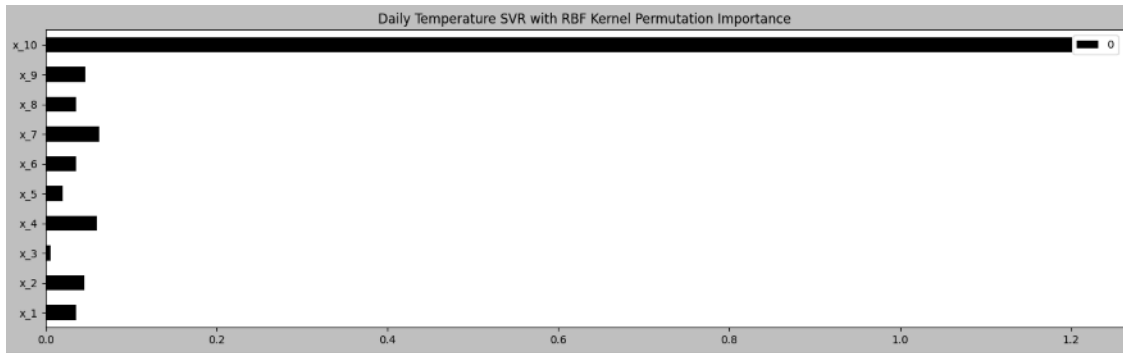


Figure 11.7: Plot output from permutation importance for SVR on daily temperature data

Figure 11.7 demonstrates an even stronger dominance of the most recent lag (x_{10}) in the daily temperature dataset, with permutation importance approximately 15 times higher than any other feature. This extreme importance difference suggests that for daily temperature forecasting, the most recent temperature reading is overwhelmingly the best predictor of the next day's temperature. This makes intuitive sense for temperature data, as weather patterns typically don't change drastically from one day to the next. The permutation importance technique effectively highlights this strong dependency, which might be less apparent when using only model-specific feature importance methods.

Both *Figure 11.6* and *Figure 11.7* display feature importance values where higher importance indicates a more significant contribution of that feature to the model's performance. In both cases, we see that the most recent lag (x_{10}) has the highest importance, though the effect is much more pronounced in the daily temperature dataset (*Figure 11.7*), where the most recent temperature reading dominates all other features.

Note that the **permutation importance** technique is **model agnostic**, and you can apply it to other models. You can change the `n_repeats` parameter, which influences the number of times to permute a feature. Setting `n_repeats` to a higher value will make the feature importance estimates more stable but will also increase computation time.

The following table provides a summary of feature importance methods:

Method	Models Applicable	Description	Considerations for Time Series
Coefficient inspection	Linear, ridge, Lasso, Elastic Net	Examines model weights; L1 can set coefficients to 0	Reflects linear relationships; larger coefficients for recent lags often indicate temporal importance
Feature importance	Ensemble (e.g., Extra Tree, Random Forest)	Based on impurity reduction across trees	Captures non-linear relationships; can identify complex temporal patterns
Permutation importance	Model-agnostic (works with any model)	Measures performance drop when shuffling features	Computationally intensive but provides a consistent comparison across different model types

Table 11.1: Summary of feature importance methods

See also

- To learn more about the different regression models available in the `scikit-learn` library, visit the main regression documentation here: https://scikit-learn.org/stable/supervised_learning.html#supervised-learning.
- To learn more about **permutation feature importance**, you can visit the `scikit-learn` page here: https://scikit-learn.org/1.5/modules/permutation_importance.html
- To learn more about how different ML algorithms for time-series forecasting compare, you can reference the following research paper:
 - Ahmed, Nesreen K., Amir F. Atiya, Neamat El Gayar, and Hisham El-Shishiny. *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*. *Econometric Reviews* 29, no. 5–6 (August 30, 2010): 594–621. <https://doi.org/10.1080/07474938.2010.481556>

Multi-step forecasting with `scikit-learn`

In the previous recipe, *One-step forecasting with `scikit-learn`*, you implemented a **one-step forecast**. In this approach, a sequence of values from the past 10 periods ($x_1, \dots, x_t$ where $t = 10$) is used to predict the next period, referred to as y_{t+1} . This is known as **one-step forecasting**, as shown in *Figure 11.2*.

For example, in the case of monthly energy consumption data, to forecast for December 2021, you would provide data for the past 10 months (February to November 2021). This approach works well for monthly or quarterly data, but what about more granular data, such as daily or hourly? In the case of daily temperature data, this setup requires you to provide temperature values for the past 10 days to obtain a one-day forecast. This may not be an efficient approach because you would have to wait for the next day to observe a new value to generate another one-day forecast.

But what if you need to predict more than one future step? For example, what if you want to forecast three months into the future ($y_{t+1}, y_{t+2}, y_{t+3}$) using the same 10-month window of data ($x_1, \dots, x_{10}$). This is known as **multi-step forecasting**. The aim is to predict a sequence of steps in the future beyond just the immediate next one.

In the *Preparing time-series data for supervised learning* recipe, we referenced the paper *Machine Learning Strategies for Time Series Forecasting*, which introduces *four strategies* for multi-step forecasting:

- **Recursive strategy:** Uses a single model recursively, feeding predictions back as inputs
- **Direct strategy:** Trains separate models for each future time step
- **Direct-Recursive (DirRec) strategy:** Combines aspects of both recursive and direct strategies
- **Multiple-output strategy:** Predicts all future time steps at once

In this recipe, you will implement the **recursive strategy** to forecast multiple steps into the future. By reusing the same model from one-step forecasting, the recursive strategy allows you to use forecasted values as input for future predictions. This approach is simple to implement and useful when you need to forecast further beyond the test data at hand.

However, it's important to note that the recursive strategy has a key limitation: **error accumulation**. Since each future forecast depends on previous predictions, any error in an earlier step can propagate and grow with each subsequent forecast. We will explore strategies to mitigate this in later recipes.

The recursive strategy works as follows: each forecasted value is fed back into the model as input to predict the next time step. For instance, if we want to forecast five periods into the future, we start by providing the most recent 10 observations (e.g., from 2016-12-28 to 2017-01-01). We use these values to forecast the next period, 2017-01-02. This forecast is then used as input for the next step, 2017-01-03, and so on.

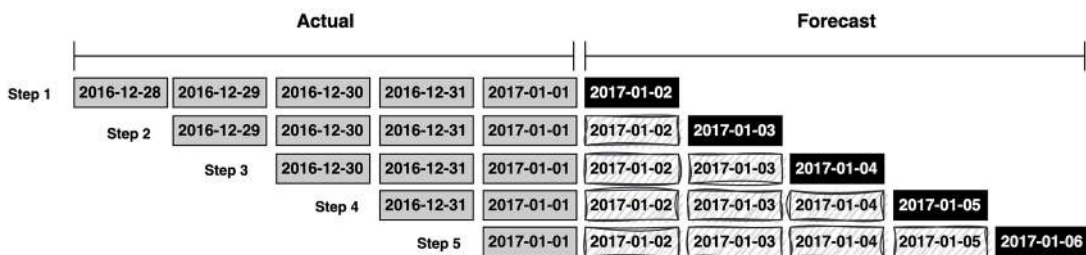


Figure 11.8: Sliding window (five periods) with multi-step (recursive) forecasts of daily temperature

In Figure 11.8, the gray boxes represent observed (actual) values, the black boxes represent the **newest forecast** at each step, and the sketched boxes represent **previously forecasted** values that are now being used as inputs. This recursive process continues until all future steps (horizons) are predicted.

While the recursive strategy is straightforward to implement for multi-step forecasting, we'll also explore later in this recipe how other approaches, such as the **direct** and **multiple-output** strategies, can help address the error accumulation problem.

How to do it...

In this recipe, you will use the models obtained from the previous recipe, *One-step forecasting with scikit-learn*. A **recursive multi-step strategy** will be used in the forecasting (prediction) phase:

1. From the previous recipe, *One-step forecasting with scikit-learn*, you should have three DataFrames (`air_results`, `dw_results`, and `en_results`) that contain the results from the trained models. If you need to regenerate these results, you should refer to the complete code in that recipe, as there are several preprocessing and model training steps required.

For reference, the final steps that produced these DataFrames were the following:

```
air_results = train_different_models(train_air, test_air, regressors)
en_results = train_different_models(train_en, test_en, regressors)
dw_results = train_different_models(train_dw, test_dw, regressors)

air_results = pd.DataFrame(air_results)
en_results = pd.DataFrame(en_results)
dw_results = pd.DataFrame(dw_results)
```



The following steps will use `air_results` (from the air passengers dataset) for demonstration. You can apply the same process to the remaining DataFrames (as demonstrated in the accompanying Jupyter notebook).

2. Extract the models and their names from the resulting DataFrames. Recall that there are 11 trained models:

```
models = air_results[['Model Name', 'Model']]
```

3. Create the `recursive_forecast()` function. This function implements the **recursive strategy** using a for loop. At each iteration, the model's `predict()` method is used to generate a one-step forecast, and the predicted value is fed back as input for the next step:

```
def recursive_forecast(data, model, steps=10):
    forecast = []
    # Avoid modifying the original data
    data = data.copy()

    for i in range(steps):
        one_step_pred = model.predict(np.array(data).reshape(1, -1))[0]
        forecast.append(one_step_pred)

        # Update the input data for the next step
        _ = data.pop(0) # Remove the oldest observation
        data.append(one_step_pred) # Add the latest prediction

    return np.array(forecast)
```

- Alternatively, you can use a **NumPy** array instead of a Python list. In NumPy, you can use the `np.roll()` function as opposed to the `pop()` and `append()` methods:

```
#another version using NumPy
def recursive_forecast(data, model, steps=10):
    forecast = []
    # Avoid modifying the original data
    data = np.array(data, copy=True)

    for i in range(steps):
        one_step_pred = model.predict(data.reshape(1, -1))[0]
        forecast.append(one_step_pred)

        # Update the input data for the next step
        data = np.roll(data, shift=-1) # Shift elements to the left
        data[-1] = one_step_pred # Replace last element with new prediction

    return np.array(forecast)
```

Both implementations work similarly, but using NumPy may be more efficient for larger datasets.

- Capture the last row from the `test_air` DataFrame (created from the previous recipe when splitting and scaling the datasets). This represents the 10 most recent observations, which will serve as input to the recursive forecasting loop:

```
air_ms = test_air.drop(columns=['y']).iloc[-1].tolist()
```

Since we are using a sliding window approach with a window size of 10, this row corresponds to the most recent 10 observations (the last row).

- Pass the extracted data (`air_ms`), the model, and the number of future steps (e.g., 10 steps) to the `recursive_forecast` function:

```
frcst_air = {}
for i in models.iterrows():
    pred = recursive_forecast(air_ms, model=i[1]['Model'], steps=10)
    pred = scale_air.inverse_y(pred)
    frcst_air[i[1]['Model Name']] = pred
```

The predicted values are stored in the `frcst_air` dictionary. The `inverse_y` method from `scaler_air` is used to scale the predictions back to their original units. As a reminder, this scaling object (`scale_air`) was created during the preprocessing steps in the previous recipe.

7. Create a plot to compare actuals versus predictions for the air passengers data:

```
for name, pred in frcst_air.items():
    # extract the last 10 rows from the original dataset air_cp
    actual = air_cp.iloc[-10:]
    ax = actual.plot(style='k-.')
    pd.Series(pred, index=air_cp.index[-10:]).plot(style='k--o', ax=ax)
    plt.title(f'Recursive Multiple Forecasting with {name}')
    plt.legend(['actual', 'forecast'])
    plt.show()
```

The preceding code will produce 11 plots, one for each model. The following is an example output for the **linear regression** and **Huber regression** models.

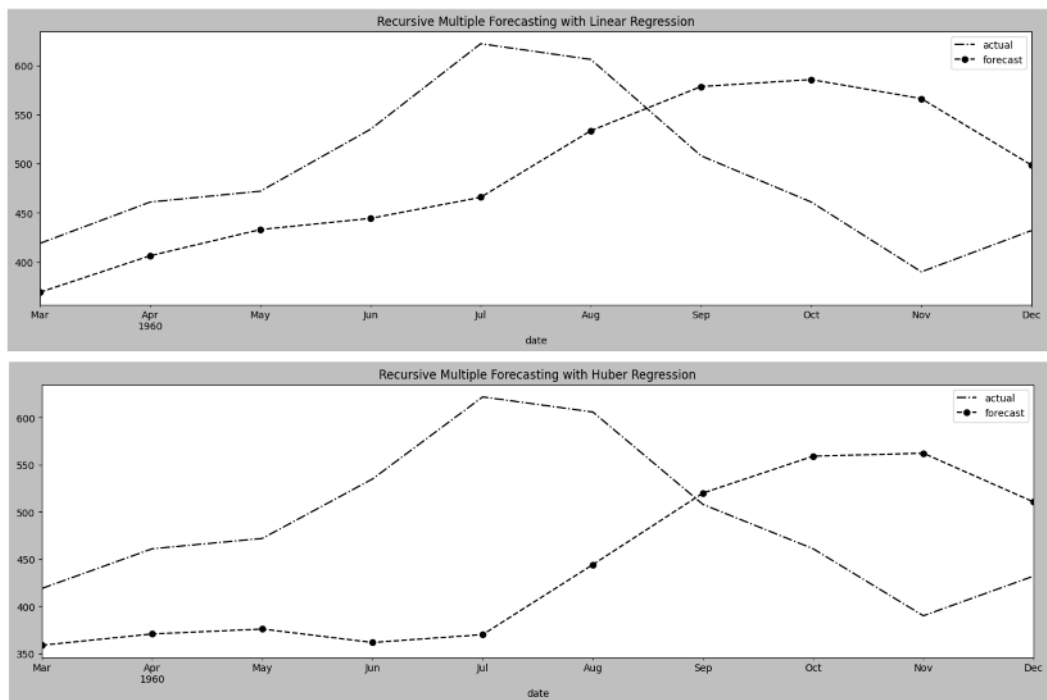


Figure 11.9: 10-day forecast using a recursive multi-step forecast for air passengers data

One challenge with the recursive strategy is **error accumulation**. As each prediction relies on previous estimates, any small errors will propagate and compound as you forecast further into the future. For these reasons, recursive forecasting tends to be most effective for short-term predictions (a few steps ahead) and particularly for short, contiguous blocks of lead times. It works best when the model has high accuracy for one-step predictions and when the time series exhibits relatively stable patterns. For longer forecast horizons, an alternative strategy such as **direct** or **multiple-output** forecasting may be more suitable.

How it works...

Multi-step forecasting is essential in situations where you need to predict more than one future step. In this recipe, you implemented a **recursive strategy** to produce a multi-step forecast, which is essentially a one-step forecast repeated multiple times, once for each future step. At each iteration, the prediction from the previous step is used as input to forecast the next time step.

One significant drawback of the recursive strategy is error accumulation. Since the model uses previously predicted values as inputs for future predictions, any errors in the initial forecast propagate and compound over subsequent steps. This causes the forecast to drift increasingly from the true values as the horizon extends. This makes the recursive strategy less reliable for longer forecast horizons.

To illustrate this, *Figure 11.10* shows an example of extending the forecast for 20 steps for the air passengers dataset. The first 10 steps can be compared against actual values, while the remaining 10 steps forecast beyond the available data. You can observe how the prediction errors accumulate over time, with the forecasted values increasingly diverging from the actual observations in the first half, and likely continuing this pattern of error accumulation in the extended forecast period.

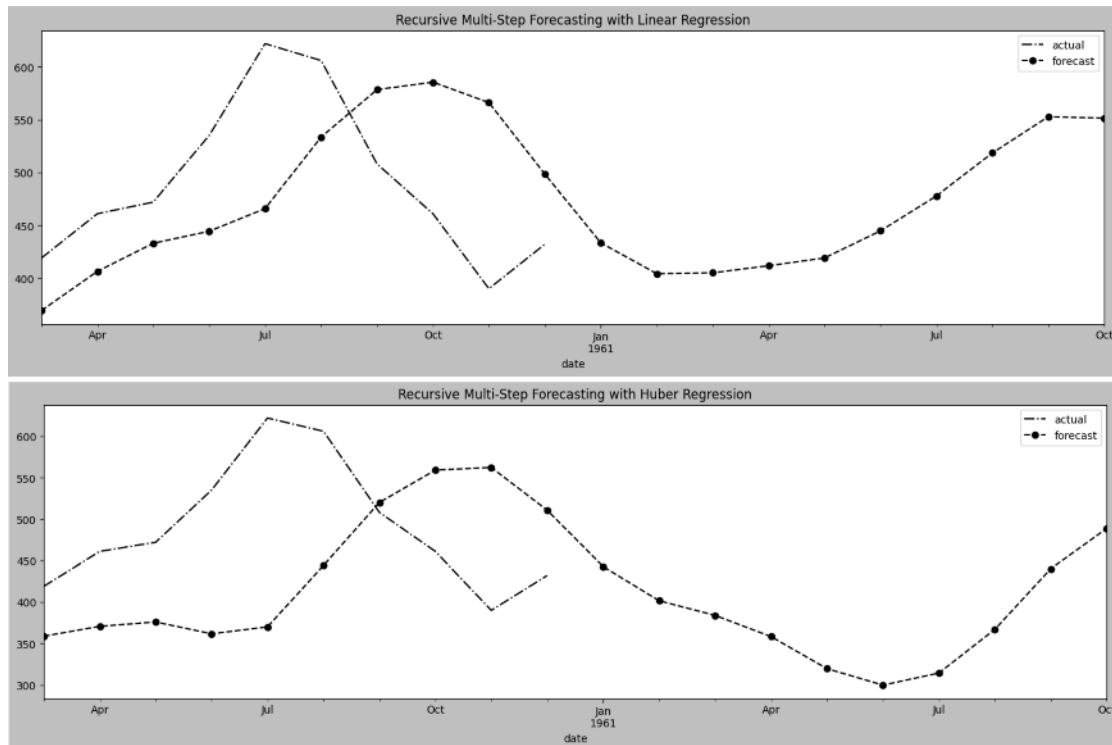


Figure 11.10: Example of using a recursive strategy for a longer future horizon (20 days) for air passengers data

The recursive approach is just one of several strategies for multi-step forecasting. Each has its own advantages and limitations.

The **direct strategy** creates a separate model for each future step (e.g., one model for prediction at $t+1$, another model at $t+2$, etc.). Each model produces a one-step forecast independently, without relying on predictions from previous steps. While this avoids error accumulation, it introduces some challenges:

- **Complexity:** Maintaining multiple models can become complex—producing a 10-day forecast requires 10 separate models
- **High variance:** Since each model operates independently, it may fail to capture dependencies between future steps, leading to inconsistent predictions

The **DirRec strategy** is a hybrid approach that combines elements of both the recursive and direct strategies, offering a way to mitigate their individual shortcomings. By using a combination of recursive and direct modeling, **DirRec** attempts to capture dependencies between predictions while reducing the impact of error accumulation. Similar to how *Figure 11.8* shows the recursive approach with a sliding window of five periods, the **DirRec** strategy would maintain this window structure but use specialized models at each step rather than reusing the same model. This allows it to benefit from the sequential nature of the recursive approach while incorporating the targeted precision of direct modeling.

The **multiple-output strategy** addresses the challenges of recursive and direct methods by training a **single model** to forecast all future steps simultaneously (one-shot). This approach helps capture the interdependence between future steps and avoids the pitfalls of error accumulation seen in the recursive method. This strategy can be implemented efficiently with modern ML frameworks that support multiple outputs.

In the upcoming *Forecasting with sktime* recipe, you will explore a simpler approach to implementing these different multi-step strategies.

There's more...

In the **multiple-output strategy**, you will create a sliding window function similar to the one you used in the `generate_lagged_features()` function. However, this new function will also take an **output window**. For example, you can specify a 10-sliding window for the **input** and a 10-sliding window for the **output**. The resulting DataFrame will contain features (e.g., $x_1, \dots, x_{10}$) for the independent and target variables (e.g., $y_1, \dots, y_{10}$). This is a one-shot **approach** where you provide a *sequence* as input and get a *sequence* as output.

The `multiple_output_features()` function takes two parameters: `window_in` for the number of input features and `window_out` for the number of output (target) variables:

```
import numpy as np

def multiple_output_features(df, window_in, window_out):
    d = df.values.squeeze()

    x = np.lib.stride_tricks.sliding_window_view(
        d, window_shape=window_in)[-window_out:]
    y = np.lib.stride_tricks.sliding_window_view(
```

```

d[window_in:], window_shape=window_out)

cols_x = [f'x_{i}' for i in range(1, window_in + 1)]
cols_y = [f'y_{i}' for i in range(1, window_out + 1)]

df_xs = pd.DataFrame(
    x, columns=cols_x,
    index=df.index[window_in:len(df) - window_out + 1]
)
df_ys = pd.DataFrame(
    y, columns=cols_y,
    index=df.index[window_in:len(df) - window_out + 1]
)

return pd.concat([df_xs, df_ys], axis=1)

```

A key difference in the `multiple_output_features()` function is using NumPy's `sliding_window_view()` function to efficiently create two separate sliding windows—one for inputs and one for outputs. This allows our models to learn the relationship between a sequence of past values and a sequence of future values, rather than just predicting a single point.

Here is an example of using the function to produce a sequence of 10 input features and 10 target variables:

```

win_in, win_out = 10, 10
air_mo = multiple_output_features(air_cp, win_in, win_out)
dw_mo = multiple_output_features(dw_cp, win_in, win_out)
en_mo = multiple_output_features(en_cp, win_in, win_out)

print(air_mo.columns)
>>
Index(['x_1', 'x_2', 'x_3', 'x_4', 'x_5', 'x_6', 'x_7', 'x_8', 'x_9', 'x_10',
      'y_1', 'y_2', 'y_3', 'y_4', 'y_5', 'y_6', 'y_7', 'y_8', 'y_9', 'y_10'],
      dtype='object')

```

You will use the same objects and functions from the previous recipe, including:

- The `regressors` dictionary, which contains 11 regressors from scikit-learn
- The `split_data()` function
- The `evaluate_results()` function

To support multiple outputs, the training logic is extended into a new helper function, `train_model_mo()`, which is implemented in `utils.py`. The `train_model_mo` utility generalizes the single-output workflow to handle multiple target variables over a forecasting horizon by first splitting the input into feature and target blocks using `win_in` (number of input lags/features) and `win_out` (number of forecast steps/target variables), then standardizing both `X` and `y` with `StandardScaler` before fitting. Because many regressors do not natively support multi-output regression, the function first attempts to fit the base estimator directly and, if that fails, wraps it with `MultiOutputRegressor` to enable multi-target prediction by fitting one regressor per target. Predictions are then inverse-transformed back to the original scale, and evaluation metrics such as RMSE, MAE, MAPE, and MASE are computed using uniform averaging across outputs to summarize overall forecasting performance.

Update the `train_different_models()` function to pass `win_in` and `win_out` into the new `train_model_mo()` function so that each candidate model is trained and evaluated consistently for the specified input window and forecast horizon:

```
from utils import train_model_mo

def train_different_models(train, test, regressors, win_in, win_out):
    results = []
    for reg_name, regressor in regressors.items():
        result = train_model_mo(
            train=train,
            test=test,
            regressor=regressor,
            reg_name=reg_name,
            win_in=win_in,
            win_out=win_out
        )
        results.append(result)
    return results

train_mo_air, test_mo_air = split_data(air_mo, test_split=0.10)
train_mo_en, test_mo_en = split_data(en_mo, test_split=0.10)
train_mo_dw, test_mo_dw = split_data(dw_mo, test_split=0.10)
```


You specified 10 features and 10 target variables based on how you transformed your data earlier using the `multiple_output_features()` function. The following is an example output during model fitting:

```
>>
Training Linear Regression ...
Training Elastic Net ...
Training Ridge Regression ...
Training Lasso Regression ...
Huber Regression does not support multiple outputs
using sklearn MultipleOutput for Huber Regression
Training Random Forest Regression ...
Gradient Boosting Regression does not support multiple outputs
using sklearn MultipleOutput for Gradient Boosting Regression
Training Extra Tree Regression ...
Support Vector Regression (RBF) does not support multiple outputs
using sklearn MultipleOutput for Support Vector Regression (RBF)
Support Vector Regression (linear) does not support multiple outputs
using sklearn MultipleOutput for Support Vector Regression (linear)
Training K-Nearest Neighbor Regression ...
```

Notice that some regressors do not natively support multiple outputs, such as **Huber regression**, **Gradient Boosting**, and **SVR**.

Convert the results into a `DataFrame` and use the `evaluate_results()` function to rank the models:

```
air_results_mo_df = pd.DataFrame(air_results_mo)
air_results_mo_sorted = evaluate_results(air_results_mo_df)
print(air_results_mo_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Huber Regression	25.349827	4.590653	0.816905
1	Support Vector Regression (linear)	25.908003	4.592281	0.821234
2	Ridge Regression	26.316594	4.927290	0.884846

```

en_results_mo_df = pd.DataFrame(en_results_mo)
en_results_mo_sorted = evaluate_results(en_results_mo_df)
print(en_results_mo_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Extra Tree Regression	114.412464	4.625316	0.366718
1	K-Nearest Neighbor Regression	126.100481	4.808844	0.387433
2	Random Forest Regression	122.100304	5.107434	0.399261

```
dw_results_mo_df = pd.DataFrame(dw_results_mo)
dw_results_mo_sorted = evaluate_results(dw_results_mo_df)
print(dw_results_mo_sorted.head(3))
>>
```

	Model Name	RMSE	MAPE	MASE
0	Support Vector Regression (RBF)	6.070763	370.317016	1.310029
1	Random Forest Regression	6.143463	371.615331	1.336797
2	Extra Tree Regression	6.130889	353.583806	1.339550

Finally, you can create a plot to compare actual versus forecast values. The following code is an example plotting **Huber regression** from the air passengers result set:

```
acutal = air_results_mo_df.loc[4]['test'].tail(1).T
forecast = air_results_mo_df.loc[4]['yhat'].tail(1).T
dates = pd.date_range(start='1960-03-31', periods=10, freq='ME')
actual.index = dates
forecast.index = dates
ax = forecast.plot(style='k--o')
actual.plot( ax=ax)
plt.title('Huber Regression for Air Passenger Multiple Output')
plt.legend(['forecast', 'actual'])
plt.show()
```

This should produce the following plot:

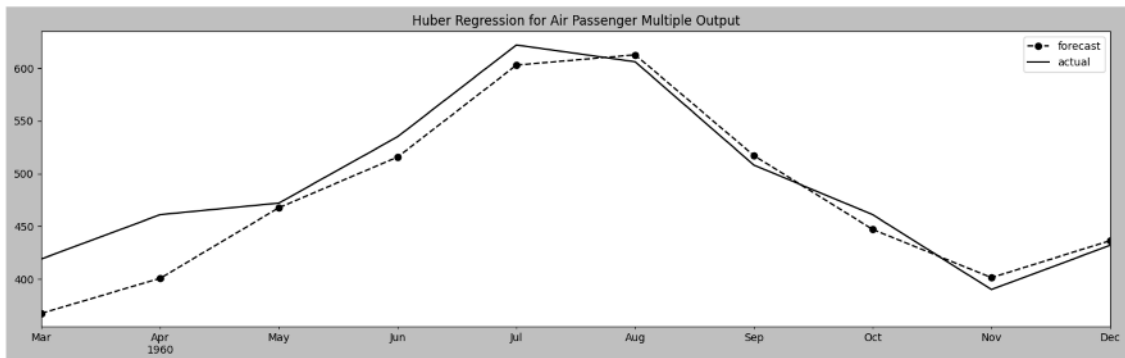


Figure 11.11: Huber regression on the air passengers data for a 10-day forecast using the multiple-output strategy

The plotting code used Huber regression as an example, but you can easily adapt it to visualize predictions from other models by changing the index in `air_results_mo_df.loc[4]` to the corresponding model's position in the results DataFrame. For instance, to visualize linear regression instead, you would identify its index in the DataFrame and replace it in the plotting code (`air_results_mo_df.loc[0]`).

Figure 11.11 demonstrates how the multiple-output strategy performs on the air passengers dataset using Huber regression. Notice how this approach produces forecasts that more closely follow the actual seasonal pattern throughout the 10-month period. Unlike the recursive strategy shown in Figure 11.9, where predictions gradually diverged from actual values (particularly in early months), the multiple-output approach captures both the upward and downward trends accurately. This improved performance illustrates how training a model to predict the entire sequence simultaneously helps maintain consistency across all forecast steps.

By comparing Figure 11.11 with Figure 11.9 for the recursive strategy, you can see how the multiple-output strategy handles future steps more robustly.

See also

- You can refer to *Machine Learning Strategies for Time Series Forecasting* to learn more about multi-step forecasting. Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-36318-4_3.
- To learn more about the `MultiOutputRegressor` class from scikit-learn, you can refer to the official page here: <https://scikit-learn.org/dev/modules/generated/sklearn.multioutput.MultiOutputRegressor.html>

Forecasting with sktime

This recipe introduces you to `sktime`, a powerful Python library explicitly designed for time-series analysis and forecasting. While `scikit-learn` offers robust general ML capabilities that we leveraged in our previous recipes, it lacks native support for time-series forecasting. `sktime` fills this gap by providing a unified framework specifically for time-series tasks, allowing you to seamlessly integrate any `scikit-learn` regressors within a specialized time-series context. The library also provides interfaces to other packages, such as `statsmodels`, `statsforecast`, and `Prophet`.

One of `sktime`'s valuable features is its elegant approach to **multi-step forecasting**, supporting strategies such as *recursive*, *direct*, and *multi-output*, which we manually implemented in the previous recipe. These strategies allow you to extend forecasts across multiple future steps efficiently.

Instead of writing custom code for each approach, `sktime` streamlines the process with its `make_reduction` function. This function effectively translates the time-series forecasting problem into a tabular regression problem, enabling you to leverage any standard `scikit-learn` regressor for time-series forecasting and easily switch between forecasting strategies to compare their effectiveness. Furthermore, `sktime` offers built-in time-series **cross-validation**, and other time-series-specific tools.

The library also excels at creating forecasting pipelines, allowing you to combine preprocessing steps such as detrending or differencing with your forecasting models. The unified interface makes it much easier to experiment with different model architectures, preprocessing techniques, and forecasting strategies.

How to do it...

You will train multiple regressors from `scikit-learn` and `sktime` for multi-step forecasting in this recipe. This will cover data preparation, model training, forecasting, and comparing performance on the three datasets:

1. Start by importing the required libraries:

```
from sklearn.ensemble import (RandomForestRegressor,
                               GradientBoostingRegressor,
                               ExtraTreesRegressor)

from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline as sklearn_make_pipeline
from sktime.forecasting.model_selection import temporal_train_test_split

from sktime.performance_metrics.forecasting import (
    mean_absolute_error as mae,
    mean_absolute_percentage_error as mape,
    mean_absolute_scaled_error as mase,
    mean_squared_error as mse
)

from sktime.forecasting.compose import make_reduction
from sktime.forecasting.base import ForecastingHorizon
```

With the necessary classes and functions loaded, proceed to split the data into train and test sets.

2. Ensure you have the three DataFrames (`air`, `energy`, and `daily_temp`) loaded as described in the Technical requirements section. Make copies of these DataFrames to avoid modifying the originals:

```
air_df = air.copy()
energy_df = energy.copy()
dw_df = daily_temp.copy()
```

Inspect the datasets for missing values. If missing data is found, then perform a simple forward fill.

3. Pass each DataFrame to the `handle_missing_data` function:

```
for df in [energy_df, air_df, dw_df]:
    handle_missing_data(df)
>>
found 2 missing observations...
```

Only the `dw_df` DataFrame had two NaN (missing) values.

4. Split the data into training and test sets using the `temporal_train_test_split()` function from `sktime`:

```
train_air, test_air = temporal_train_test_split(air_df,
                                                test_size=12)
train_en, test_en = temporal_train_test_split(energy_df,
                                                test_size=24)
train_dw, test_dw = temporal_train_test_split(dw_df,
                                                test_size=365)
```

5. The `sktime` library follows a similar paradigm to `scikit-learn`, using the `fit` method to train models and the `predict` method to generate forecasts. As in previous recipes, define your regressors by creating a Python dictionary with the regressor names and classes:

```
regressors = {
    'Linear Regression': LinearRegression,
    'Random Forest Regression': lambda:
        RandomForestRegressor(random_state=42),
    'Gradient Boosting Regression': lambda:
        GradientBoostingRegressor(random_state=42),
    'Extra Tree Regression': lambda:
        ExtraTreesRegressor(random_state=42),
    'Support Vector Regression (RBF)': lambda:
        SVR(kernel='rbf'),
    'Support Vector Regression (linear)': lambda:
        SVR(kernel='linear'),
    'K-Nearest Neighbor Regression': lambda:
        KNeighborsRegressor()
}
```

Setting `random_state=42` for tree-based models ensures reproducibility and consistent results across different runs, enabling fair comparisons between different forecasting approaches.

The regressors dictionary simplifies looping through each model for training and evaluation.

6. You created the `train_model()` and `train_different_models()` functions in previous recipes. Here, you will use the same structure but with some modifications to utilize `sktime`'s time-series forecasting capabilities.

The following `train_model()` function is designed to handle model training and evaluation for each regressor:

```
def train_model(train, test, reg, reg_name, strategy):

    fh = ForecastingHorizon(test.index, is_relative=False)
```

```

sklearn_pipeline = sklearn_make_pipeline(StandardScaler(), reg)

forecaster = make_reduction(estimator=sklearn_pipeline,
                             window_length=10,
                             strategy=strategy)
print(f'training {reg_name} ...')
forecaster.fit(train)
yhat = forecaster.predict(fh)
actual = test.copy()

rmse_test = np.sqrt(mse(actual, yhat))
mae_test = mae(actual, yhat)
mape_test = mape(actual, yhat)
mase_test = mase(actual, yhat, y_train=(train))

model_metadata = {
    'Model Name': reg_name,
    'Model': forecaster,
    'RMSE': rmse_test,
    'MAPE': mape_test,
    'MASE': mase_test,
    'MAE': mae_test,
    'yhat': yhat,
    'actual': actual}

return model_metadata

```

Notice the use of `ForecastingHorizon` and `make_reduction`. The `ForecastingHorizon` defines how far into the future we want to predict based on indices (`DatetimeIndex`) in the test set. Setting `is_relative=False` instructs `ForecastingHorizon` to use the exact dates provided for the forecasting horizon (absolute indices). If `is_relative=True`, then instead of providing dates, you would provide a sequence of integers representing the steps after the end of the training data you want to forecast (relative indices).

The `make_reduction` function adapts scikit-learn regressors for time-series forecasting by transforming the time-series problem into a tabular format. It uses a specified strategy ("recursive", "multioutput", or "direct") to generate forecasts.

The `window_length=10` parameter determines how many past observations are used to make predictions, similar to the lag features we created in previous recipes. In other words, `window_length=10` transforms a time series into a tabular format using a sliding window to create features from the past 10 time steps (lagged features). This value should be tuned based on your specific dataset's characteristics, but 10 provides a reasonable starting point.

We also utilized scikit-learn's `make_pipeline` (aliased as `sklearn_make_pipeline`) to combine `StandardScaler` with the regressor to ensure all features have *mean=0* and *variance=1*. This is particularly important for distance-based algorithms such as SVR and KNN, which are sensitive to the scale of the input features. Scaling ensures that these models perform optimally.

7. For the first round, the **recursive** strategy is chosen, similar to the recursive approach used in previous recipes, where a univariate time series was transformed into a tabular format suitable for multi-step forecasting.

Now, let's define the `train_different_models()` function to loop through each regressor:

```
def train_different_models(train, test, regressors, strategy):
    results = []
    for reg_name, reg in regressors.items():
        results.append(train_model(train,
                                   test,
                                   reg(),
                                   reg_name,
                                   strategy))

    return results
```

The `train_different_models()` and `train_models()` functions have an additional `strategy` parameter used in `make_reduction`. `sktime`'s `make_reduction` can take three strategies:

- "recursive": Forecasts each step using the previous prediction as an input
- "multioutput": Predicts all future steps simultaneously as multiple outputs
- "direct": Creates separate models to predict each future step

You will use the "recursive" strategy for this recipe, which returns a `RecursiveTabularRegressionForecaster` class.

8. Now, pass the training and test DataFrames, the dictionary of regressors, and the "recursive" strategy to the `train_different_models` function. This will run each regressor on the specified datasets and produce multi-step forecasts:

```
air_results = train_different_models(train_air, test_air,
                                     regressors, strategy="recursive")
en_results = train_different_models(train_en, test_en,
                                     regressors, strategy="recursive")
dw_results = train_different_models(train_dw, test_dw,
                                     regressors, strategy="recursive")
```

This will execute multi-step recursive forecasting for each model in the regressors dictionary on the air, energy, and daily temperature datasets.

9. Use the `evaluate_results()` function from the previous recipes to rank the models based on their performance. The function takes your results and a specified metric (e.g., MASE) and returns a DataFrame with models sorted by their score on the selected metric.

The **MASE** is particularly useful for time series because it scales the error relative to the in-sample error of a naïve forecast. This makes it easier to compare models across different datasets, even when the scales of the time series vary significantly.

10. Apply the `evaluate_results` function to rank models by performance for each dataset:

```
air_results_df = pd.DataFrame(air_results)
air_results_sorted = evaluate_results(air_results_df)

en_results_df = pd.DataFrame(en_results)
en_results_sorted = evaluate_results(en_results_df)

dw_results_df = pd.DataFrame(dw_results)
dw_results_sorted = evaluate_results(dw_results_df)
```

Print the top three models for each dataset to identify the best performers:

```
print(air_results_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
              Model Name      MASE
0      Linear Regression  1.580819
1  Gradient Boosting Regression  2.205693
2      Extra Tree Regression  2.425161

print(en_results_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
              Model Name      MASE
0      Extra Tree Regression  0.423111
1  K-Nearest Neighbor Regression  0.423276
2      Linear Regression    0.516344

print(dw_results_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
              Model Name      MASE
0  Support Vector Regression (RBF)  1.777303
1      Linear Regression    1.806715
2  Support Vector Regression (linear)  1.815235
```


The results show that different models perform best for different datasets:

- **Linear regression** worked well for the air passengers data
 - **Extra Tree regression** excelled with energy consumption data
 - **SVR with RBF kernel** was best for the daily temperature data
11. You will use the `plot_results()` function from the previous recipes to visualize the forecasted data and compare it to actual values.
 12. Use the `plot_results` function to visualize **forecast** versus **actual** using the **recursive** strategy from **sktime**:

```
plot_results(air_results_df, 'Air Passengers')
plot_results(en_results_df, 'Energy Consumption')
plot_results(dw_results_df, 'Daily Temperature')
```

The plots will show the actual values versus forecasted values for each model and dataset. This visual comparison helps you evaluate model performance beyond just numerical metrics. For example, you'll be able to see whether a model captures seasonality or trend patterns in the data.

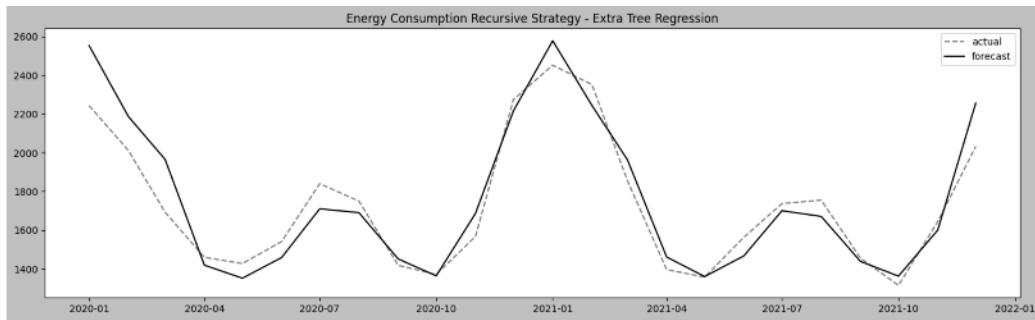


Figure 11.12: Plot of actual versus forecast values using KNN and the recursive strategy for energy consumption data

How it works...

In this recipe, you trained multiple scikit-learn regressors using **sktime**'s `make_reduction` function for efficient multi-step forecasting. This process covered data preparation, model fitting, forecasting, and performance evaluation across three different datasets.

Let's examine the `make_reduction` function since it is a powerful tool in **sktime** that bridges the gap between traditional ML regressors and time-series forecasting. Here is a breakdown of its key parameters:

```
make_reduction(estimator, strategy='recursive',
               window_length=10, scitype='infer')
```

There are four parameters:

- **estimator**: This is the ML estimator or regressor (e.g., `LinearRegression`) that will be transformed for time-series forecasting.

- **strategy:** This defines the forecasting strategy. There are three strategies: `direct`, `recursive` (default), or `multioutput`. The three strategies we mentioned earlier produce different forecaster classes:

```
make_reduction(RandomForestRegressor, strategy='recursive')
>>
RecursiveTabularRegressionForecaster

make_reduction(RandomForestRegressor, strategy='multioutput')
>>
MultioutputTabularRegressionForecaster

make_reduction(RandomForestRegressor, strategy='direct')
>>
DirectTabularRegressionForecaster
```

- **window_length:** This specifies the sliding window length used to generate features from past observations. When selecting `window_length`, consider the seasonality and autocorrelation structure of your time series. A longer window captures more historical patterns but may introduce noise, while a shorter window is more responsive to recent changes.
- **scitype:** This stands for *scientific type*, and in this context, it defaults to `infer`, which automatically chooses between `tabular-regressor` and `time-series-regressor` based on the estimator you provide.
- **make_reduction** simplifies the transformation of the time-series dataset for ML, and based on the chosen strategy, it returns a different forecaster class to be used:

Each returned forecaster class implements a different approach to multi-step forecasting:

- **RecursiveTabularRegressionForecaster:** Fits a single model and uses each prediction as an input to forecast the next. This approach is simple and efficient but can accumulate errors over long forecast horizons.
- **MultioutputTabularRegressionForecaster:** Uses an estimator that predicts all future points at once. If your estimator doesn't natively support multiple outputs, `sktime` will wrap it in `scikit-learn's MultiOutputRegressor`.
- **DirectTabularRegressionForecaster:** Creates a separate model for each forecast step, potentially improving accuracy at the cost of training multiple models.

The results across our three datasets highlight an important lesson in time-series forecasting: no single algorithm dominates across all datasets. The linear regression performed best for air passengers, Extra Tree excelled with energy consumption, and SVR with RBF kernel was superior for daily temperature. This reinforces why having a standardized way to compare multiple models (as provided by `sktime`) is valuable for real-world time-series problems.

There's more...

Let's expand our implementation to support both multi-output and direct strategies. Unlike the recursive approach we used earlier, these strategies handle multi-step forecasting differently:

- **Multi-output strategy:** Forecasts multiple steps in a single model call
- **Direct strategy:** Creates separate models for each forecasting horizon

To implement these strategies, we need to modify our `train_model` function to handle models that don't natively support multi-output forecasting. This is achieved by wrapping such models in scikit-learn's `MultiOutputRegressor` class. Here's the updated function:

To support both multi-output and direct strategies, the `train_model` function needs to get extended to handle models that may not natively support multi-output forecasting. The main idea is to keep the original structure of `train_model()` and `train_different_models()`, but update the internals to work correctly with multi-step strategies.

The `train_model_mo_sktime()` now passes the forecasting horizon `fh` into `fit()` so that reduction-based forecasters can align multi-step predictions with the target index. If the base regressor supports multi-output, it is used directly inside the reduction forecaster. If it does not, the regressor is automatically wrapped with scikit-learn's `MultiOutputRegressor`, which fits one underlying model per forecast step and thus enables multi-output predictions. This wrapped estimator is then passed into `make_reduction`, allowing `strategy='multioutput'` or `strategy='direct'` to be applied transparently.

The `train_different_models_sktime()` function is updated only slightly mainly to call the `train_model_mo_sktime()` instead. To keep the code listing concise, the modified sktime helpers are defined in `utils.py`.

By wrapping these models with the `MultiOutputRegressor` class, we extend their capability to handle multi-step forecasts. We then pass this wrapped regressor to `make_reduction`, allowing it to apply the multi-output strategy.

Testing the multi-output strategy

Now let's apply our updated function with the multi-output strategy. To use the multi-output strategy, pass the training and test DataFrames, the dictionary of regressors, and the multi-output strategy to the `train_different_models` function:

```
from utils import train_different_models_sktime

air_results = train_different_models_sktime(
    train_air, test_air, regressors, strategy="multioutput")
en_results = train_different_models_sktime(
    train_en, test_en, regressors, strategy="multioutput")
dw_results = train_different_models_sktime(
    train_dw, test_dw, regressors, strategy="multioutput")
```

During execution, you'll notice that some models work natively with multi-output forecasting while others require the `MultiOutputRegressor` wrapper:

```
training Linear Regression ...
training Random Forest Regression ...
using MultiOutputRegressor for Gradient Boosting Regression ...
training Extra Tree Regression ...
using MultiOutputRegressor for Support Vector Regression (RBF) ...
using MultiOutputRegressor for Support Vector Regression (linear) ...
training K-Nearest Neighbor Regression ...
```

After training, you can rank models by performance with the same `evaluate_results` function. The following are the top three models for each dataset based on the multioutput strategy:

```
air_results_mo_df = pd.DataFrame(air_results_mo)
air_results_mo_sorted = evaluate_results(air_results_mo_df)
print(air_results_mo_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
           Model Name      MASE
0      Linear Regression  0.800119
1  Support Vector Regression (linear)  1.169109
2      Gradient Boosting Regression  1.654544

en_results_mo_df = pd.DataFrame(en_results_mo)
en_results_mo_sorted = evaluate_results(en_results_mo_df)
print(en_results_mo_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
           Model Name      MASE
0  Random Forest Regression  0.328362
1      Extra Tree Regression  0.344888
2  K-Nearest Neighbor Regression  0.384354

dw_results_mo_df = pd.DataFrame(dw_results_mo)
dw_results_mo_sorted = evaluate_results(dw_results_mo_df)
print(dw_results_mo_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
           Model Name      MASE
0      Extra Tree Regression  1.492585
1  Support Vector Regression (RBF)  1.507306
2      Random Forest Regression  1.608000
```

You can use the `plot_results()` function to visualize predictions against actual data for each dataset. Here is an example showcasing the top-performing model for the daily temperature dataset.

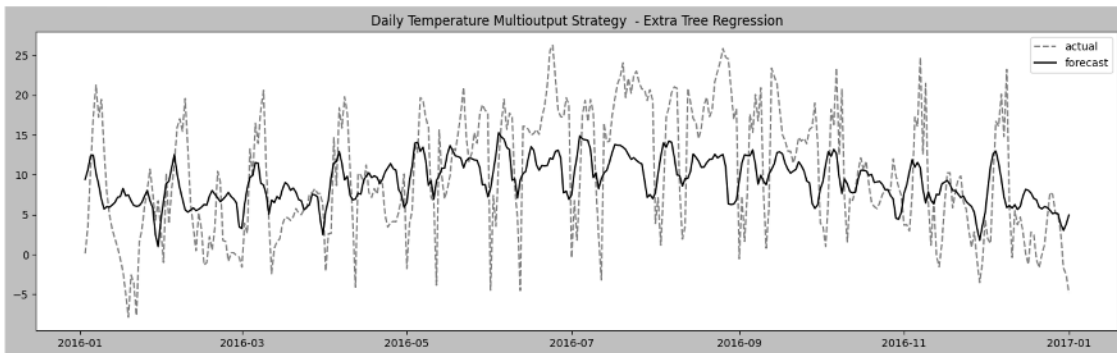


Figure 11.13: Plot of actual versus forecast values using the SVR with RBF kernel and multi-output strategy for daily temperature data

Exploring the direct strategy

Further, using the same functions, you can explore the direct strategy. Unlike the recursive and multi-output approaches, the direct strategy provides a separate model for each forecasting horizon, enabling each model to predict one specific step ahead (without relying on previously predicted values). This approach can sometimes produce more stable predictions across horizons since each model is optimized independently. The drawback is that the direct strategy can be a computationally more expensive and time-consuming process than the recursive and multi-output strategies.

The following is an example using the same function but for the direct strategy:

```
air_results_dir = train_different_models_sktime(train_air, test_air,
                                              regressors, "direct")
air_results_dir_df = pd.DataFrame(air_results_dir)
air_results_dir_sorted = evaluate_results(air_results_dir_df)
print(air_results_dir_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
           Model Name      MASE
0      Linear Regression  0.800119
1  Support Vector Regression (linear)  1.169109
2      Extra Tree Regression  1.43494

en_results_dir_df = pd.DataFrame(en_results_dir)
en_results_dir_sorted = evaluate_results(en_results_dir_df)
print(en_results_dir_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>
           Model Name      MASE
0  Random Forest Regression  0.380342
```

```

1 K-Nearest Neighbor Regression 0.384354
2 Gradient Boosting Regression 0.398006

dw_results_dir_df = pd.DataFrame(dw_results_dir)
dw_results_dir_sorted = evaluate_results(dw_results_dir_df)
print(dw_results_dir_sorted.loc[:, ['Model Name', 'MASE']].head(3))
>>

```

	Model Name	MASE
0	Support Vector Regression (RBF)	1.507306
1	Gradient Boosting Regression	1.630985
2	Extra Tree Regression	1.637348

To visualize predictions against actuals, use the `plot_results` function as before. The following is an example for the *air passengers* dataset:

```
plot_results(air_results_dir_df, 'Air Passengers Direct Strategy')
```

The output displays the forecast compared to the actual values, as shown in *Figure 11.14*.

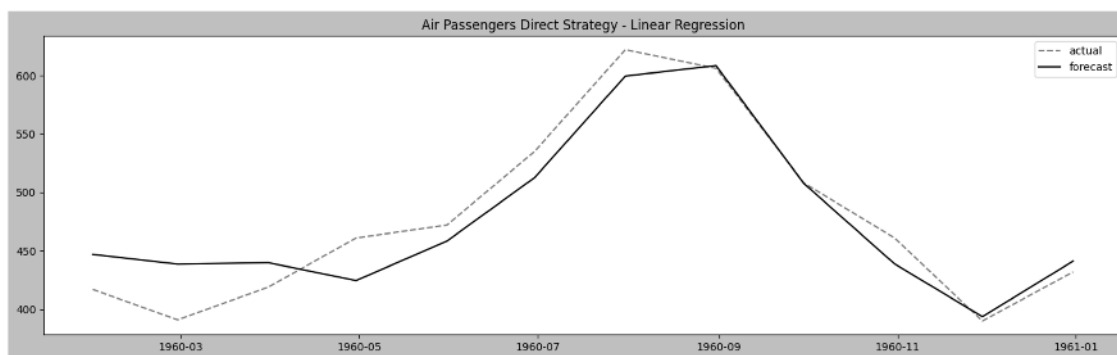


Figure 11.14: Plot of actual versus forecast values using linear regression and the direct strategy for *air passengers* data

Comparing across all three strategies and datasets, we can observe several interesting patterns:

- For air passengers: Multi-output and direct strategies (both with linear regression) significantly outperformed the recursive approach, cutting the error almost in half (MASE of 0.80 versus 1.58).
- For energy consumption: All three strategies performed well, with tree-based models consistently among the top performers. Random Forest achieved the best overall result with the multi-output strategy (MASE 0.33).
- For daily temperature: The SVR with RBF kernel performed consistently well across strategies, while Extra Tree regression showed strong performance with the multi-output approach.

This demonstrates that while the choice of algorithm matters, the forecasting strategy can have an equally important impact on model performance. The multi-output strategy generally produced the best results across our datasets, but this won't necessarily be true for all time-series problems.

For your own forecasting challenges, it's worth experimenting with different combinations of models and strategies to find the optimal approach for your specific data. The `sktime` library makes this experimentation straightforward by providing a unified interface for all these approaches.

See also

In this recipe, you used `sklearn` to run different regression models. What made this possible was using `sktime`'s `TransformedTargetForecaster` class and the `make_reduction` function:

- To learn more about the `MultiOutputRegressor` class from `scikit-learn`, you can refer to the official page here: <https://scikit-learn.org/dev/modules/generated/sklearn.multioutput.MultiOutputRegressor.html>
- To learn more about the `make_reduction` function, you can visit the documentation page here: https://www.sktime.net/en/stable/api_reference/auto_generated/sktime.forecasting.compose.make_reduction.html

Hyperparameter tuning with `sktime`

In previous recipes, you trained different regressors using their default **hyperparameter** values—parameters that define the model's structure and behavior but are not learned from the data (they are supplied before training the model). **Fine-tuning** these hyperparameters can significantly improve model performance by optimizing settings to better fit the data considered, a process known as **hyperparameter tuning**.

In this recipe, you will use `sktime`'s `ForecastingGridSearchCV` to perform hyperparameter tuning through cross-validated **grid search** to find the optimal combination of parameter values. The `sktime` library also provides other hyperparameter tuning options, such as `ForecastingRandomizedSearchCV`, `ForecastingSkoptSearchCV`, and `ForecastingOptunaSearchCV`, offering different approaches to hyperparameter optimization.

Generally, in grid search, we define a grid of hyperparameters—a range of values we want to try for each parameter. Since time-series data is sequential, we need to use temporal cross-validation to ensure we maintain that order. `sktime` provides three methods for this: `SingleWindowSplitter`, `SlidingWindowSplitter`, and `ExpandingWindowSplitter`. These methods preserve the temporal order, dividing the data into folds with either an expanding window or a sliding window approach. `ExpandingWindowSplitter`, for example, progressively increases the training set size while keeping the test set size constant, mimicking how we'd typically forecast in practice.

`ForecastingGridSearchCV` performs an exhaustive search, evaluating every possible combination of hyperparameters across the specified number of folds. This approach can be computationally expensive. For example, having three hyperparameters with three values each would result in $3 \times 3 \times 3 = 27$ combinations. A five-fold cross-validation would mean $27 \times 5 = 135$ total runs (model fits). This is why the number of folds and the size of the parameter grid are key factors impacting the complexity and runtime of grid search. Generally, there is a trade-off between model performance and computational efficiency when performing hyperparameter tuning.



In Chapter 12, you will learn about Optuna for hyperparameter optimization.

Getting ready

As described earlier, **cross-validated grid search** is a process for identifying the optimal hyperparameter values for a model. However, this process can be computationally expensive, especially when working with large datasets, a high number of folds, or a detailed parameter grid. In this exercise, you will work with the same datasets used in previous recipes: air passengers, energy consumption, and daily temperature. Refer to the *Technical requirements* section for details on how to load and pre-process these datasets.

To manage computational complexity and runtime, you will focus on three regressors: **Random Forest**, **KNN**, and **SVR**. Additionally, we will limit the number of folds to five and use a simplified parameter grid. This approach ensures that the grid search remains efficient while still providing meaningful results. You can always expand the parameter grid later if needed.

You will use the datasets prepared in the previous recipe, specifically the `train_air`, `test_air`, `train_en`, `test_en`, `test_dw`, and `train_dw` DataFrames. These datasets are already in the correct format for time-series forecasting, so you can focus on hyperparameter tuning without additional preprocessing.

Additionally, you will utilize the `evaluate_results` and `plot_results` functions from the previous recipe.

How to do it...

1. Assuming you have loaded the libraries from the *Technical requirements* section, as well as those in the previous recipe, you'll need to import additional libraries needed for **grid search** and **cross-validation**:

```
from sktime.forecasting.all import (ForecastingGridSearchCV,  
                                   ExpandingWindowSplitter)
```

2. Create a dictionary of hyperparameters for each regressor. These hyperparameters will be tested in the grid search to identify the optimal set for each model on the specific dataset. The following are examples for the three regressors—RandomForest, KNN, and SVR:

```
# Random Forest parameters  
rf_param_grid = {"n_estimators": [100, 200],  
                 "max_depth": [10, 15],  
                 "min_samples_split": [2, 5, 10]}  
  
# K-Nearest Neighbors parameters  
knn_param_grid = {"n_neighbors": [3, 5, 7, 10],
```



```

        "weights": ['uniform', 'distance'],
        "p": [1, 2]}

# Support Vector Regressor parameters
svr_param_grid = {"kernel": ['linear', 'rbf', 'sigmoid']}

```

3. As you have done in previous recipes, organize the regressors in a dictionary that includes the model name, class, and associated parameter grid. This setup will streamline the hyper-parameter tuning process:

```

regressors = {
    'Random Forest Regressor': [RandomForestRegressor, rf_param_grid],
    'K-Nearest Neighbor Regressor': [KNeighborsRegressor, knn_param_grid],
    'Support Vector Regression': [SVR, svr_param_grid]
}

```

4. You will need to update the `train_model` function from the previous recipe to incorporate a cross-validated grid search. The full code is available on [GitHub](#).

```

...

# Full forecasting horizon for prediction
fh = ForecastingHorizon(test.index, is_relative=False)

# Multi-step horizon for CV, capped at 30 steps
fh_cv = np.arange(1, min(30, len(test) + 1))

...
initial_window = max(50, len(train) // 3)
remaining_length = len(train) - initial_window
step_length = max(1, (remaining_length - len(test)) // 2)

cv = ExpandingWindowSplitter(initial_window=initial_window,
                             step_length=step_length,
                             fh=fh_cv)

try:
    # Wrap the regressor in a MultiOutputRegressor
    m_reg = MultiOutputRegressor(sklearn.pipeline)
    forecaster = make_reduction(
        estimator=m_reg,
        window_length=10, # 10 lags, adjustable
        strategy="multioutput")

```

```
# Perform grid search with cross-validation
gscv = ForecastingGridSearchCV(
    forecaster=forecaster,
    param_grid=param_grid,
    cv=cv, verbose=1)

gscv.fit(train, fh=fh)
...
```

The full implementation is in the `utils.py` function. You will notice that the new function `train_model_sktime_cv()` is set up to perform only a multi-output strategy, as specified in `make_reduction` with `strategy="multioutput"`. The `ExpandingWindowSplitter` is configured with a starting window size and incremental growth to reduce the number of folds, which will be explained further in the *How it works...* section.

`window_length` defines how many past observations the model uses to make a prediction—in the code, it is fixed at `window_length=10`. However, you should explore different values as the optimal choice depends on your dataset's seasonal patterns and inherent time dependencies (e.g., 7 for weekly cycles in daily data, 12 for monthly data). For instance, if your data exhibits strong seasonality, a longer `window_length` might be necessary to capture these patterns, but it could also include irrelevant historical data.

There are also two forecast horizons defined in `train_model_sktime_cv` that serve different purposes: `fh` is used for the final model fitting and prediction and is set to the full test index, while `fh_cv` is used during cross-validation to evaluate model performance. They do not need to be the same value. Currently, `fh_cv` is limited to a maximum of 30 steps to improve computational performance, which is particularly important for larger datasets such as the daily temperature data.

5. The `train_different_models` function is updated to loop through each regressor and parameter grid. You will load this modified version from `utils.py`.

Pass the training and test `DataFrames` along with the dictionary of regressors (and their parameter grids) to the `train_different_models_sktime_cv` function. This will execute a grid search for each regressor on the specified datasets, generating multi-step forecasts:

```
air_results_gs = train_different_models_sktime_cv(
    train_air, test_air, regressors)
en_results_gs = train_different_models_sktime_cv(
    train_en, test_en, regressors)
dw_results_gs = train_different_models_sktime_cv(
    train_dw, test_dw, regressors)
```

You should see an output indicating the grid search progress, similar to the following:

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Grid Search CV using MultiOutputRegressor for Random Forest Regressor ...
Fitting 3 folds for each of 16 candidates, totaling 48 fits
Grid Search CV using MultiOutputRegressor for K-Nearest Neighbor
Regressor ...
Fitting 3 folds for each of 3 candidates, totalling 9 fits
Grid Search CV using MultiOutputRegressor for Support Vector Regression
...
CPU times: user 1.66 s, sys: 92.5 ms, total: 1.75 s
Wall time: 9.6 s
```

Notice the number of folds, parameter combinations, and total fits. For example, in the case of the **Random Forest regressor**, there are 3 folds and 12 combinations of hyperparameters, resulting in a total of 36 fits.

6. Create a `get_best_params()` function. The models fitted through the cross-validated grid search give access to additional attributes, including `best_params_`, `best_forecaster_`, `cv_results_`, and `best_score_`.

The `get_best_params` function will print out the best parameters based on the grid search cross-validation via the `best_params_` attribute:

```
def get_best_params(df):
    for row in df.iterrows():
        print(row[1]['Model Name'])
        print(row[1]['Model'].best_params_)
```

The following is an example run:

```
air_results_gs_df = pd.DataFrame(air_results_gs)
get_best_params(air_results_gs_df)
>>
Random Forest Regressor
{'max_depth': 15, 'min_samples_split': 2, 'n_estimators': 100}
K-Nearest Neighbor Regressor
{'n_neighbors': 3, 'p': 1, 'weights': 'distance'}
Support Vector Regression
{'kernel': 'linear'}

en_results_gs_df = pd.DataFrame(en_results_gs)
get_best_params(en_results_gs_df)
>>
Random Forest Regressor
{'max_depth': 15, 'min_samples_split': 2, 'n_estimators': 100}
K-Nearest Neighbor Regressor
{'n_neighbors': 3, 'p': 2, 'weights': 'distance'}
```

```

Support Vector Regression
{'kernel': 'linear'}

dw_results_gs_df = pd.DataFrame(dw_results_gs)
get_best_params(dw_results_gs_df)
>>
Random Forest Regressor
{'max_depth': 10, 'min_samples_split': 10, 'n_estimators': 100}
K-Nearest Neighbor Regressor
{'n_neighbors': 7, 'p': 1, 'weights': 'uniform'}
Support Vector Regression
{'kernel': 'rbf'}

```

In a Jupyter notebook, scikit-learn and sktime render these composite forecasters as expandable diagrams that clearly expose their nested structure. For example, when inspecting a fitted `ForecastingGridSearchCV`, the top level shows the grid search object, which wraps a `MultiOutputTabularRegressionForecaster`; expanding this reveals a `MultiOutputRegressor`, and expanding that layer shows the underlying scikit-learn **Pipeline** with its steps such as `StandardScaler` followed by the chosen base regressor (for instance, `RandomForestRegressor`).

7. You can use the `evaluate_results` function to rank the models for each dataset based on their performance metrics (e.g., MASE). Here are the results for the three datasets:

```

air_results_gs_sorted = evaluate_results(air_results_gs_df)
print(air_results_gs_sorted.loc[:, ['Model Name', 'MASE']])
>>

```

	Model Name	MASE
0	Support Vector Regression	1.169109
1	Random Forest Regressor	1.638100
2	K-Nearest Neighbor Regressor	2.672546

```

en_results_gs_sorted = evaluate_results(en_results_gs_df)
print(en_results_gs_sorted.loc[:, ['Model Name', 'MASE']])
>>

```

	Model Name	MASE
0	Random Forest Regressor	0.392807
1	K-Nearest Neighbor Regressor	0.480424
2	Support Vector Regression	0.534361

```

dw_results_gs_sorted = evaluate_results(dw_results_gs_df)
print(dw_results_gs_sorted.loc[:, ['Model Name', 'MASE']])

```

```
print(dw_results_gs_sorted)
>>
```

	Model Name	MASE
0	Support Vector Regression	1.507306
1	K-Nearest Neighbor Regressor	1.630997
2	Random Forest Regressor	1.658639

From these results, you can observe how each model performs across different datasets. For example, **SVR** performs best on the air passengers and daily temperature datasets, while the **KNN regressor** outperforms the others on the *energy consumption* dataset.

- Finally, you can visualize the performance of the models by plotting the forecasted values against the actual values using the `plot_results` function. This helps in qualitatively assessing how well the models capture the underlying patterns in the data:

```
plot_results(
    air_results_gs_df,
    'Air Passengers Multioutput Strategy with Grid Search CV')
plot_results(
    en_results_gs_df,
    'Energy Consumption Multioutput Strategy with Grid Search CV')
plot_results(
    dw_results_gs_df,
    'Daily Temperature Multioutput with Grid Search CV')
```

How it works...

The `sktime` library offers three window-splitting options for **cross-validation**, each handling the sequential nature of time-series data differently:

- **SingleWindowSplitter**: Created a single training and test set split without iterating over multiple folds. Useful for quick evaluations or very large datasets.
- **SlidingWindowSplitter**: Slides over the dataset by a fixed step length, keeping the training and test sizes constant at each iteration.
- **ExpandingWindowSplitter**: Expands the training set size with each iteration, allowing the model to learn from an increasing amount of historical data while keeping the test size constant.

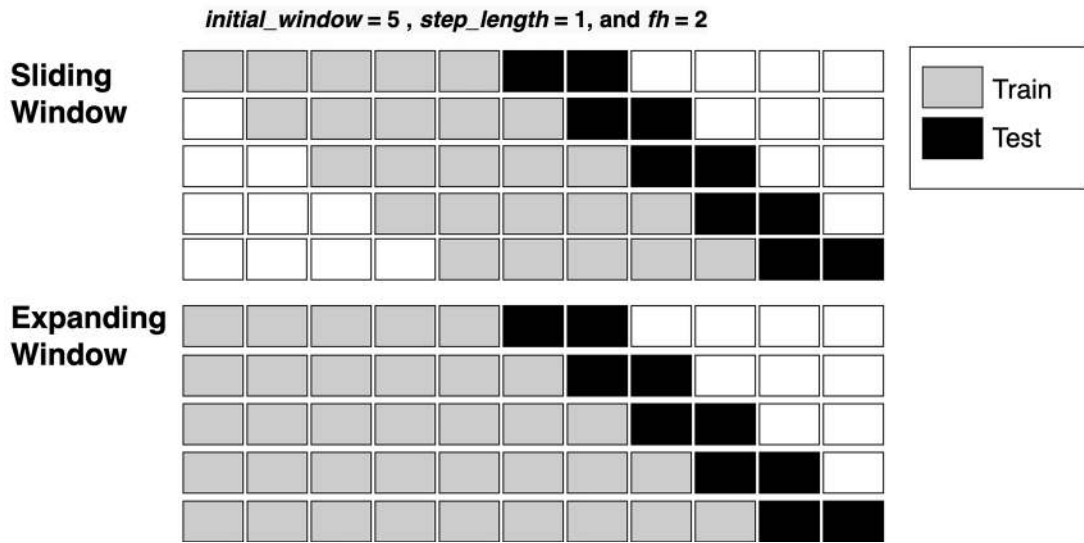


Figure 11.15: Difference between *SlidingWindowSplitter* and *ExpandingWindowSplitter*

Figure 11.15 illustrates the key differences between the `SlidingWindowSplitter` and `ExpandingWindowSplitter`. The `SlidingWindowSplitter` maintains fixed training and test window sizes and simply moves forward by a step length at each iteration. In contrast, `ExpandingWindowSplitter` grows the training set by incorporating more historical data with each fold, offering an expanding context for learning.

Each row in Figure 11.15 represents a model fitted on the designated training set and evaluated on the corresponding test set. Once all folds are processed, the mean score is calculated, providing an estimate of model performance.

In the updated `train_model` function, we used `ExpandingWindowSplitter` with the `initial_window` and `step_length` parameters to define the fold structure:

- `initial_window = max(50, len(train) // 3)`: This parameter sets the starting size of the training window. For `train_en`, the initial training set (fold 1) will contain at least 50 data points or one-third of the training data, whichever is larger. You can adjust the denominator to try a different fraction of training data (e.g., 4 for 25% of the data)
- `step_length = max(1, (len(train) - initial_window - len(test)) // 2)`: This parameter controls the growth of the training window at each fold—the amount by which the training set expands. A smaller `step_length` will result in more folds being created, increasing computational cost, but can provide a more thorough evaluation. You can adjust the denominator to explore a different number of folds (e.g., 4 to produce 5 folds).

Let's examine how these calculations apply to our three datasets:

- For the *air passengers* dataset (132 training points), note the following:
 initial_window would be $\max(50, 132 // 3) = \max(50, 44) = 50$ points
 step_length would be $\max(1, (132 - 50) - 12 // 2) = \max(1, 35) = 35$ points
 Creates three folds: 50, 85, and 120 training points
- For the *energy consumption* dataset (564 training points), note the following:
 initial_window would be $\max(50, 564 // 3) = \max(50, 188) = 188$ points
 step_length would be $\max(1, (564 - 188) - 24 // 2) = \max(1, 176) = 176$ points
 Creates three folds: 188, 364, and 540 training points
- For the **daily temperature** dataset (2,558 training points), note the following:
 initial_window would be $\max(50, 2558 // 3) = \max(50, 852) = 852$ points
 step_length would be $\max(1, (2558 - 852) - 365 // 2) = \max(1, 670) = 670$ points
 Creates three folds: 852, 1,522, and 2,192 training points

The current setup controls the number of folds (three) and the total number of model fits in hyperparameter tuning, balancing computational cost with model evaluation thoroughness.

`ForecastingGridSearchCV` combines cross-validation with an exhaustive grid search over specified hyperparameter values. The total number of fits it needs to complete depends on three factors:

- **Number of folds:** Determined by the `ExpandingWindowSplitter` configuration (e.g., three folds)
- **Hyperparameter combinations:** The product of all hyperparameter values in the grid (e.g., 12 for Random Forest, 16 for KNN, and 3 for SVR)
- **Dataset size:** Larger datasets increase the time required for each fit (e.g., the daily temperature dataset would take the longest time)

For example, using `train_en`, this results in 36 (3 folds $\times$ 12 hyperparameter combinations) fits for Random Forest, 48 (3 folds $\times$ 16 hyperparameter combinations) for KNN, and 9 (3 folds $\times$ 3 hyperparameter combinations) for SVR, completing in 41.9 seconds. While for `train_dw`, the same number of fits took 14 minutes and 30 seconds due to the significantly larger dataset size (2,558 training points versus 564).

Alternative approaches in `sktime`, such as `ForecastingRandomizedSearchCV` instead of `ForecastingGridSearchCV`, or `SlidingWindowSplitter` instead of `ExpandingWindowSplitter`, can significantly reduce computation time while still providing meaningful performance insights, particularly when working with large datasets or extensive parameter spaces.

There's more...

You can further customize the cross-validated grid search by specifying a scoring parameter in `ForecastingGridSearchCV` to control the evaluation metric used during the tuning process. By default, if the scoring parameter is set to `None`, it uses MAPE to assess model performance across folds. However, you can override this behavior to align the tuning with your specific goals.

Other common metrics include MAE and MASE, each with its own strengths depending on your dataset's characteristics:

- **MAE** is less sensitive to outliers compared to MSE, but still scale-dependent
- **MASE** is useful for comparing models across different datasets by scaling errors relative to a naïve forecast

For example, to use MAE instead of MAPE, import the metric and pass it to scoring in two ways.

Option 1: Using the `MeanAbsoluteError` class

The first approach imports and instantiates the `MeanAbsoluteError` class directly:

```
from sktime.performance_metrics.forecasting import MeanAbsoluteError

gscv = ForecastingGridSearchCV(
    forecaster=forecaster,
    param_grid=param_grid,
    cv=cv,
    scoring= MeanAbsoluteError(),
    verbose=1)
```

Option 2: Using the `mean_absolute_error` function

Alternatively, you can import the metric as a function and pass it directly:

```
from sktime.performance_metrics.forecasting import mean_absolute_error as mae

gscv = ForecastingGridSearchCV(
    forecaster=forecaster,
    param_grid=param_grid,
    cv=cv,
    scoring=mae,
    verbose=1)
```

The class-based approach (`MeanAbsoluteError`) provides more flexibility if you need to customize the metric's behavior, while the function-based approach (`mean_absolute_error`) is often more straightforward for simple use cases.

To explore all available metrics in `sktime`, you can use the `registry` function:

```
from sktime.registry import all_estimators
all_estimators("metric", as_dataframe=True)
```

This command returns a `DataFrame` listing all registered metrics in `sktime`, which is useful for discovering additional metrics that might suit your specific needs.

See also

- You can learn more about `ForecastingGridSearchCV` here: https://sktime-backup.readthedocs.io/en/v0.11.4/api_reference/auto_generated/sktime.forecasting.model_selection.ForecastingGridSearchCV.html
- You can learn more about `ExpandingWindowSplitter` here: https://sktime-backup.readthedocs.io/en/v0.11.4/api_reference/auto_generated/sktime.forecasting.model_selection.ExpandingWindowSplitter.html
- You can learn more about `SlidingWindowSplitter` here: https://sktime-backup.readthedocs.io/en/v0.11.4/api_reference/auto_generated/sktime.forecasting.model_selection.SlidingWindowSplitter.html

Forecasting with exogenous variables and ensemble learning in `sktime`

In this recipe, you'll explore two additional techniques: incorporating exogenous variables and ensemble learning with multiple time-series forecasters. The `EnsembleForecaster` class in `sktime` allows you to combine several forecasters, where each forecaster is trained individually, each contributing to making the final prediction. The collective prediction is made by averaging the forecasts from each regressor, which is the default aggregation in `EnsembleForecaster`. Think of this as the power of the collective, or the “wisdom of the crowd,” to improve predictive accuracy. `EnsembleForecaster` supports other aggregations, including median, min, and max.

In this recipe, you will build an ensemble using the same regressors from previous recipes, including **linear regression**, **Random Forest**, **Gradient Boosting**, **KNN**, and **SVM**. You will use the reduction technique explored in earlier recipes, with `make_reduction`, to adapt the regressors for forecasting.

A baseline **naïve regressor** will be used to serve as a benchmark for comparing results against `EnsembleForecaster`. Additionally, you will incorporate **exogenous** variables in the **ensemble model** to create a multivariate time-series forecast. The `sktime` library makes it easy to include exogenous variables with their forecasters. For example, with `make_reduction`, the returned forecaster instance has a `fit()` method, which accepts exogenous variables. Similarly, when using `ForecastingGridSearchCV`, exogenous variables can be easily included in the `fit()` method once the forecaster is instantiated.

Getting ready

In this recipe, you will use the energy consumption (*energy* `DataFrame`) dataset loaded in the *Technical requirements* section of this chapter.

You will need to install the following libraries:

```
pip install holidays
```

In addition to the modules and libraries from the *Technical requirements* section, you'll need the following additional classes and functions for this recipe:

```
from sktime.forecasting.all import (EnsembleForecaster, NaiveForecaster)
from sktime.forecasting.base import ForecastingHorizon
from sktime.utils.plotting import plot_series
from sktime.performance_metrics.forecasting import mean_absolute_scaled_error
as mase
from sktime.split import temporal_train_test_split
from sktime.forecasting.compose import make_reduction
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import (RandomForestRegressor,
                              GradientBoostingRegressor)

from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline as sklearn_make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
```

How to do it...

1. You will use the energy consumption dataset (a univariate time series) and create additional features to use as exogenous variables. The process is called **feature engineering**. This allows you to capture temporal effects in the forecasting model:

```
import holidays

# US holidays (you'd need to adjust for your region)
us_holidays = holidays.US(years=energy_ens.index.year.unique())

# Create a function to check if a month contains a holiday
def contains_holiday(year, month):
    for date, name in sorted(us_holidays.items()):
        if date.year == year and date.month == month:
            return 1
    return 0

month_to_season = {
```

```

1: 'Winter', 2: 'Winter', 12: 'Winter',
3: 'Spring', 4: 'Spring', 5: 'Spring',
6: 'Summer', 7: 'Summer', 8: 'Summer',
9: 'Fall', 10: 'Fall', 11: 'Fall'
}

energy_ens = energy.copy()
energy_ens['month'] = energy_ens.index.month
energy_ens['year'] = energy_ens.index.year
# Cyclical encoding for month
energy_ens['month_sin'] = np.sin(2 * np.pi * energy_ens.index.month / 12)
energy_ens['month_cos'] = np.cos(2 * np.pi * energy_ens.index.month / 12)
energy_ens['season'] = energy_ens['month'].map(month_to_season)
energy_ens['time'] = np.arange(1, len(energy_ens)+1)
energy_ens['holiday'] = energy_ens.apply(lambda row: contains_
    holiday(row['year'], row['month']), axis=1)

```

2. In **sktime**, the reference for the target variables (**endogenous**) is **y**, while **exogenous** variables are represented by **X**. Define the **y** and **exog** (exogenous) variables as shown:

```

y = energy_ens['y']
exog = energy_ens.drop(columns=['y'])
print(y.head())
>>
Month
1973-01-01    1957.641
1973-02-01    1712.143
1973-03-01    1510.079
1973-04-01    1183.421
1973-05-01     1006.326
Freq: MS, Name: y, dtype: float64

print(exog.head())
>>
      month  year  month_sin  month_cos  season  time  holiday
Month
1973-01-01     1  1973    0.500000  8.660254e-01  Winter     1         1
1973-02-01     2  1973    0.866025  5.000000e-01  Winter     2         1
1973-03-01     3  1973    1.000000  6.123234e-17  Winter     3         0
1973-04-01     4  1973    0.866025 -5.000000e-01  Spring     4         0
1973-05-01     5  1973    0.500000 -8.660254e-01  Spring     5         1

```

3. Split the data into training and test sets for both y and exogenous, `exog`, variables:

```
train, test, exog_train, exog_test = temporal_train_test_split(
    y=y, X=exog,
    test_size=30
)
```

4. We need to do some data preprocessing on our exogenous variables since they now contain categorical variables. Our preprocessing pipeline will include **OneHotEncoding** for categorical features and **StandardScaler** for numerical features:

```
# Define categorical and numerical features for exogenous data
exog_categorical_features = ['month', 'season']
exog_numerical_features = ['year', 'month_sin', 'month_cos',
                           'time', 'holiday']

# Create a transformer for exogenous data
exog_transformer = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(drop='first'), exog_categorical_features),
        ('num', StandardScaler(), exog_numerical_features)
    ],
    remainder='passthrough')

# Fit and transform exogenous data
exog_transformer.fit(exog_train)
exog_train_encoded = pd.DataFrame(
    exog_transformer.transform(exog_train),
    index=exog_train.index)
exog_test_encoded = pd.DataFrame(
    exog_transformer.transform(exog_test),
    index=exog_test.index)
```

5. Create a dictionary of the regressors to be used with `EnsembleForecaster`. Each regressor should be converted into a compatible forecaster (as you have done in previous recipes) using `make_reduction` (creating our lagged features) with the 'multioutput' strategy. Note that not all regressors natively support multi-output, so they will need to be wrapped in sklearn's `MultiOutputRegressor`. To streamline the process, define a `create_forecaster` function to transform the time-series data using `make_reduction`:

```
regressors = {
    "Linear Regression": LinearRegression(),
    "Random Forest Regression": RandomForestRegressor(random_state=42),
    "Support Vector Regression (RBF)": SVR(),
```

```

    "K-Nearest Neighbor Regression": KNeighborsRegressor(),
    "Gradient Boosting Regression": GradientBoostingRegressor(
        random_state=42)}

def create_forecaster(
    reg, window_length=12, strategy='multioutput'
):
    if isinstance(
        reg, (SVR, KNeighborsRegressor, GradientBoostingRegressor)
    ):
        mo_reg = MultiOutputRegressor(reg)
        forecaster = make_reduction(
            estimator=mo_reg,
            window_length=window_length,
            strategy=strategy)
    else:
        # Linear, RandomForest support multioutput natively
        forecaster = make_reduction(
            estimator=reg,
            window_length=window_length,
            strategy=strategy)
    return forecaster

updated_regressors = []
for reg_name, regressor in regressors.items():
    reg = create_forecaster(regressor)
    updated_regressors.append((reg_name, reg))

```

- For comparison, you will use `NaiveForecaster` and `RandomForestRegressor` as benchmarks against the performance of `EnsembleForecaster`. The `RandomForestRegressor` will also incorporate **exogenous** variables using a 'recursive' strategy:

```

ensemble = EnsembleForecaster(updated_regressors, aggfunc='mean')
naive = NaiveForecaster()
rf = make_reduction(RandomForestRegressor(), strategy='recursive')

```

- Train all forecasters on the training set with the `fit()` method, providing the univariate time series (`y`) along with the exogenous variables (`X`). For **multi-output**, you will need to define a forecast horizon using `ForecastingHorizon`:

```

fh = ForecastingHorizon(test.index, is_relative=None)
ensemble.fit(y=train, X=exog_train_encoded, fh=fh)
naive.fit(y=train, X=exog_train_encoded)
rf.fit(y=train, X=exog_train_encoded)

```

8. Once training is complete, you can use the `predict()` method with the forecast horizon and the unseen exogenous variables (`exog_test`). You will append the results to a `DataFrame` for easier comparison and plotting:

```
y_hat = pd.DataFrame(test)
y_hat['EnsembleForecaster'] = ensemble.predict(fh=fh, X=exog_test_encoded)
y_hat['NaiveForecaster'] = naive.predict(fh=fh, X=exog_test_encoded)
y_hat['RandomForest'] = rf.predict(fh=fh, X=exog_test_encoded)
print(y_hat.head())
>>
```

	y	EnsembleForecaster	NaiveForecaster	RandomForest
Month				
2019-07-01	1734.089	1666.746604	1434.287	1719.42633
2019-08-01	1677.960	1644.616569	1434.287	1712.40632
2019-09-01	1466.196	1457.223813	1434.287	1457.94170
2019-10-01	1374.674	1413.301493	1434.287	1365.55800
2019-11-01	1809.114	1624.772899	1434.287	1632.89759

9. Calculate and display the MASE scores for each forecaster:

```
print('NaiveForecaster',
      mase(test, y_hat['NaiveForecaster'], y_train=train))
print('EnsembleForecaster',
      mase(test, y_hat['EnsembleForecaster'], y_train=train))
print('RandomForestRegression',
      mase(test, y_hat['RandomForest'], y_train=train))
>>
NaiveForecaster 1.3822824614250524
EnsembleForecaster 0.33278150816939045
RandomForestRegression 0.33694468011991524
```

Overall, both `EnsembleForecaster` and `RandomForestRegressor` performed similarly and significantly better than the **naïve forecaster**.

10. Plot the forecast versus actual values for all three forecasters for visual comparison:

```
styles = ['k--', 'rx-', 'yv-', '-o']
for col, s in zip(y_hat, styles):
    y_hat[col].plot(
        style=s, label=col, title='Comparing the different forecasters')
plt.legend()
plt.show()
```

This should produce what is shown in *Figure 11.16*.

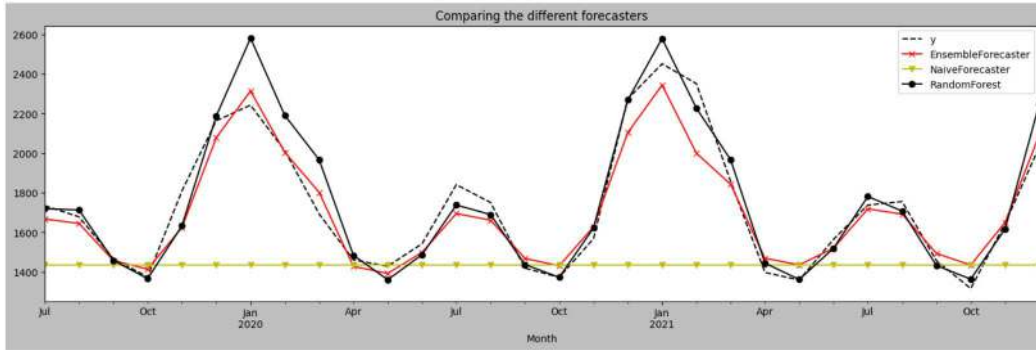


Figure 11.16: Comparing the different forecasters

Remember that none of the models used within `EnsembleForecaster` were optimized. Ideally, you can apply hyperparameter tuning, as demonstrated in the previous recipe, *Hyperparameter tuning with sktime*.

How it works...

The `EnsembleForecaster` class in `sktime` is similar to the `VotingRegressor` class in `sklearn`. Both are ensemble methods designed to fit several regressors and make collective predictions by aggregating their outputs, typically through the mean. However, unlike `VotingRegressor`, `EnsembleForecaster` offers flexibility via the `aggfunc` parameter, allowing you to select different aggregation methods: `mean` (default), `median`, `min`, or `max`.

We use ensemble methods, such as `EnsembleForecaster`, to combine the predictions of diverse models to improve overall accuracy and robustness. While both `EnsembleForecaster` and `VotingRegressor` aggregate predictions from multiple models, `EnsembleForecaster` is specifically designed for time-series forecasting. It integrates seamlessly with `sktime`'s forecasting framework, allowing you to incorporate exogenous variables and handle time-series-specific challenges such as forecast horizons.

When `EnsembleForecaster` is fit, each individual forecaster is trained independently on the same data. During prediction, each forecaster generates its own forecast, and these individual forecasts are then combined according to a specific aggregation function (`aggfunc`). This approach often helps reduce variance and often outperforms any single model. The ensemble works particularly well when the individual forecasters have different strengths and weaknesses, as the error from one model may be offset by the correct predictions from another. In this recipe, we leveraged the power of `EnsembleForecaster` by combining five different regressors: *linear regression*, *Random Forest*, *SVR* (with an RBF kernel), *KNN*, and *Gradient Boosting*. Each of these regressors has different strengths and weaknesses, and by combining them, we aim to create a more well-rounded and accurate model.

To further enhance the model's ability to capture temporal patterns, we performed feature engineering to create our **exogenous** variables.

The exogenous variables added to the dataset (`energy_ens`) include the following:

- **month:** Extracted from the index and treated as categorical to capture yearly seasonality. This allows the model to learn distinct patterns for each month of the year, accounting for monthly variations in energy consumption.
- **year:** Extracted from the index and treated as numerical to capture long-term trends or changes in energy consumption patterns over time. This helps the model detect gradual shifts that might occur across multiple years, such as increasing energy efficiency or growing demand.
- **month_sin** and **month_cos:** Cyclical encodings for the month variable and treated as numerical. Using *sine* and *cosine* transformations preserves the cyclical nature of months (circular relationships between months)—December (12) is as close to January (1) as November (11) is to October (10). This mathematical representation helps the model understand that December and January are adjacent in the annual cycle, avoiding the artificial discontinuity that would occur with simple numeric encoding.
- **season:** Derived from month using `month_to_season` mapping and treated as categorical. This groups months into meaningful seasonal categories (Winter, Spring, Summer, Fall) to capture seasonal effects at a broader scale than individual months. Seasonal patterns in energy consumption are often driven by temperature changes and daylight hours, making this a particularly valuable feature for energy forecasting.
- **time:** A simple sequential counter, representing numbers from 1 to the length of the series and treated as numerical. This is used to capture potential long-term linear trends in the data that might not be fully explained by year alone, such as gradual changes in consumption habits or technology adoption.
- **holiday:** A binary indicator (1 or 0) for months containing a US holiday and treated as numerical. This helps the model account for seasonal variations related to holiday periods.

Feature redundancy and complementarity



While the `month`, `season`, `month_sin`, and `month_cos` variables contain overlapping information, we intentionally keep all of them. Different model types may leverage different representations more effectively—categorical features work well with tree-based models, while cyclical encodings are optimal for linear and neural network models. This redundancy allows the model to choose the most useful representation for capturing temporal patterns in energy consumption.

Unlike previous recipes where `window_length` was not specific to our dataset, in this recipe for our monthly energy consumption dataset, we use `window_length=12` in our `make_reduction` function. This created lagged features based on the previous 12 months to make a prediction and ensure we capture the annual seasonality pattern that is common in energy consumption data.

Before training our models, we create a preprocessing pipeline to handle the mixture of categorical and numerical features:

- **Categorical features encoding:** The `OneHotEncoder` with `drop='first'` transforms categorical variables (*month*, *season*) into binary features while avoiding multicollinearity issues.
- **Numerical feature scaling:** The `StandardScaler` normalizes numerical features (*year*, *month_sin*, *month_cos*, *time*, *holiday*) to have a mean of 0 and a standard deviation of 1, ensuring all features contribute proportionally to model fitting. Standardization is often beneficial for models such as `SVR` and `KNN` that are sensitive to feature scaling.
- **Creating a unified transformer:** The `ColumnTransformer` combines these operations into a single preprocessing pipeline that can be applied consistently to both training and test data:

```
exog_transformer = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(drop='first'), exog_categorical_features),
        ('num', StandardScaler(), exog_numerical_features)
    ],
    remainder='passthrough')
```

The `create_forecaster` function was used to help handle different regressor types. For regressors that don't natively support multi-output (`SVR`, `KNeighborsRegressor`, and `GradientBoostingRegressor`), it wraps them in a `MultiOutputRegressor` before using `make_reduction`.

Note that for any time-series forecast model with exogenous variables, it is assumed that the exogenous variable's future values are known at prediction time and are provided:

```
y_hat['EnsembleForecaster'] = ensemble.predict(fh=fh, X=exog_test_encoded)
```

This aligns with realistic scenarios where factors such as holidays, month, and year are known in advance, but the target variable must be predicted.

When evaluating the different models, we used **MASE**, a scale-independent method for measuring forecasting accuracy that works well across different datasets. Our results were as follows:

```
NaiveForecaster 1.3822824614250524
EnsembleForecaster 0.33278150816939045
RandomForestRegression 0.33694468011991524
```

MASE represents the ratio between the **MAE** of the forecast being evaluated against the MAE of a naïve forecast:

$$MASE = \frac{MAE_{forecast\ model}}{MAE_{naive\ forecast}}$$

The naïve forecast typically uses the previous observation as the prediction for the next time step (for non-seasonal data) or the observation from the previous season (for seasonal data).

Interpreting these values, `NaiveForecaster`'s MASE score of 1.38 indicates that this model performs approximately 38% worse than the in-sample naïve forecasting method used as the benchmark. When MASE exceeds 1, it suggests that the forecasting model produces larger errors than the simple naïve approach did during the training period.

In contrast, both `EnsembleForecaster` (MASE 0.33) and `RandomForestRegression` (MASE 0.34) indicate that the models' errors are roughly 33% and 34% of those produced by the naïve method, in other words, representing 67% and 66% improvement in accuracy over the naïve approach.

The similar performance between `EnsembleForecaster` and `RandomForestRegression` suggests that, in this case, the ensemble doesn't provide substantial additional benefits over the already powerful `RandomForestRegression` model. The `RandomForestRegression` model may already be capturing most of the patterns in the data, leveraging its inherent ensemble properties. However, in other datasets or with different forecasting horizons, the ensemble approach might show greater advantages.

There's more...

In `EnsembleForecaster`, the `weights` parameter allows you to manually control the contribution of each forecaster when aggregating predictions. By default, `weights` is set to `None`, which assigns equal weights (contribution) to each. However, you can provide a list of float values to specify different weights, giving certain forecasters more or less influence on the final prediction.

The `AutoEnsembleForecaster` class in `sktime` behaves similarly to `EnsembleForecaster`, but it automatically computes the optimal weights for each regressor based on a specific method, such as `feature-importance`. This automation can save time and improve performance compared to manually assigning weights, especially when the optimal weights are not known in advance.

The `method` parameter in `AutoEnsembleForecaster` determines how the weights are computed. For example, with `method='feature-importance'`, it calculates weights based on the importance of each forecaster's features.

Using the same list of regressors, you can compare the results of `AutoEnsembleForecaster` with `EnsembleForecaster`:

```
from sktime.forecasting.compose import AutoEnsembleForecaster

auto = AutoEnsembleForecaster(forecasters=updated_regressors,
                              method='feature-importance')
auto.fit(y=train, X=exog_train_encoded, fh=fh)
```

After training, you can retrieve the computed weights using the `weights_` attribute. The weights reflect the relative contribution of each forecaster to the final prediction, with higher weights indicating greater influence. You can loop through the list of regressors and weights to display them, as shown:

```
for reg, w in zip(updated_regressors, auto.weights_):
    print(f'{reg[0]} weight = {w}')
>>
Linear Regression weight = 0.3473247050264662
```

```

Random Forest Regression weight = 0.4693637848158479
Support Vector Regression (RBF) weight = 0.06098912739947967
K-Nearest Neighbor Regression weight = 0.09500546954920448
Gradient Boosting Regression weight = 0.027316913209001766

```

Notice that *Random Forest regression* received the highest weight, indicating it has the most influence on the ensemble's predictions. This suggests that, based on the *feature importances* calculated during training, the Random Forest model's predictions were considered the most reliable for this particular dataset and forecasting horizon.

Once the weights are determined, `AutoEnsembleForecaster` uses them to create a weighted average of the individual forecaster predictions. In `sktime`'s class implementation, you can see this as `np.average(x, weights=self.weights_)` in the `predict` method of the class.

Using the `predict` method, append the forecasts to the same `DataFrame` to compare the results:

```

y_hat['AutoEnsembleForecaster'] = auto.predict(fh=fh, X= exog_test_encoded)
cols = ['y', 'EnsembleForecaster', 'AutoEnsembleForecaster']
print(y_hat[cols].head())
>>

```

	y	EnsembleForecaster	AutoEnsembleForecaster
Month			
2019-07-01	1734.089	1666.746604	1703.337876
2019-08-01	1677.960	1644.616569	1675.229811
2019-09-01	1466.196	1457.223813	1453.412571
2019-10-01	1374.674	1413.301493	1395.770161
2019-11-01	1809.114	1624.772899	1629.203249

You can create a plot to compare the results, as shown:

```

styles = ['k--', 'rx-', 'yv-']
forecasts = y_hat[cols]

for model_name, style in zip(forecasts, styles):
    forecasts[model_name].plot(
        style=style, label=model_name,
        title='EnsembleForecaster vs AutoEnsembleForecaster'
    )
plt.legend()
plt.show()

```

This should produce a plot, as shown in *Figure 11.17*, comparing the actual values (y) with the predictions from `EnsembleForecaster` and `AutoEnsembleForecaster`.

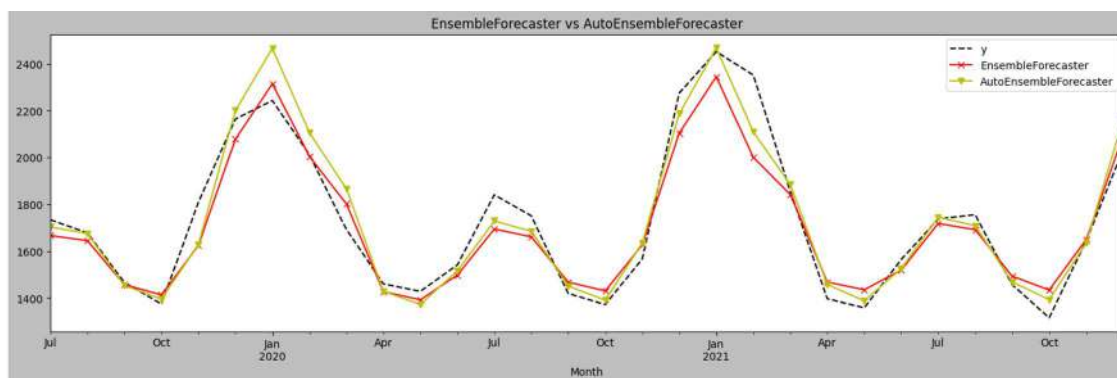


Figure 11.17: Comparing outputs from EnsembleForecaster and AutoEnsembleForecaster

Finally, you can calculate the MASE scores of each forecaster:

```
print('AutoEnsembleForecaster',
      mase(test, forecasts['AutoEnsembleForecaster'], y_train=train))
print('EnsembleForecaster',
      mase(test, forecasts['EnsembleForecaster'], y_train=train))
>>
AutoEnsembleForecaster 0.30637769737992643
EnsembleForecaster 0.32695612676202823
```

AutoEnsembleForecaster performed slightly better, indicating that adjusting the weights can influence the results.

If you prefer more control over the weights, EnsembleForecaster is the better choice, as it allows you to assign specific weights manually. This approach is useful when you have domain knowledge or prior experience, suggesting that certain forecasters should have more influence than others.

See also

- To learn more about the AutoEnsembleForecaster class, you can read the official documentation here: https://www.sktime.net/en/latest/api_reference/auto_generated/sktime.forecasting.compose.AutoEnsembleForecaster.html
- To learn more about the EnsembleForecaster class, you can read the official documentation here: https://www.sktime.net/en/latest/api_reference/auto_generated/sktime.forecasting.compose.EnsembleForecaster.html

Forecasting with XGBoost

XGBoost (which stands for **eXtreme Gradient Boosting**) is a powerful and widely used ML algorithm due to its efficiency, scalability, and high performance in both regression and classification tasks.

In the previous recipe, you were introduced to the `EnsembleForecaster` and `AutoEnsembleForecaster` classes in `sktime` that aggregate the predictions (e.g., weighted or unweighted average) from the different forecasters. This technique is considered a type of **voting ensemble**. You were also introduced to several regressors from `scikit-learn`, such as `RandomForestRegressor`, which is an ensemble algorithm (Random Forest) that creates multiple small models (decision trees) on different bootstrap samples (random sampling with replacement from the dataset), in which each model is trained on a different bootstrap sample. This sampling technique is applied at the row and column (feature) levels. This technique is considered a type of **bagging ensemble** (or **bootstrap aggregation ensemble**). Another `scikit-learn` regressor that you used, and we briefly discussed, is `GradientBoostingRegressor`, which is an algorithm (Gradient Boosting) that builds models sequentially in an additive manner, with each new model specifically designed to correct errors made by the previous ensemble. This technique is called **boosting ensemble**.

In **bagging** and **boosting**, decision trees are commonly used as base learners, but they serve different roles in each approach. In boosting methods such as `XGBoost`, the trees are deliberately kept simple and shallow (“**weak learners**”) to maintain bias while the ensemble iteratively reduces errors. In bagging methods such as Random Forest, fully grown trees are trained on different bootstrap samples, creating diversity that reduces the overall variance when predictions are averaged together. In the *Hyperparameter tuning with sktime* recipe, you used the following:

```
rf_param_grid = {"n_estimators" : [100, 200],
                 "max_depth": [10, 15],
                 "min_samples_split": [2, 5, 10]}
```

`n_estimators` referred to the number of base learners (decision trees), while `max_depth` and `min_samples_split` controlled the complexity of each individual tree.

`XGBoost` is an optimized and highly efficient implementation of the Gradient Boosting algorithm. It incorporates several enhancements over standard Gradient Boosting, including regularization techniques such as **L1 (Lasso)** and **L2 (ridge)** regularization to prevent overfitting. It uses a more sophisticated approach to tree pruning, and it is designed for computational efficiency and scalability, often utilizing parallel processing to speed up training.

When using `XGBoost` for forecasting, you will need to transform your time series into lagged features. We have done this by converting the time series into tabular format manually to be suitable for supervised ML (see the *Preparing time-series data for supervised learning* recipe). You were also introduced to the reduction capability in `sktime` through the `make_reduction` function.

In this recipe, you will be using the `DartsXGBModel` class through `sktime`'s integration with the `Darts` library. As you may recall from *Chapter 9*, `Darts` is a specialized Python library designed for time-series forecasting that provides a collection of modern forecasting models. The `Darts` library has adapted `XGBoost` specifically for time-series forecasting tasks, providing features such as automatic lag creation and handling of time-series-specific requirements. One advantage of using `sktime` is that it creates a unified interface and wraps around many popular libraries, such as *Prophet*, *Darts*, and *statsforecast*, and it integrates with the popular `scikit-learn` library. By integrating with `Darts`, `sktime` allows you to leverage `Darts`' specialized time-series capabilities within `sktime`'s consistent API.

DartsXGBModel eliminates the need for the manual configuration setup required when using sklearn regressors. In previous recipes, when using the `make_reduction` transformation, we had to explicitly specify the forecasting strategy (*recursive*, *direct*, or *multi-output*) and sometimes manually wrap regressors that don't support multi-output using `MultiOutputRegressor`. With `DartsXGBModel`, many of these steps are handled internally.

Getting ready

In this recipe, you will use the energy consumption (*energy* DataFrame) dataset loaded in the *Technical requirements* section of this chapter.

You will need to install the following libraries:

```
pip install darts xgboost
```

In addition to the modules and libraries from the *Technical requirements* section, you'll need the following additional classes and functions for this recipe:

```
from sktime.forecasting.darts import DartsXGBModel
from sktime.forecasting.base import ForecastingHorizon
from sktime.split import temporal_train_test_split
from sktime.performance_metrics.forecasting import mean_absolute_scaled_error
as mase
```

How to do it...

1. Start by making a copy of the energy DataFrame to avoid overwriting the original data, then split it into train and test sets:

```
energy_ens = energy.copy()
# Split the data into train and test sets
y_train, y_test = temporal_train_test_split(energy_ens, test_size=30)
print(f"Training data: {y_train.shape[0]} observations")
print(f"Test data: {y_test.shape[0]} observations")
>>
Training data: 558 observations
Test data: 30 observations
```

2. Using `DartsXGBModel` is straightforward. You will need to create your `ForecastingHorizon` and define the number of lags to create your lagged feature (similar to the `make_reduction` function), followed by model fitting to train on the training set:

```
fh = ForecastingHorizon(y_test.index, is_relative=False)
xgb_model = DartsXGBModel(
    lags=12, output_chunk_length=len(y_test), multi_models=True)
xgb_model.fit(y_train, fh=fh)
```

`output_chunk_length` defines the number of steps the model will predict at once. Recall that in `make_reduction` we had to specify a strategy (*recursive*, *direct*, or *multi-output*). When `multi_models=True`, it resembles a **direct** strategy by creating separate models (XGBoost models) trained for each future step. If `multi_models=False`, then it will create a single model that predicts `output_chunk_length` in one step, similar to the **multi-output** strategy.

3. Use the `predict` method to create your forecast and visualize the output along with the actual test data for comparison:

```
y_pred = xgb_model.predict(fh)
ax = y_test.plot(title='Energy Consumption Forecast with XGBoost',
                 style='k--')
y_pred.plot(ax=ax, style='k-');
```

This should produce the following plot.

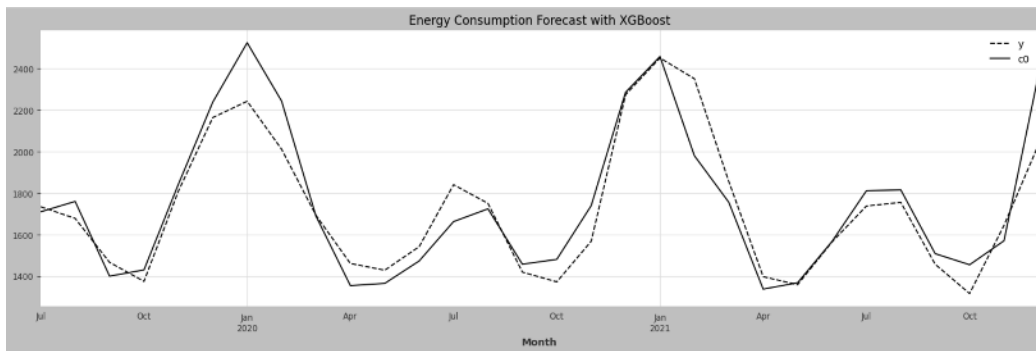


Figure 11.18: Plot of actual versus forecast values using the XGBoost algorithm for monthly energy consumption

4. Compute the MASE score for the XGBoost forecast:

```
# Calculate MASE for XGBoost forecast
mase_score = mase(y_true=y_test, y_pred=y_pred, y_train=y_train)
print(f"MASE: {mase_score}")
>>
MASE: 0.4244915760114761
```

How it works...

The `DartsXGBModel` class in `sktime` integrates the powerful XGBoost algorithm with time-series forecasting capabilities through the `Darts` library. `DartsXGBModel` streamlines the process of using XGBoost for time-series forecasting by handling much of the data transformation and model configuration internally.

When you specify `lags=12`, the model automatically creates 12 lagged features from your time-series data. This transforms your univariate time series into a supervised learning problem where each target value is predicted using the previous 12 observations. This is similar to the manual approach we did in the *One-step forecasting with scikit-learn* recipe and using the `make_reduction` function that we leveraged starting from the *Forecasting with sktime* recipe.

The `output_chunk_length` and `multi_models` parameters determine how the model handles multi-step forecasting. When `multi_models=True` (as in our example), the model uses a direct strategy, creating separate XGBoost models for each future time step. This means that for a 30-step forecast, 30 different XGBoost models are trained, each specializing in a specific forecasting horizon.

During the fitting process (the `.fit()` method), `DartsXGBModel` automatically does the following:

- Transforms the time series into a supervised learning format with lagged features
- Trains the XGBoost model(s) on this transformed data
- Sets up the prediction pipeline for generating forecasts

There's more...

`DartsXGBModel` offers several advanced features that allow you to customize and extend its capabilities for time-series forecasting.

You can create time-based features using the `add_encoders` parameter to automatically create temporal features from our date index without manual engineering. This is similar to the features we created manually in the previous recipe, *Forecasting with exogenous variables and ensemble learning in sktime*.

In this example, we use the following encoders:

- **Cyclic encoding:** Captures cyclical patterns such as months and quarters by transforming them into sine and cosine components
- **Datetime attribute encoding:** Extracts specific attributes such as month and quarter from the datetime index
- **Position encoding:** Adds relative time positions (e.g., the number of time steps since the start of the series)

Define a dictionary of encoders:

```
from darts.dataprocessing.transformers import Scaler

encoders = {
    'cyclic': {
        'past': ['month', 'quarter'],
        'future': ['month', 'quarter']
    },
    'datetime_attribute': {
        'past': ['month', 'quarter'],
        'future': ['month', 'quarter']
    }
}
```



```

    },
    'position': {
        'past': ['relative'],
        'future': ['relative']
    },
    'transformer': Scaler(),
}

```

You can also enable probabilistic forecasting with XGBoost by setting the likelihood parameter to "quantile" and specifying the quantiles you're interested in:

```

# Define the XGBoost model with time series features
xgb_model = DartsXGBModel(
    # Specify lagged values to use as features
    lags=12, # Use the last 12 months as features
    # Enable past covariates by specifying lags
    lags_past_covariates=12, # Use 12 lags of past covariates
    # Enable future covariates by specifying lags
    lags_future_covariates=[0], # [0] means use current future values
    output_chunk_length=len(y_test),
    multi_models=False,

    # Add automatic encoders for time features
    add_encoders=encoders,

    # Enable probabilistic forecasting
    likelihood="quantile",
    quantiles=[0.1, 0.5, 0.9],

    # XGBoost parameters
    kwargs={
        'n_estimators': 100,
        'max_depth': 5,
        'learning_rate': 0.1,
        'subsample': 0.8
    }
)

```

This configuration tells the model to generate predictions for the 10th, 50th (median), and 90th percentiles, allowing you to get prediction intervals rather than just point forecasts.

The `kwargs` parameter in `DartsXGBModel` allows you to pass standard XGBoost hyperparameters directly to the underlying algorithm. This gives you fine-grained control over the model's behavior and performance:

- **n_estimators:** Controls the number of boosting rounds or Gradient-Boosted trees. Higher values can improve model performance but increase training time and may lead to overfitting if not properly regularized.
- **max_depth:** Limits the maximum depth of each tree in the ensemble. Smaller values create simpler trees that are less prone to overfitting but might not capture complex patterns.
- **learning_rate** (also called “eta”): Determines how quickly the model learns by shrinking the contribution of each tree, making the boosting process more conservative. Lower values (0.01–0.3) typically yield better generalization but require more trees to achieve the same performance.
- **subsample:** Implements stochastic Gradient Boosting by using only a fraction of the data in each iteration. Values less than 1.0 introduce randomness that helps prevent overfitting. Setting it to 0.8 means each tree uses only 80% of the available data.

For a full list of parameters, you can check out the official XGBoost documentation here: https://xgboost.readthedocs.io/en/stable/python/python_api.html#module-xgboost.sklearn.

When working with the `add_encoders` dictionary, you need to specify whether each feature should be encoded for past values, future values, or both. For example, when we defined `cyclic`, we used the `past` and `future` keys:

- **past:** These features are created from historical timestamps up to the forecast point
- **future:** These features are created from timestamps in the forecast horizon

In order to use these encoded features, you will need to enable them with the corresponding lag parameters (otherwise, they will be ignored). The `lags_past_covariates` and `lags_future_covariates` parameters are essential when using the `add_encoders` dictionary to enable these time-based features:

- `lags_past_covariates=12` instructs the model to use the previous 12 time steps (lags) of past covariates as features specified under the `past` keys in the encoders dictionary. Without this parameter, any past encoders specified in `add_encoders` would be ignored.
- `lags_future_covariates=[0]` enables the use of features specified under the `future` key in the encoders dictionary, using only the current value of future covariates at each prediction step. Without this parameter, any future encoders in `add_encoders` would be ignored.

You can now fit and evaluate this enhanced model:

```
# Create a forecasting horizon
fh = ForecastingHorizon(y_test.index, is_relative=False)
# Fit the model
xgb_model.fit(y=y_train, fh=fh)
print(xgb_model)
>>
```

```
DartsXGBModel(add_encoders={'cyclic': {
    'future': ['month', 'quarter'], 'past': ['month', 'quarter']},
    'datetime_attribute': {'future': ['month', 'quarter'],
    'past': ['month', 'quarter']},
    'position': {'future': ['relative'],
    'past': ['relative']}},
    kwargs={'learning_rate': 0.1, 'max_depth': 5,
    'n_estimators': 100, 'subsample': 0.8},
    lags=12, lags_future_covariates=[0], lags_past_covariates=12,
    likelihood='quantile', multi_models=False, output_chunk_length=30,
    quantiles=[0.1, 0.5, 0.9])

# Generate forecast
y_pred = xgb_model.predict(fh)

# plot the forecast
ax = y_test.plot(title='Energy Consumption Forecast with XGBoost', style='k--')
y_pred.plot(ax=ax, style='k-');
```

The output shows the forecast compared to the actual energy consumption values, as displayed in Figure 11.19.

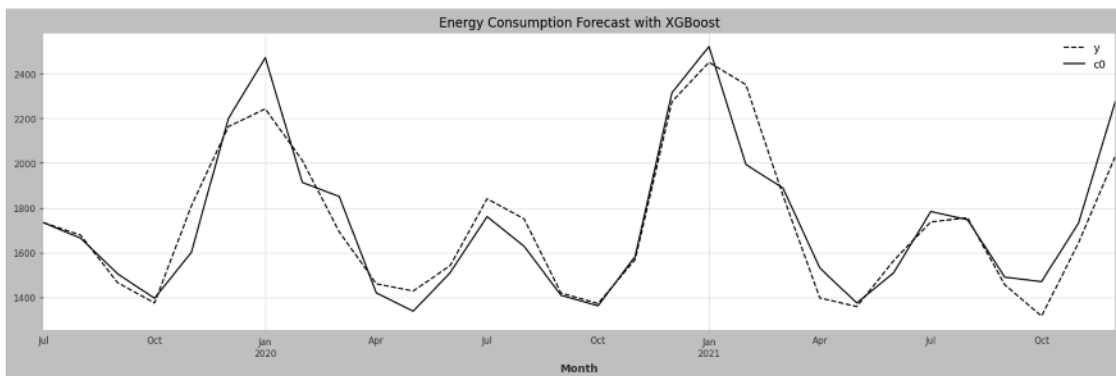


Figure 11.19: Plot of actual versus forecast values using the XGBoost algorithm for monthly energy consumption with additional configurations

Next, calculate the MASE to evaluate the model's performance:

```
# Calculate MASE for XGBoost forecast
mase_score = mase(y_true=y_test, y_pred=y_pred, y_train=y_train)
print(f"MASE: {mase_score}")
>>
MASE: 0.36222420055220445
```

Notice that with these enhancements, our MASE score improved from 0.42 to 0.36, indicating that incorporating time features and optimizing XGBoost parameters can lead to more accurate forecasts. This demonstrates how `DartsXGBModel` provides a powerful yet flexible approach to time-series forecasting, combining the strengths of XGBoost with specialized time-series functionality.

Further, you can generate prediction intervals to visualize the uncertainty in your forecasts:

```
# Generate prediction intervals
pred_intervals = xgb_model.predict_interval(fh, coverage=0.8)

# Plot the results
y_pred.plot()

# Plot prediction intervals
lower = pred_intervals.iloc[:, 0]
upper = pred_intervals.iloc[:, 1]
plt.fill_between(pred_intervals.index,
                 lower,
                 upper,
                 color='red',
                 alpha=0.2,
                 label='80% Prediction Interval')

plt.title('Energy Consumption Forecast with XGBoost')
plt.legend()
plt.tight_layout()
plt.show()
```

The `predict_interval` method generates the lower and upper bounds of a prediction interval with the specified coverage level (80% in this case).

The resulting visualization gives you both the point forecast and a shaded region representing the uncertainty range, which is crucial for making informed decisions based on forecasts.

See also

- The official **XGBoost** documentation offers a great tutorial and educational content for those who want to dive deeper: <https://xgboost.readthedocs.io/en/stable/tutorials/model.html>
- To learn more about Darts' **XGBoost model**, you can visit the official documentation page here: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.xgboost.html
- To learn more about sktime's **DartsXGBModel** wrapper, you can visit the official documentation page here: https://www.sktime.net/en/latest/api_reference/auto_generated/sktime.forecasting.darts.DartsXGBModel.html

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



12

Deep Learning for Time Series Forecasting

In *Chapter 11*, we covered machine learning techniques for time series forecasting. In this chapter, we will focus on **deep learning** (a subset of machine learning), given promising research into its effectiveness, particularly when working with large sequences due to its ability to extract sophisticated patterns with minimal manual intervention. The rapid adoption of deep learning in general, for example, in **computer vision (CV)** and **natural language processing (NLP)**, has been accelerated by techniques such as **transfer learning**, which leverages **pretrained models** to save significant development and training time.

Deep learning adapts well to both structured and unstructured data and can be applied to **supervised**, **unsupervised**, and **semi-supervised** learning tasks. In *Chapter 13*, you will explore using deep learning models in an unsupervised setting for outlier detection. In this chapter, the focus will be on supervised learning techniques.

In this chapter, you will explore how to use deep learning for time series forecasting, focusing on architectures suited for **sequential** data, such as **recurrent neural networks (RNNs)** and their variants, **long short-term memory (LSTM)**, and **gated recurrent unit (GRU)** networks. Other architectures, including **temporal convolutional network (TCN)**, **N-BEATS**, **N-HiTS**, and **temporal fusion transformer (TFT)**, have also shown promise with time series data.

The three most popular deep learning frameworks in Python are **TensorFlow**, **Keras**, and **PyTorch**. Additionally, there are specialized libraries that offer implementations for deep learning architectures tailored for time series forecasting, such as **sktime**, **darts**, **tsai**, **neuralforecast**, and **neuralprophet**.

In *Chapter 11*, we covered how to prepare time series data for machine learning, such as transforming it into a tabular format with lagged features, splitting the data into train and test sets, as well as standardizing (scaling) our data for improved performance. Although deep learning architectures for sequential data, such as LSTM and GRUs, can natively model sequences, they still require windowed input data (e.g., sliding windows of past observations) and scaling for stable training. Deep learning models excel at sequence-to-sequence tasks.

For example, **seq2seq** models take a sequence as input and produce a sequence as output, making them intuitive for multi-step forecasting. Recall in *Chapter 11*, we saw that some machine learning algorithms do not natively support multi-output (for multi-step forecasting), so we had to leverage scikit-learn's `MultiOutputRegressor`. This brings us to the unique advantages these deep learning models give us:

- These architectures are specifically designed to handle sequential data directly, eliminating the need for manual feature engineering that traditional machine learning requires.
- They can capture complex temporal patterns and dependencies that might be missed by simpler models.
- Their native sequence-to-sequence capabilities make multi-step forecasting more intuitive.
- They easily handle multivariate forecasting.
- They can transfer knowledge between related forecasting tasks through transfer learning. Libraries such as `Darts` and `NeuralForecast` have simplified this process.

Deep learning models still require thoughtful consideration when it comes to parameter specifications, making **hyperparameter tuning** an essential task as their performance can be sensitive to how the models are tuned (e.g., *learning rate* or *network depth*). Though powerful, these models come with challenges, including longer training times, higher computational requirements, and potential overfitting with smaller datasets. Throughout the recipes, we will highlight the advantages of these deep learning models, their potential issues, and how they compare to simpler, more interpretable models.

In this chapter, you will explore the following recipes:

- Understanding artificial neural networks
- Forecasting with LSTM (`Keras`, `Darts`, and `NeuralForecast`)
- Handling multivariate time series and exogenous variables
- Hyperparameter tuning for deep learning forecasters
- Forecasting with TCNs
- Forecasting with transformers (`N-HiTS`)

Technical requirements

Creating a new Python environment for this chapter when installing new libraries such as `TensorFlow` or `PyTorch` is highly recommended. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup* recipe from *Chapter 0* on GitHub. The chapter covers two methods: using `conda` and `venv`.

The following libraries will be used throughout the chapter: `TensorFlow`, `Darts`, and `NeuralForecast`. You can install them with `pip`, as follows:

```
pip install tensorflow
pip install darts
pip install neuralforecast
```

When you install `darts`, it will install additional libraries such as `torch`, `pytorch-lightning`, and `statsforecast`, to name a few.

In the first recipe, you will be using the **Monthly Energy Consumption** dataset used in *Chapter 11*.

Start by loading the dataset and libraries that are frequently used throughout:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path
import warnings

plt.rcParams["figure.figsize"] = [14, 7]

warnings.filterwarnings('ignore')
from utils import load_dataset
energy = load_dataset()
```

This should produce a time series plot showcasing monthly energy consumption from January 1973 to December 2021.

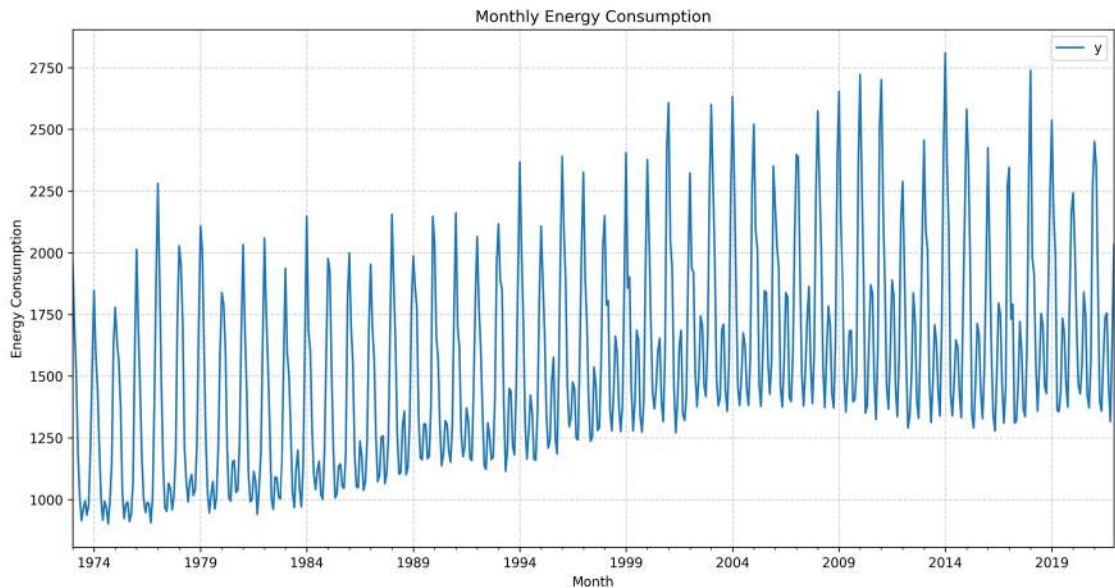


Figure 12.1: Monthly energy consumption plot

Understanding artificial neural networks

Deep learning utilizes **artificial neural networks** (ANNs), which are composed of connected neurons, or nodes. The following diagram represents a shallow neural network (single hidden layer) to highlight its different components:

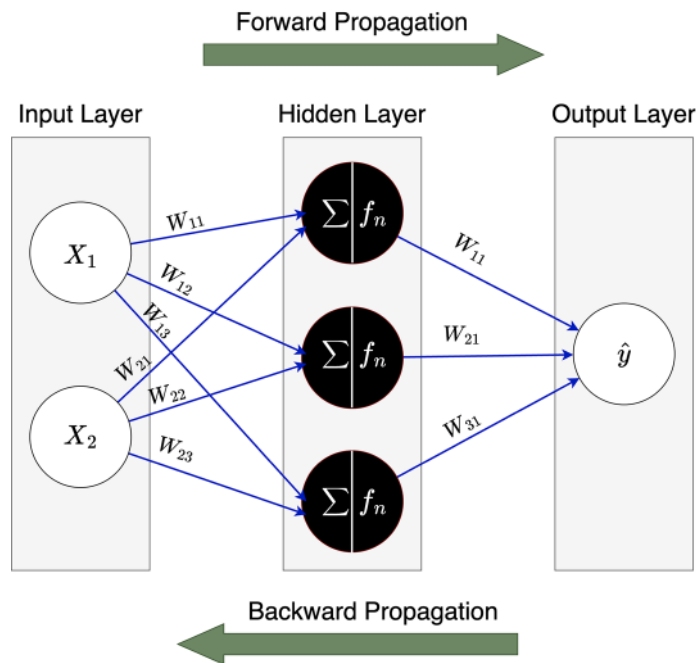


Figure 12.2: A single-layer neural network

A neural network with more than one hidden layer is considered a **deep neural network (DNN)**. While even a single hidden layer can model complex functions, deeper networks often perform better on more challenging tasks because they can learn hierarchical, layered representations of data, capturing more complex relationships. In Figure 12.2, there are three layers – an input layer, a hidden layer, and an output layer.

The hidden layer consists of connected neurons that perform mathematical operations. In its basic form, each neuron performs a linear transformation of the inputs (linear operations such as addition and multiplication). For instance, the first neuron in the hidden layer applies a linear operation such as the following:

$$z = X_1 \times W_{11} + X_2 \times W_{21} + b$$

For multiple neurons in a layer, we would extend this using matrix notation:

$$z = W \cdot X + b$$

Let's break this down:

- W is the weight matrix
- X is the input vector
- b is the bias term
- z is the output of the linear transformation

The notation assumes that the input vector, X , is a column vector. If you use a row vector for X , you will need to transpose the weight matrix, W , accordingly.

However, a purely linear model lacks the flexibility to capture non-linear relationships in data. This is where **activation functions**, such as **Sigmoid**, **rectified linear unit (ReLU)**, or **Tanh**, **come into play**. By **applying an activation function after the linear transformation**, each neuron introduces non-linearity, allowing the network to model more complex patterns. Without these activation functions, even a deep network would act as a linear model, significantly limiting its effectiveness.

Activation functions allow neurons to learn complex relationships by controlling which information gets passed to subsequent layers in deeper networks. For example, after a linear transformation, the ReLU function would transform it as follows:

$$A = \text{ReLU}(Z) = \max(0, z)$$

This non-linearity allows neural networks to capture intricate patterns in data beyond simple linear relationships.

Neurons use different activation functions depending on their role:

- Hidden layers often use non-linear activation functions such as Sigmoid, ReLU, or Tanh
- The output layer's activation function depends on the task:
 - **Sigmoid** is commonly used for binary or multi-label classification tasks, as it outputs probabilities independently for each class
 - **Softmax** is used in multiclass classification to ensure probabilities sum to one across classes
 - **Linear** activation typically used for regression tasks (such as forecasting)

Training a network can be described in two main steps: **forward propagation**, when the data is sent through the layers (input all the way to output layers) to make predictions using current weights ($W_{11}, W_{12}, \dots$) and biases (b), and **backward propagation** to adjust the weights ($W_{11}, W_{12}, \dots$) and biases (b) by comparing the predictions to actuals, guided by the **loss function** that measures errors. A full pass is considered when the entire dataset passes through the network once in a forward direction (forward propagation) and then again in a backward direction (backward propagation). This cycle is called an **epoch**, where each epoch represents one complete pass through the training data. The goal of backward propagation is to update the weights to minimize a loss function. For time series forecasting, common loss functions may include **mean squared error (MSE)**, **mean absolute error (MAE)**, and **mean absolute scaled error (MASE)**.

The network weights ($W_{11}, W_{12}, \dots$) are the parameters that the model estimates (learns) through the training process. These weights are typically initialized randomly at the start of the training process and gradually optimized through backpropagation to reduce error (minimize the loss function). This iterative process helps the model improve by learning from mistakes. Additionally, the model includes a bias term (b) associated with each neuron, which allows shifting activations independently of the input data.

When training a deep learning network, you will need to set several hyperparameters that control how the model learns, including the following:

- **Learning rate:** Controls the step size for updating the weights. A higher learning rate can speed up training but risks overshooting the optimal solution, while a lower learning rate ensures more precise updates but may slow down convergence or even getting stuck in suboptimal solutions (local minima).
- **Number of epochs:** Determines the total number of complete passes (forward and backward propagation) through the entire dataset.
- **Batch size:** Defines the number of training samples (data points) processed together through the network before updating the model's weights (parameters). Smaller batch sizes provide more frequent updates, lower memory usage, and faster iteration but can introduce noise and make convergence challenging. Larger batch sizes, on the other hand, can lead to more stable and accurate updates but require more memory and may result in slower convergence (due to fewer updates per epoch).
- **Number of layers and nodes:** Defines the neural network architecture. Additional layers (deeper networks) allow for learning more complex patterns, while more nodes in each layer enhances the network's ability to learn complicated relationships between inputs and outputs. However, overly complex networks can overfit (fail to generalize well), require more computing power, and take longer to train.
- **Loss function:** Measures how well the model's predictions match actual outcome (quantifying the difference between predictions and actual value). The loss function guides the optimization process by providing feedback that helps the model improve on each iteration.
- **Activation function:** Introduces non-linearity into neural networks, allowing them to learn complex patterns in data.

These hyperparameters significantly influence how efficiently the network learns and converges toward optimal solutions (i.e., optimal network weights).

In this chapter, as you build different neural network architectures for time series forecasting, you'll gain practical experience with these hyperparameters and observe how they impact the model's learning process and overall performance.

Figure 12.3 shows variants of RNNs, starting from **SimpleRNN** and moving to more advanced variants such as GRU, LSTM, and **deep LSTM** (which stacks multiple LSTM layers).

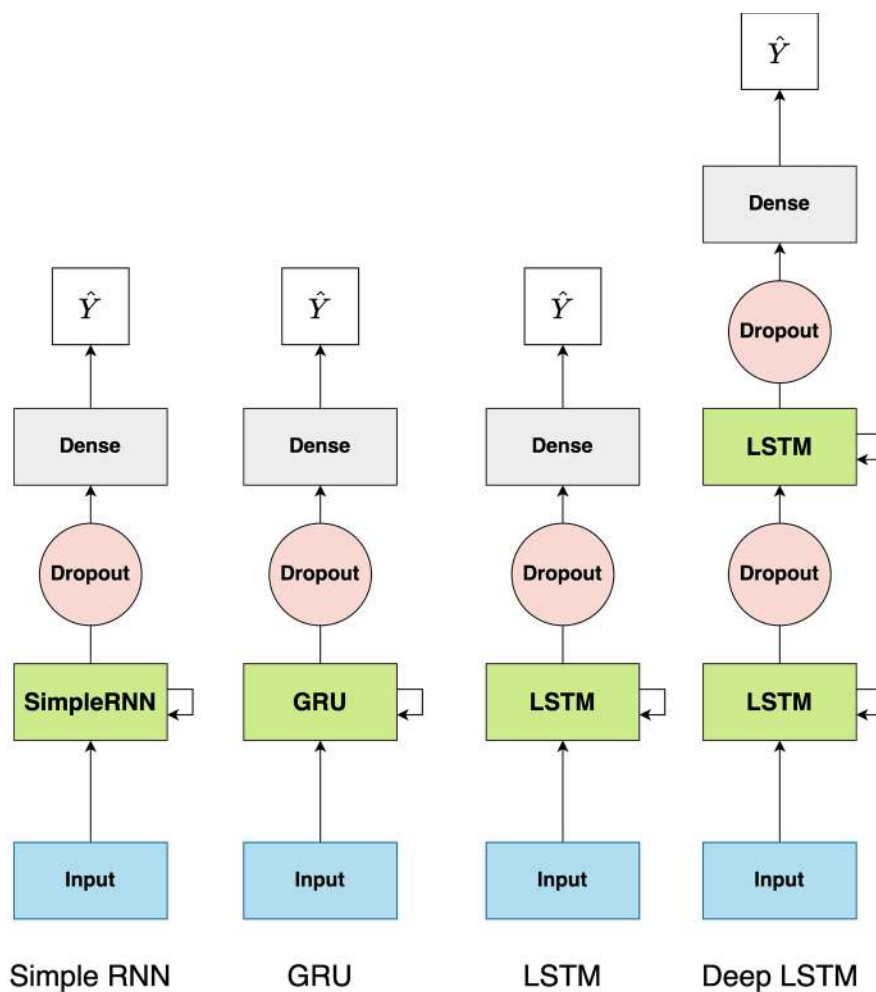


Figure 12.3: RNN architecture and its variants for sequential data

Each architecture processes sequential data, passing information from one time step to the next (shown by the arrows). You'll also notice **Dense** layers (to combine features for the final prediction) and **Dropout** layers (to prevent overfitting by randomly dropping connections during training).

In the context of RNNs, LSTM, and GRUs, the **Tanh (hyperbolic tangent)** activation function is most commonly used:

$$A = \tanh(Z)$$

You will explore LSTM in more depth in the next recipe, but for now, observe from *Figure 12.5* that both LSTM and GRU use **Tanh** and **Sigmoid** activation functions internally.

While RNNs and their variants have been traditional choices for sequential data, **transformer architectures** have revolutionized sequence modeling since their introduction in 2017, based on the paper *Attention Is All You Need*, which you can read here: <https://arxiv.org/abs/1706.03762>.

It's hard to escape the discussion around transformers in the age of NLP and **generative AI**, and more specifically, when talking about **large language models (LLMs)**. What makes transformers interesting is that, unlike RNNs, which process data sequentially, transformers use attention mechanisms to process all elements of a sequence simultaneously. This ability for parallel processing enables transformers to capture both short- and long-range dependencies more effectively. The self-attention mechanism is the core innovation of transformers, allowing the model to weigh the importance of different positions in the input sequence when producing output for a specific position. This helps transformers handle longer sequences more effectively than RNNs, which often struggle with capturing dependencies across distant time steps.

There has been a lot of research and interest around leveraging transformers beyond language models to be used in time series forecasting. Several specialized transformer architectures have been proposed, such as TFT, **Informer**, and **MultiPatchFormer**. These models maintain the attention mechanism's strengths while incorporating adaptations specific to temporal data, such as multi-resolution processing that captures patterns at different time scales simultaneously. Other deep learning models, such as **Neural Basis Expansion Analysis Time Series (N-BEATS)** and **Neural Hierarchical Interpolation for Time Series (N-HiTS)**, leverage deep neural architectures but are not transformer-based.

Forecasting with LSTM (Keras, Darts, and NeuralForecast)

Sequential data refers to datasets where data points are ordered in a specific sequence, and there is a dependency between the data points. Examples of sequential data includes spoken language, audio signals, gene sequences, stock prices, sales data, and weather measurements. Time series data is a specific type of sequential data where observations are recorded over time, in order, and at consistent intervals (e.g., hourly, daily, or monthly). When working with sequential data, where the order and timing of observations matters, certain deep learning architectures are better suited for this task, such as RNNs and their variants, such as LSTM and GRU.

RNNs gained popularity in the field of NLP due to their ability to handle sequential data, where past observation (such as words) strongly influences the next word in a sentence. This need for the neural network to retain memory (**hidden state**) inspired the RNN architecture. Similarly, in time series forecasting, past observations influence future observations, and the network needs some form of memory to keep track of these events.

Unlike **feed-forward networks (FFNs)**, RNNs include a feedback loop where the output from one step is fed back (the recursive part) to the network as input at the next step. This recursive structure (feedback loop) allows the network to learn from prior time steps and enables an RNN to retain short-term memory. As shown in *Figure 12.4*, the RNN cell is unfolded across time steps (five in this example), illustrating how outputs are recursively passed forward.

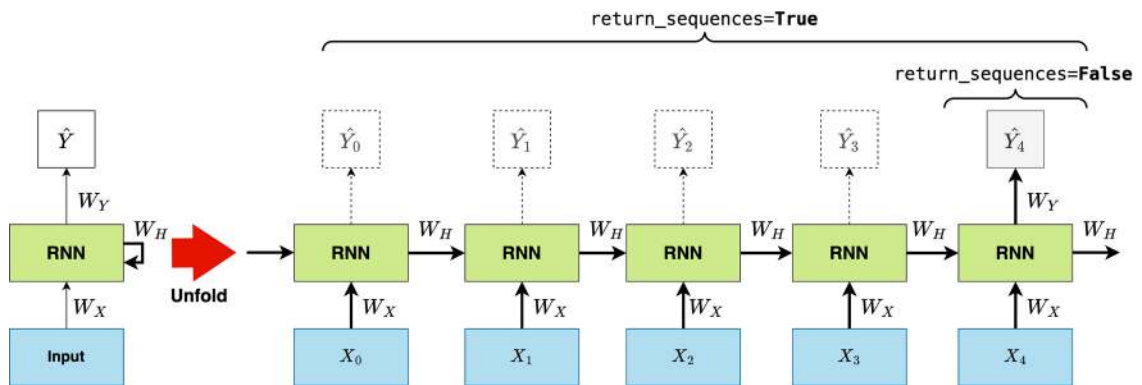


Figure 12.4: Unfolding an RNN

In Figure 12.4, unlike an FNN, an RNN has two sets of weights and outputs that help it retain memory over sequences: W_X is the input weight applied to the current input, X_t , and

W_H is the recurrent weight applied to the hidden state from previous step.

Despite their ability to process sequential data, basic RNNs have limitations when dealing with longer sequences that demand extended memory retention, as they lack the mechanisms for long-term memory. More specifically, RNNs suffer from the **vanishing gradient problem** – during backpropagation through time, as the error signal travels backward through multiple time steps (effectively through many hidden layers in a deep network), the gradients tends to become exponentially smaller with each step backward. This makes it difficult to adjust the weights of earlier layers in long sequences, reducing its ability to learn long-range dependencies. As a result, the output becomes more influenced by the closer layers (recent nodes) with minimal influence of much earlier sequences. In other words, any memory of earlier layers decays through time, hence the term vanishing.

This can be problematic if you have a very long sequence (such as a long paragraph) and you want to predict the next word based on overall context that relate to distant past words. In time series data, how critical long-term memory is will depend on your goal and the domain knowledge about the underlying process of the time series data.

If long-term memory is needed, then LSTM networks offer a more robust solution to standard RNNs. LSTMs address the vanishing gradient problem by introducing memory cells and gates that control the flow of information. These gates include the **input gate** (controlling what new information enters the memory), **forget gate** (determining what information to discard), **output gate** (controlling what information flows to the output), and what's sometimes called the **input modulation** or **cell input gate** (creating candidate values for updating the cell state). This gating mechanism allows LSTMs to be selective about which relevant information to retain over longer periods while discarding irrelevant information. This allows LSTMs to capture both short-term and long-term dependencies effectively. The key innovation with LSTMs is their ability to be selective, regulate memory, and determine what information should be added or removed at each step.

Another alternative is GRUs, which were also proposed to solve the vanishing gradient problem by introducing gating mechanism similar to those in LSTMs but with a simpler architecture (only two gates – **update** and **reset**). These gates are mathematical functions that act as filters to ensure that only relevant pieces of information are retained (determine which information to be retained and which to be discarded). Generally, GRUs tend to train faster than LSTMs as they have fewer parameters. However, LSTMs may outperform GRUs on tasks with highly complex or very long-range dependencies, though in practice, the difference depends on the dataset. This is due to the more complex gating and memory mechanisms in LSTMs. *Figure 12.5* shows RNN, LSTM, and GRU cells. A key feature of LSTMs shown in *Figure 12.5* is the dual-state mechanism consisting of the cell state (C_t) and hidden state (H_t).

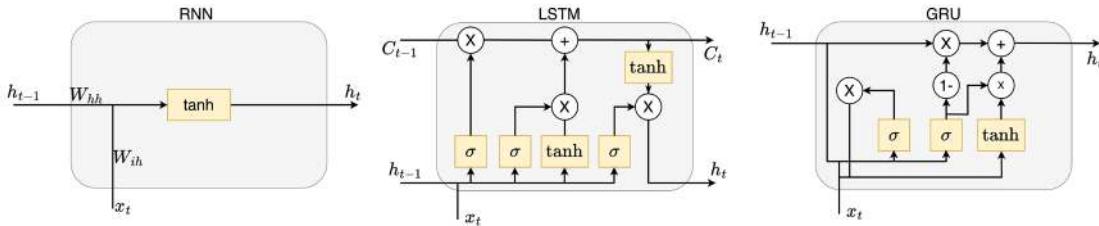


Figure 12.5: RNN, LSTM, and GRU cells

The cell state acts as the long-term memory, flowing relatively unchanged through the entire sequence and protected by the gates. This allows information to be preserved across many time steps. The hidden state, on the other hand, serves as short-term memory and the output at each time step, containing information relevant to the current context. This separation between long-term memory (cell state) and working memory (hidden state) is what enables LSTMs to maintain information over extended sequences, making them particularly valuable for time series with long-range dependencies, such as annual seasonality in monthly data.

Notice from *Figure 12.5* that an RNN cell mainly uses a Tanh activation function, while both LSTM and GRU use Tanh and sigmoid activation functions internally. Tanh is generally preferred because it outputs values between -1 and 1, helping to stabilize gradients during training.

An important parameter when using **Keras** RNN, GRU, or LSTM layers is `return_sequences`, which determines the shape of the output (whether the model produces a forecast for each time step or only at the final time step of the input sequence). This is further illustrated in *Figure 12.4*:

- `return_sequences=True`: The layer outputs a prediction, Y_t , at each time step, t . This is useful for multi-step forecasting where you want the model to predict a value for every time step in the output window. An example is sequence-to-sequence forecasting, where the model takes a sequence as input and outputs a sequence of future values. A practical application would be using the previous 30 days of energy consumption to predict consumption for each of the next 7 days, enabling planning for weekly energy demand.
- `return_sequences=False`: The layer outputs only the final prediction, Y , at the end of the sequence. This is suitable for **one-step forecasting**, where you use a sequence of past observations as input to predict the next value. An example is using the previous 30 days of stock prices to predict only tomorrow's closing price.

In PyTorch, the RNN, GRU, and LSTM layers always return the full sequence of outputs by default. To obtain only the final output (as with `return_sequences=False` in Keras), you would manually select the last time step from the output tensor.

Getting ready

You should already have the energy DataFrame available from the *Technical requirements* section. In this recipe you will train an LSTM model using three different libraries: Keras, Darts (PyTorch), and NeuralForecast (PyTorch). There are common preprocessing steps between the three approaches, such as scaling and splitting the data into train, test, and validation sets. You will create some utility functions and classes to prepare our data to use in the recipe.

To handle missing data, you can use a simple fill forward strategy. For more advanced techniques for handling missing data you can refer to *Chapter 6*. The `fill_missing_forward` function takes a DataFrame, and if there are missing values, it will use the `ffill` method.

Use the function to ensure we handle any missing data:

```
from utils import fill_missing_forward
en_df = fill_missing_forward(energy)
```

How to do it...

In this recipe, you will build and train an LSTM model using three different frameworks: Keras (with a TensorFlow backend), Darts, and NeuralForecast (both Darts and NeuralForecast use a PyTorch backend). The aim is to align configurations as closely as possible while acknowledging each framework's unique characteristics. The goal is to configure these models as similarly as possible, though you'll discover that each library has its own approach to implementation (some are more straightforward than others).

We'll implement a consistent setup across all three frameworks, including the following:

- **Splitting the data:** We will split the data into training, validation, and test sets. The test set is our final hold-out set for scoring our models using RMSE and MAE.
- **Applying scaling:** We will standardize the data before feeding it to the models, a crucial step for neural network performance.
- **Adding early stopping:** We will implement a mechanism so that when the validation loss stops improving after a certain number of epochs (our **patience**), the training will automatically halt to prevent overfitting.
- **Enabling checkpointing:** We will save the best version of our model, the one with the lowest validation loss, so we can retrieve and use it for forecasting.
- **Implementing learning rate schedulers:** To improve performance and speed up convergence, we will adjust the learning rate during training instead of keeping it static. This is where we will see a key philosophical difference between the libraries. We will explore **reactive schedulers** such as `ReduceLROnPlateau` (validation score plateau), which cuts the learning rate when performance stalls, and compare that to **proactive schedulers** such as `StepLR` (step-based decay), which follows a predefined schedule.



While we aim for identical configurations, each framework has unique characteristics. For instance, NeuralForecast uses step-based training rather than epoch-based, and includes an MLP decoder by default, which creates architectural differences that impact direct performance comparisons. These differences will be highlighted as we progress through each implementation (See *Figure 12.11*).

Using Keras: one-step forecasting

Keras offers three methods for defining models:

- **Sequential API:** This is easiest to use and simple to implement as you stack layers one at a time (one input, one output, no branching)
- **Functional API:** This allows for building more complex architectures (multiple inputs, multiple outputs, and branching)
- **Model subclassing:** You subclass the `keras.Model` class to implement your own custom class that inherits from the base class, `keras.Model`

In this recipe, you will work with the Sequential API:

1. Start by importing the necessary library for the Keras recipe:

```
from tensorflow.keras import Sequential, Input
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import (EarlyStopping,
                                       ReduceLROnPlateau,
                                       TensorBoard,
                                       ModelCheckpoint)

from tensorflow.keras.metrics import RootMeanSquaredError,
MeanAbsoluteError
```

2. In *Chapter 11*, when preparing the time series data for supervised machine learning, you learned how to transform the time series data into a tabular format with a lagged feature. In deep learning, conceptually, you will perform something similar by creating input sequences through windowing sequences (implicit lagged feature representation). This transformation is needed when working with Keras. Later, when working with Darts and NeuralForecast, this transformation will be handled behind the scenes. Import the `create_sequences` utility function, which takes a `DataFrame` and a `window_size` parameter to create these sequential windows:

```
from utils import create_sequences
```

3. Transform the `en_df` `DataFrame` into a supervised learning format suitable for LSTM modeling. Set `window_size=12` to use the previous 12 months (1 year) as input features for predicting the next month (one-step ahead forecasting):

```
en_seq = create_sequences(en_df, 12)
en_seq.shape
>>
(576, 13)
```

4. The resulting DataFrame has 13 columns: 12 lagged features labeled (x1 through x12), and the target variable, y. For example, the first row in `en_seq` uses months 1 to 12 to predict month 13. The second row uses months 2 to 13 to predict month 14, preserving chronological order.



Note that using `window_size=12` will reduce your original dataset by 12 months (since the first 12 months can't be used as full input windows). The original `en_df` DataFrame had 588 months, but now `en_seq` has 576 months.

5. You need to scale and split your data into *train*, *validation*, and *test* sets. For this recipe, instead of using a library, import the `TimeSeriesStandardScaler` utility class, which provides methods for splitting, scaling transformation, and inverse transformation (similar to `StandardScaler` from `scikit-learn`):

```
from utils import TimeSeriesStandardScaler
```

This custom scaler is designed specifically for time series data. It maintains temporal order during splitting and fits the scaling parameters exclusively on the training set to prevent data leakage.



Note: The `TimeSeriesStandardScaler` works for univariate time series where features are lagged values of the target variable (like our windowed sequences), allowing X and y to be scaled together using the same parameters. For multivariate cases with exogenous variables (external predictors with different units or distributions), you should scale features and target separately to preserve their distinct statistical properties.

6. Instantiate the `TimeSeriesStandardScaler` class to split and standardize your datasets:

```
scaler = TimeSeriesStandardScaler(en_seq, test_size=12, val_size=36)
k_train_scaled, k_val_scaled, k_test_scaled = scaler.fit_transform()
print(f'''
    original: {energy.shape}
    sequence_df: {en_seq.shape}
    train_scaled: {k_train_scaled.shape}
    val_scaled: {k_val_scaled.shape}
    test_scaled: {k_test_scaled.shape}
''')
```

```
>>
original: (588, 1)
sequence_df: (576, 13)
train_scaled: (528, 13)
val_scaled: (36, 13)
test_scaled: (12, 13)
```

7. The `train_scaled`, `val_scaled`, and `test_scaled` datasets were created as DataFrames, but Keras models require input as NumPy arrays. The `prepare_keras_input` function converts each DataFrames into NumPy arrays and splits the target (y) from the features, returning them as separate arrays ready for LSTM training:

```
def prepare_keras_input(*args):
    y = [col.pop('y').values.reshape(-1, 1) for col in args]
    x = [col.values.reshape(*col.shape, 1) for col in args]

    return *y, *x
```

8. Use the `prepare_keras_input` function by passing the three DataFrames:

```
(y_train_k,
 y_val_k,
 y_test_k,
 x_train_k,
 x_val_k,
 x_test_k) = prepare_keras_input(k_train_scaled.copy(),
                                k_val_scaled.copy(),
                                k_test_scaled.copy())

print(f"x_train_k shape: {x_train_k.shape}")
print(f"y_train_k shape: {y_train_k.shape}")

>>
x_train_k shape: (528, 12, 1)
y_train_k shape: (528, 1)
```

9. A simple LSTM model architecture will consist of an Input layer, an LSTM layer, and a Dense layer for the forecast output. Using the Sequential API, the implementation would look like this:

```
model = Sequential()
model.add(Input(shape=input_shape))
model.add(LSTM(units=units))
model.add(Dense(1))
```

10. `Dense(1)` is used for one-step forecasting (a single neuron to produce a single value for predicting one time step ahead). To predict multiple future steps at once, you can change the output layer to `Dense(n_steps)`, where `n_steps` is the number of future steps you want to predict.

11. To avoid model overfitting, a good practice is to add a Dropout layer for regularization:

```
model = Sequential()
model.add(Input(shape=input_shape))
model.add(LSTM(units=units))
model.add(Dropout(dropout_rate))
model.add(Dense(1))
```

12. The Sequential API allows you to stack layers using the add() method. In our implementation, we will add another LSTM and Dropout layer:

```
model = Sequential()
model.add(Input(shape=input_shape))
model.add(LSTM(units=units, return_sequences=True))
model.add(Dropout(dropout_rate))
model.add(LSTM(units=units//2)) # Second LSTM layer with fewer units
model.add(Dropout(dropout_rate))
model.add(Dense(1))
```

13. When stacking LSTM layers (e.g., feeding the output of one LSTM layer into another), it's important that the preceding LSTM layer returns the full sequence of outputs (all hidden states) for each time step instead of just the final state. This allows the subsequent LSTM layer to have the full sequence to process. This is achieved by setting the return_sequences=True parameter on that LSTM layer. Without this parameter, only the final hidden state would be passed to the next layer. This is illustrated in *Figure 12.4*.
14. It is a common practice to reduce the number of units in deeper layers, often creating a 'funnel' shape. Notice that the second LSTM in our stack uses fewer units (units//2). This approach helps the model capture hierarchical features while reducing the total number of parameters, thus lowering both model complexity and the risk of overfitting.
15. Putting it all together, you will create create_keras_lstm_model, which takes a few key parameters such as input_shape (the structure of the input data), units (this base value determines the number of neurons/cells in our LSTM layers), and dropout_rate (fraction of neurons to drop for regularization):

```
def create_keras_lstm_model(input_shape, units=32, dropout_rate=0.2):
    model = Sequential()
    # Input shape expected by LSTM: (batch_size, timesteps, features)
    # We provide (timesteps, features) via input_shape
    model.add(Input(shape=input_shape))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units//2)) # Second LSTM layer with fewer units
    model.add(Dropout(dropout_rate))
    model.add(Dense(1)) # Output layer for forecasting
```

```

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='mean_squared_error',
    metrics=[RootMeanSquaredError(name='rmse'),
             MeanAbsoluteError(name='mae')])

return model

```

16. The function will return a compiled Keras model that you can fit on your data. The `compile()` method configures the model for training after you've defined its architecture (the layers using the Sequential API). The `compile()` method takes three key components required for training:
- optimizer:** This is the algorithm to use to update the model weights. Here, we use the Adam optimizer.
 - loss:** This is the loss function that the model tries to minimize during training. We use MSE, which is commonly used for regression tasks.
 - metrics:** This takes a list of metrics used to evaluate the model's performance during training and testing. These metrics (RMSE and MAE) are displayed for each epoch for monitoring, to understand how the model is doing overall, but they do not affect the training process, unlike the loss function.

In a nutshell, the `compile()` method instructs Keras how to train the model: how to optimize, how to measure success, and which algorithms to use for updating the weights.

17. You are now ready to create your model instance to use for fitting on your data:

```
keras_input_shape = (x_train_k.shape[1], x_train_k.shape[2])
```

18. Our input shape should be (timesteps, features), where timesteps represents the number of time steps in each sequence (determined by our window size, which is 10), and features is the number of variables per time step. Since we are working with a univariate series, features is just 1. So, our input shape here is (10, 1):

```

keras_lstm_model = create_keras_lstm_model(
    input_shape=keras_input_shape,
    units=32,
    dropout_rate=0.2
)
keras_lstm_model.summary()

```

19. This should produce the following output:

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 12, 32)	4,352
dropout (Dropout)	(None, 12, 32)	0
lstm_1 (LSTM)	(None, 16)	3,136
dropout_1 (Dropout)	(None, 16)	0
dense (Dense)	(None, 1)	17

Total params: 7,505 (29.32 KB)
Trainable params: 7,505 (29.32 KB)
Non-trainable params: 0 (0.00 B)

Figure 12.6: LSTM model architecture summary with two LSTM layers

20. Figure 12.6 shows a two-layer LSTM network with 7,505 trainable parameters. The model contains two LSTM layers (32 units and 16 units), each followed by dropout regularization, and the network ends with a dense output layer for single-step forecasting.
21. With the `keras_lstm_model` object, you are now ready to fit the model on your data. You will create a `train_keras_model` function that takes the Keras model, train/validation sets, and accepts a few parameters. Inside the function, you will create three callbacks (EarlyStopping, ReduceLROnPlateau, and TensorBoard) for visualization and inspection:

```
def train_keras_model(
    model, x_train, y_train, x_val, y_val,
    epochs=500,
    patience=15,
    batch_size=32
):
    """
    Trains a Keras model with early stopping and learning rate reduction.

    """
    print(f"\nTraining Keras model for max {epochs} epochs"
          f"(patience={patience}, batch_size={batch_size})...")
    early_stopping = EarlyStopping(
        monitor="val_loss",
        patience=patience,
        min_delta=0.001, # Small improvement threshold
        restore_best_weights=True, # Restore weights from best epoch
        verbose=1
    )
```

```

# persisting model best weights
model_checkpoint = ModelCheckpoint(
    filepath='./logs/best_model.keras',
    monitor="val_loss",
    save_best_only=True,
    mode="min",
    verbose=1,
    save_freq="epoch"
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=patience//3,
    min_lr=1e-6,
    verbose=1
)

tensorboard= TensorBoard(log_dir='./logs', histogram_freq=1)

history = model.fit(
    x_train, y_train,
    shuffle=False, # Important for time series
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(x_val, y_val),
    callbacks=[early_stopping, reduce_lr,
               tensorboard, model_checkpoint],
    verbose=1
) # Set to 1 or 2 for progress, 0 for silent
print("Keras training complete.")
return history

```

The EarlyStopping callback saves unnecessary compute time if the model is not improving. The patience value, an integer, is a threshold representing the number of epochs with no improvements before stopping the training. Similarly, ReduceLROnPlateau will reduce the learning rate when there is no improvement.

22. In `model.fit()`, we used `shuffle=False` because, with time series data, we need to preserve the temporal order. The `batch_size` parameter (set to 32 in our example) controls how many samples are processed before the model weights are updated. Smaller batches (16–32) often work well with noisy data with more frequent weight updates. For time series with strong seasonal patterns, ensure that the batch size isn't so small that the model can't see enough of the pattern to learn effectively. For instance, if you have strong seasonal patterns that span across many time steps, you'll want to ensure that `batch_size`, in conjunction with `window_size`, allows the model to see a representative portion of these patterns within each batch to learn them effectively. Too small a batch might only show incomplete parts of a longer cycle, making it harder for the model to pick up on those important structures.
23. Use `train_keras_model` with `patience=15`, `epochs=500`, and `batch_size=32`:

```
history_lstm_keras = train_keras_model(
    keras_lstm_model,
    x_train_k, y_train_k,
    x_val_k, y_val_k,
    epochs=500,
    patience=15,
    batch_size=32)
```

24. The training will run for a max of 500 epochs. In this example, since `verbose=1` in `EarlyStopping`, `ReduceLROnPlateau`, and `model.fit()`, you will see updates for each epoch showing loss scores, any reduction in learning rate from `ReduceLROnPlateau`, and a message on early stopping once a plateau has been reached. The `restore_best_weights=True` parameter ensures that the weights of the best epoch are preserved and not the last epoch.
25. You can launch TensorBoard to visualize and understand the model's training process, including metric tracking, using the following magic commands inside JupyterLab:

```
%load_ext tensorboard
%tensorboard --logdir=./logs
```

This instructs TensorBoard to read from the `./logs` folder location. The following interactive screen should launch.



Note that TensorBoard logs from all training runs will accumulate in the `./logs` directory. If you run training multiple times, TensorBoard will display all runs together. To manage distinct runs, either clear this directory between training sessions or use timestamped subdirectories (e.g., `./logs/run_20231215_143022`) for better organization.

26. Evaluate the model's performance on the test set by comparing predictions against actual values. Use the `plot_forecast` utility function, which generates two plots: training/validation loss curves and actual vs. forecasted values on the test set.

```
from utils import plot_forecast
plot_forecast(keras_lstm_model,
              x_test_k,
              y_test_k,
              k_test_scaled.index,
              history_lstm_keras, scaler)
```

Since we used `restore_best_weights=True` in `EarlyStopping`, the model automatically uses the best weights from training (lowest validation loss), not the weights from the final epoch.

Remember, this LSTM model is trained for one-step-ahead forecasting; for each input sequence (the previous 12 months), it predicts only the next time step (the next month).

27. The output displays two visualizations: the first shows model loss per epoch (training and validation loss), similar to TensorBoard output, allowing you to verify the model converged without overfitting. The second plot compares the forecast against actual test set values, with MAE and RMSE metrics printed below. See *Figure 12.7*:

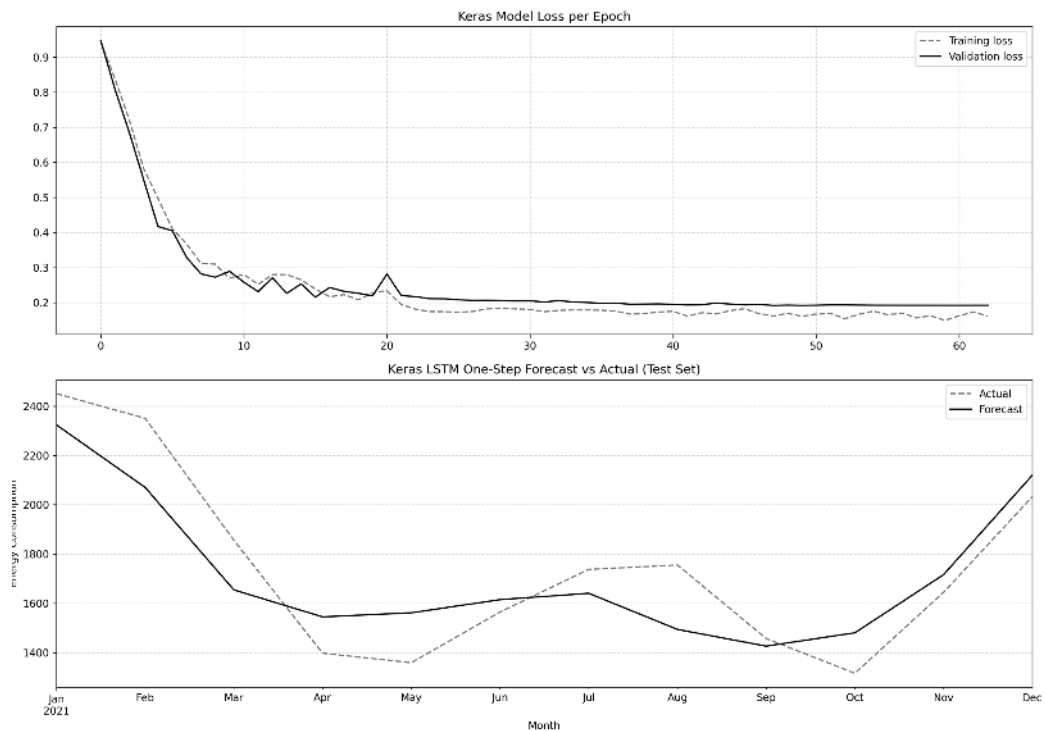


Figure 12.7: Keras LSTM model performance using the `plot_forecast` function

28. In our Keras LSTM implementation, when you see a plot of 12 predicted months (*Figure 12.7*) versus the test set, remember the following:
- Each prediction is made independently, using only the actual historical data available up to that point (not using previously predicted values as input).
 - This is different from recursive (iterative) forecasting, where the model's own predictions are fed back as input to predict further into the future. You will see how easy it is to implement recursive or multi-output using Darts and NeuralForecast.
29. The `plot_forecast()` function will also print out the MAE and RMSE scores:

```
Keras Test Set MAE: 143.5149
Keras Test Set RMSE: 163.1068
```

Using the Darts RNNModel class (one-step forecast)

We were introduced to the Darts library in *Chapter 9* and *Chapter 11*. Darts is a versatile library for time series forecasting and goes beyond just offering statistical time series models. It also offers advanced deep learning implementations. In this recipe, you will explore the `RNNModel` class, which provides three variants of RNNs: vanilla RNN, LSTM, and GRU through the `model` parameter. Darts uses **PyTorch Lightning** for its `RNNModel` implementation. PyTorch Lightning is a high-level, open source wrapper built on top of **PyTorch**. Let's get started:

1. Start by importing the required libraries:

```
from darts.models import RNNModel
from darts import TimeSeries
from darts.dataprocessing.transformers import Scaler
from darts.metrics import rmse, mae
from pytorch_lightning.callbacks import EarlyStopping
import torch
```

2. Make a copy of the `en_df` dataset. You will perform similar preprocessing steps as we did in the Keras recipe, which includes splitting and scaling the data. This time, we will do them using different techniques as a way to show alternative options:

```
en_cp = en_df.copy().reset_index()
ts = TimeSeries.from_dataframe(en_cp,
                              time_col='Month',
                              value_cols='y',
                              freq='MS')

ts = ts.astype(np.float32)

n_total = len(ts)
n_test = 12
n_val = 36
```

```

# Calculate split indices
test_start = n_total - n_test
val_start = test_start - n_val

# Create splits maintaining temporal order
d_train = ts[:val_start]
d_val = ts[val_start:test_start]
d_test = ts[test_start:]

print(f'''
train: {len(d_train)}
val : {len(d_val)}
test: {len(d_test)}''')
>>
train: 540
val : 36
test: 12

```

3. You will scale using `StandardScaler` from `scikit-learn` and wrap it with the `Scaler` class from `Darts`:

```

from sklearn.preprocessing import StandardScaler
d_scaler = Scaler(StandardScaler())

d_train_scaled = d_scaler.fit_transform(d_train)
d_val_scaled = d_scaler.transform(d_val)
d_test_scaled = d_scaler.transform(d_test)
d_series = d_scaler.transform(ts)

```

Note that if you not provide a scaler to the `Scaler` class, it would default to `MinMaxScaler` from `scikit-learn`.

4. Let's define some of the parameters that are needed for `RNNModel` as a Python dictionary:

```

# Darts LSTM parameters
darts_lstm_params = {
    'model': 'LSTM',          # Specify the RNN type
    'input_chunk_length': 12, # How many time steps the model sees
                              # (your 'window')
    'training_length': 13,    # Length of input and output (12+1)
    'hidden_dim': 32,         # LSTM units
    'n_rnn_layers': 2,        # Number of LSTM layers
    'dropout': 0.2,           # Dropout rate
}

```

```

'batch_size': 32, # Batch size for training
'n_epochs': 500, # Max number of epochs
'optimizer_kwargs': {'lr': 0.001}, # Adam is default
'random_state': 42, # For reproducibility
'force_reset': True # Ensure fresh model training
}

```

5. Define `make_early_stopper` function and `make_trainer_kwargs` using `EarlyStopping` from PyTorch Lightning with a similar setup to the Keras implementation:

```

PATIENCE=15
def make_early_stopper(patience=15):
    return EarlyStopping(
        monitor="val_loss", # Darts/PyTorch Lightning
                           # default validation loss metric name
        patience=patience, # Early stopping patience
        min_delta=0.001, # Minimum change to qualify as an improvement
        mode="min", # We want to minimize loss
        verbose=True,
    )

def make_trainer_kwargs(patience=15):
    early_stopper = make_early_stopper(patience)
    return {
        "callbacks": [early_stopper],
        "enable_progress_bar": True, # Show training progress bar
        "accelerator": "auto",
    }

```

6. Define `make_lr_scheduler_kwargs` to reduce the learning rate when there is no improvement. You will use `ReduceLROnPlateau` from `torch` and use a similar configuration as you did in the Keras implementation:

```

def make_lr_scheduler_kwargs(patience=15):
    return {
        "mode": "min",
        "monitor": "val_loss",
        "factor": 0.2,
        "patience": patience // 3,
        "min_lr": 1e-6,
    }

```

7. Putting it all together, let's define RNNModel:

```

trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)
darts_lstm_model = RNNModel(
    **darts_lstm_params, # unpack our parameters dictionary
    model_name='energy_lstm',
    log_tensorboard=True,      # Set True to use TensorBoard
    save_checkpoints=True,     # Set True to save model checkpoints
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs=lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs # Pass trainer args including callbacks
)

```

8. We configured `lr_scheduler_cls` and `lr_scheduler_kwargs` to mirror the same `ReduceLROnPlateau` functionality we used in our Keras implementation, leveraging `lr_scheduler` from PyTorch under the hood.
9. Now, you are ready to fit the model on the training data:

```

darts_lstm_model.fit(train_scaled,
                     val_series=val_scaled,
                     verbose=True) # verbose=True shows epoch progress

```

```
>>
```

```

GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

```

	Name	Type	Params	Mode
0	criterion	MSELoss	0	train
1	train_criterion	MSELoss	0	train
2	val_criterion	MSELoss	0	train
3	train_metrics	MetricCollection	0	train
4	val_metrics	MetricCollection	0	train
5	rnn	LSTM	12.9 K	train
6	V	Linear	33	train


```

13.0 K   Trainable params
0        Non-trainable params
13.0 K   Total params
0.052    Total estimated model params size (MB)
7        Modules in train mode
0        Modules in eval mode

```

The model summary confirms the setup, including hardware configuration (GPU), number of parameters, memory size, and layers (LSTM followed by linear).

10. Once training is complete, you can use TensorBoard for quick inspection:

```
%load_ext tensorboard
%tensorboard --logdir=darts_logs/
```

11. Darts provides `load_from_checkpoint` as a convenience method to help you load the best-performing model (lowest validation loss) from the checkpoint (with `best=True`). Once loaded, you can use the `predict` method to produce a forecast and plot the results to compare the forecast with the actuals:

```
best_darts_model = darts_lstm_model.load_from_checkpoint(
    model_name="energy_lstm", best=True)
```

12. You will use `historical_forecasts`, which is primarily used for model evaluation and backtesting to simulate how the model would have performed if you had made rolling one-step-ahead predictions in the past. In the following setup, for each forecasted month, the model takes as input the actual previous 12 months (never its own prediction). This is similar to our Keras implementation with one-step forecast. Note, for this process, you will need to provide the entire time series (`d_series`) covering train, validation, and test sets, so that at every forecast step, the model can access true historical context.

Here is how you can generate and evaluate one-step-ahead forecasts using `historical_forecasts`:

```
one_step_forecast_scaled= darts_lstm_model.historical_forecasts(
    series=d_series,
    start=d_test_scaled.start_time(),
    forecast_horizon=1, # We only want one-step-ahead predictions
    stride=1,           # Move one step at a time
    retrain=False,      # Do not retrain the model at each step
    verbose=True
)
one_step_forecast = d_scaler.inverse_transform(one_step_forecast_scaled)
rmse_val = rmse(d_test, one_step_forecast)
mae_val = mae(d_test, one_step_forecast)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 73.4404
Darts Test Set RMSE: 126.9493
```

13. Notice how, at each step, the model always uses the actual 12 months of data directly preceding the forecast point. It does not feed its own predictions back into the input window for the next step.



When using one-step-ahead forecasting approaches, such as Keras' one-step prediction or Darts' `historical_forecasts`, you can only forecast a single future point (e.g., one month ahead) beyond your available data.

Next, you will see how the behavior is different when using the `predict` method, which performs a multi-output forecast using a recursive strategy.

Using Darts RNNModel for recursive multi-output

1. Building on the previous steps, you will now use the `best_model` object to generate a recursive multi-output forecast through the `predict()` method. To align the forecast starting point with your test set, first, combine the training and validation sets. This creates a continuous historical forecast series ending where your test period begins:

```
# Combine training and validation data to form the input series
d_train_val_scaled = d_train_scaled.append(d_val_scaled)
# Generate recursive forecast for the entire test horizon
preds_scaled = best_darts_model.predict(n=len(d_test),
                                         series=d_train_val_scaled)

preds_inv = d_scaler.inverse_transform(preds_scaled)

# Calculate error metrics
rmse_val = rmse(d_test, preds_inv)
mae_val = mae(d_test, preds_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 115.5418
Darts Test Set RMSE: 161.0847
```

2. Unlike one-step forecasting, `predict()` uses a self-feeding approach:
 - The first prediction uses 12 actual months
 - The second prediction uses 11 actual months plus the one forecasted month
 - The third prediction uses 10 actual months plus the two forecasted months, and so on
3. As discussed in *Chapter 11*, this recursive strategy enables forecasting into completely unknown future periods (beyond your dataset), but suffers from compounding errors (error accumulation).
4. Next, you will explore direct multioutput approaches using Darts' `BlockRNNModel` and Neural-Forecast's LSTM, which predicts all future steps simultaneously to avoid error accumulation.

Using Darts BlockRNNModel for direct multi-output

Unlike recursive strategies, BlockRNNModel performs direct multi-step forecasting by predicting multiple future time steps simultaneously. This avoids error accumulation and typically improves accuracy for fixed horizons.

You will be building from the previous example and assuming you already have the data split and scaled:

1. Start by importing the BlockRNNModel class:

```
from darts.models import BlockRNNModel
```

2. In a similar manner, let's define our model parameters using a Python dictionary:

```
# Darts Block LSTM parameters
darts_blocklstm_params = {
    'model': 'LSTM',          # Specify the RNN type
    'input_chunk_length': 24, # How many time steps the model sees
                             # (your 'window')
    'output_chunk_length': 12, # How many steps the model predicts
                              # forward at a time
    'hidden_dim': 32,        # LSTM units
    'n_rnn_layers': 2,       # Number of LSTM layers
    'dropout': 0.2,          # Dropout rate
    'batch_size': 32,        # Batch size for training
    'n_epochs': 500,         # Max number of epochs
    'optimizer_kwargs': {'lr': 0.001}, # Adam is default
    'random_state': 42,      # For reproducibility
    'force_reset': True      # Ensure fresh model training
}
```

3. These are the key changes from RNNModel:
 - output_chunk_length=12 to forecast 12 steps at once (direct multi-output). In RNNModel, output_chunk_length always defaults to 1 (one-step ahead).
 - input_chunk_length=24 to provide a longer lookback to better capture seasonal patterns. In other words, we are training the model to take 24 months as input to predict the next 12 months.
4. The remaining configurations are identical to what we used with RNNModel, which includes EarlyStopping and lr_scheduler. Let's reuse our previous training configuration:

```
PATIENCE=15
trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)
```


5. Putting it all together, you can initialize and train the model:

```
darts_blocklstm_model = BlockRNNModel(
    **darts_blocklstm_params,
    model_name='energy_blocklstm',
    # Log_tensorboard=True,      # Set True to use TensorBoard
    save_checkpoints=True,      # Set True to save model checkpoints
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs # Pass trainer args including callbacks
)
```

6. Now, fit the model by providing the training and validation datasets:

```
darts_blocklstm_model.fit(d_train_scaled,
                          val_series=d_val_scaled,
                          verbose=True)
```

```
>>
```

```
GPU available: True (mps), used: True
```

```
TPU available: False, using: 0 TPU cores
```

```
HPU available: False, using: 0 HPUs
```

	Name	Type	Params	Mode
0	criterion	MSELoss	0	train
1	train_criterion	MSELoss	0	train
2	val_criterion	MSELoss	0	train
3	train_metrics	MetricCollection	0	train
4	val_metrics	MetricCollection	0	train
5	rnn	LSTM	12.9 K	train
6	fc	Sequential	396	train


```
-----
13.3 K   Trainable params
0        Non-trainable params
13.3 K   Total params
0.053    Total estimated model params size (MB)
8        Modules in train mode
Modules in eval mode
```

Notice the architectural difference in the final layers between `RNNModel` and `BlockRNNModel`. In `RNNModel`, the final layer is a linear layer to produce one scalar value (one output per timestep), requiring a recursive iteration for multi-output forecasts. In `BlockRNNModel`, the final layer is a sequential layer that produces a fixed-length vector output (e.g., 12 steps) in a single pass, which enables efficient direct multi-output forecasting.

7. Load the best performing model based on validation loss using `load_from_checkpoint`:

```
best_block_model = darts_blocklstm_model.load_from_checkpoint(
    model_name="energy_blocklstm", best=True
)
```

8. Use the `predict` method to generate the forecasts in one shot (direct multi-output):

```
d_train_val_scaled = d_train_scaled.append(d_val_scaled)
block_preds_scaled = best_block_model.predict(
    n=len(d_test),
    series=d_train_val_scaled)
block_preds_inv = d_scaler.inverse_transform(block_preds_scaled)

# calculate error metrics
rmse_val = rmse(d_test, block_preds_inv)
mae_val = mae(d_test, block_preds_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 137.6819
Darts Test Set RMSE: 183.7500
```

Notice how the MAE and RMSE have significantly improved on a longer future horizon compared to the recursive strategy.

9. You can plot the forecast and compare it against the actual test set:

```
PATIENCE=15
trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)
```

10. This should produce the following plot:

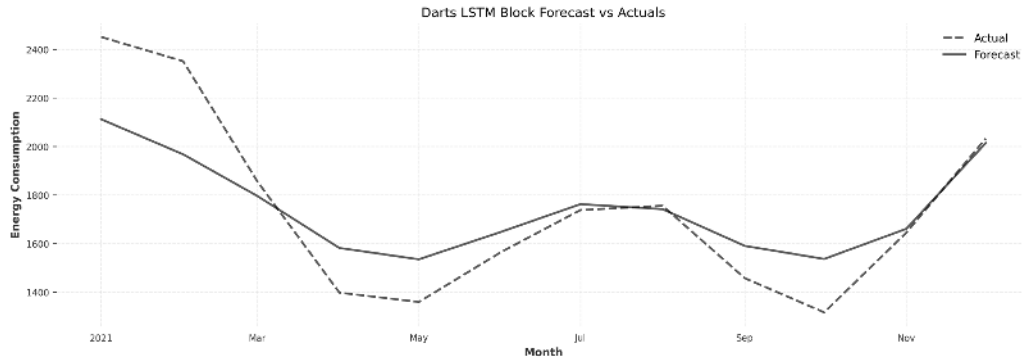


Figure 12.8: Darts block LSTM forecast against actual (test)

Using NeuralForecast for direct multi-output

In *Chapter 10*, you were introduced to **StatsForecast** from **Nixtla** and explored their **AutoArima**, **AutoETS**, and **AutoTheta** implementations. In addition to **StatsForecast**, **Nixtla** also offers **MLForecast**, **NeuralForecast**, **HierarchicalForecast**, and **TimeGPT**.

In this section, you will use **NeuralForecast**, a modern Python library designed for efficient and scalable neural forecasting, offering a unified, easy-to-use interface for time series modeling. It features a rich out-of-the-box collection of deep learning models, including **LSTM**, **MLP**, **RNN**, **TCN**, **NBEATS**, **NHiTS**, and **Transformers**. Additionally, **NeuralForecast** provides support for exogenous variables, probabilistic forecasting, and automatic model selection. In this recipe, you will use **NeuralForecast**'s **LSTM** model, which combines an **LSTM** encoder with an **MLP** decoder for fast and robust multi-step forecasting:

1. Start by importing the libraries and making a copy of the energy DataFrame:

```
from neuralforecast import NeuralForecast
from neuralforecast.models import LSTM as NF_LSTM
from neuralforecast.losses.pytorch import MAE, MSE
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.callbacks import ModelCheckpoint

logger = TensorBoardLogger("nf_tensorboard_log", name="LSTM")
# copy of the energy DataFrame
nf_df = energy.copy().reset_index()
```

2. Our `nf_df` DataFrame has two columns: `Month` and `y`. **NeuralForecast** expects the DataFrame to include at least these three columns: `unique_id`, `ds`, and `y`. These are default column names, and **NeuralForecast** allows custom names through the `id_col` `str = 'unique_id'`, `time_col`: `str = 'ds'`, and `target_col`: `str = 'y'` parameters in the `fit` method. The `unique_id` column (`id_col`) helps distinguish between multiple time series, and it is still required even for a single time series analysis.

Let's prepare our DataFrame to include a `unique_id` column and, for consistency, rename our `Month` column to `ds`:

```
nf_df.rename(columns={'Month': 'ds'}, inplace=True)
nf_df['unique_id'] = 'energy' # Add a unique_id column for NeuralForecast
print(nf_df.head())
>>
```

	ds	y	unique_id
0	1973-01-01	1957.641	energy
1	1973-02-01	1712.143	energy
2	1973-03-01	1510.079	energy
3	1973-04-01	1183.421	energy
4	1973-05-01	1006.326	energy

3. We will split our data into train and test using a similar split to that of our Keras and Darts implementation, with the last 12 months preserved (hold out) for testing. `NeuralForecast` allows splitting the training set further into a validation set, but it is done internally, and we don't have to explicitly split the data. You will see this in a later step. Let's split the data as previously mentioned:

```
WINDOW_SIZE = 24 # This is `input_size` in NeuralForecast terms
TEST_LENGTH = 12 # Number of steps to forecast, 'h' in NeuralForecast

train_nf = nf_df.iloc[:-TEST_LENGTH]
test_nf = nf_df.iloc[-TEST_LENGTH:]

print(train_nf.shape)
print(test_nf.shape)
>>
(576, 3)
(12, 3)
```

4. Similar to both Keras and Darts implementations, you will implement early stopping. By default, `NeuralForecast` does not save training checkpoints, so you need to enable this capability using a callback:

```
# Add checkpoint callback
checkpoint_callback = ModelCheckpoint(
    monitor="valid_loss",
    mode="min",
    save_top_k=1,
)
```

5. The `save_top_k` parameter controls how many of the best models (based on validation loss) will be saved:
 - `save_top_k=1` saves only the best model
 - `save_top_k=5` saves the top five best models
 - `save_top_k=-1` saves all models
 - `save_top_k=0` will not save any model (disables saving)
6. Let's define some of the parameters that are needed for the NeuralForecast LSTM model as a Python dictionary:

```
import math

batch_size = 32
steps_per_epoch = math.ceil(len(train_nf) / batch_size)

# NeuralForecast LSTM parameters
PATIENCE = 15
nf_lstm_params = {
    'h': TEST_LENGTH, # Forecast horizon
    'input_size': WINDOW_SIZE,
    'encoder_n_layers': 2, # Number of layers for the LSTM
    'encoder_hidden_size': 32, # Hidden state units
    # Dropout regularization applied to LSTM outputs:
    'encoder_dropout': 0.2,
    # to match 500 epochs like Keras and Darts:
    'max_steps': steps_per_epoch * 500,
    'batch_size': batch_size,
    'learning_rate': 0.001,
    'num_lr_decays': PATIENCE//3,
    # Number of validation iterations before early stopping:
    'early_stop_patience_steps': PATIENCE,
    # Number of training steps model evaluates its performance on val set:
    'val_check_steps': steps_per_epoch,
    'scaler_type': 'standard', # Type of scaler for normalization
    'random_seed': 42, # For reproducibility
    'logger': logger, # to use with TensorBoard
    # 'decoder_layers':0, # Set to zero to remove the MLP decoder layer.
    'enable_checkpointing': True,
    'recurrent': False, # True = Recursive while False = Direct
    'callbacks': [checkpoint_callback]
}

# Instantiate the NeuralForecast LSTM model
lstm = NF_LSTM(**nf_lstm_params)
```

Steps vs. epochs

Up to this point, you have trained models using Keras and Darts, where training duration is specified in terms of epochs. Now, with NeuralForecast, you will notice the library uses steps instead of epochs to specify training duration.

Let's break down some of these key concepts:

- **Batch size:** This determines the number of time series samples the model processes at once before updating internal parameters. For example, we've been using a batch size of 32.
- **Step:** A step occurs every time the model processes one batch and performs a parameter update.
- **Epoch:** An epoch represents one complete pass through the entire training dataset. Since the dataset consists of batches, one epoch consists of multiple steps. For instance, with 531 training records and a batch size of 32, one epoch requires $\text{ceil}(531/32) = 18$ steps.

While the terminology differs (epochs in Keras, `n_epochs` in Darts, and `max_steps` in NeuralForecast), the underlying concepts remain the same. You can always convert between epochs and total steps using the number of steps per epoch:

- $\text{Steps per epoch} = \text{ceil}(\text{Number of training samples} / \text{Batch size})$
- $\text{Total steps} = \text{Epochs} \times \text{Steps per epoch}$
- $\text{Epochs} = \text{Total steps} / \text{Steps per epoch}$

7. Instantiate the NeuralForecast LSTM model and create the NeuralForecast object. NeuralForecast can train more than one model; thus, it takes a list. Here, we will have a list that contains just one object (only the LSTM model):

```
# Instantiate the NeuralForecast LSTM model
lstm = NF_LSTM(**nf_lstm_params)

# Define the models to be used in NeuralForecast
models = [lstm]

# Create the NeuralForecast object
# The `freq` parameter time series alignment
nf_lstm_model= NeuralForecast(models=models, freq='MS')
```

8. Now, you are ready to train the model on the training set. `NeuralForecast` handles the data splitting internally for validation as well as scaling. The `fit` method takes a `DataFrame` for training and a validation size (`val_size`). It can be set to zero or set to the size of the forecast horizon (in our case, `TEST_LENGTH`). While `val_size` is optional, it becomes required when using early stopping. If you enable early stopping without setting `val_size > 0`, you will encounter an exception:

```
# Fit the model.
# NeuralForecast handles data splitting internally for validation
nf_lstm_model.fit(df=train_nf, val_size= 36)
>>
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
```

	Name	Type	Params	Mode
0	loss	MAE	0	train
1	padder_train	ConstantPad1d	0	train
2	scaler	TemporalNorm	0	train
3	hist_encoder	LSTM	12.9 K	train
4	mlp_decoder	MLP	4.4 K	train

```
-----
17.3 K    Trainable params
0         Non-trainable params
17.3 K    Total params
0.069     Total estimated model params size (MB)
10        Modules in train mode
0         Modules in eval mode
```

9. Once training is complete, and like the previous two implementations (Keras and Darts), you can launch `TensorBoard` for visual inspection:

```
%load_ext tensorboard
%tensorboard --logdir=nf_tensorboard_log/
```

10. To make a prediction, you can simply use the `predict()` method. It will default to use the forecast horizon (`h`) defined in the model and start from the end of the training dataset provided (in our case, `train_nf`):

```
nf_forecast = nf_lstm_model.predict()
```

11. This should produce a 12-month forecast, which you can compare against your out-of-sample test dataset:

```
nf_forecast.columns
>>
Index(['unique_id', 'ds', 'LSTM'], dtype='object')
```

12. Notice the returned DataFrame has three columns: `unique_id`, `ds`, and `LSTM` (which contains our forecast). The `unique_id` column is used to distinguish between multiple time series. You can extract the predictions and set `ds` as the index:

```
# Extract the predictions for our 'energy' unique_id and set the index
nf_forecast_df = nf_forecast[
    nf_forecast['unique_id'] == 'energy'
].set_index('ds')

# Select the 'LSTM' column, which contains the predictions
nf_forecast_df = nf_forecast_df[['LSTM']]
```

13. Plot forecast against actual (test data) and compute both MAE and RMSE scores:

```
test_nf.set_index('ds', inplace=True)

plt.figure(figsize=(16, 5))
test_nf['y'].plot(label='Actual', linestyle='--', alpha=0.65)
nf_forecast_df['LSTM'].plot(label='Forecast', color='k', alpha=0.65)
plt.title('NeuralForecast LSTM Multioutput Forecast vs Actuals')
plt.xlabel('Month')
plt.ylabel('Energy Consumption')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()

# Calculate metrics
mae_nf = mean_absolute_error(test_nf['y'], nf_forecast_df['LSTM'])
rmse_nf = np.sqrt(mean_squared_error(test_nf['y'], nf_forecast_
df['LSTM']))

print(f"NeuralForecast Test Set MAE: {mae_nf:.4f}")
print(f"NeuralForecast Test Set RMSE: {rmse_nf:.4f}")
>>
NeuralForecast Test Set MAE: 117.4197
NeuralForecast Test Set RMSE: 146.7713
```


14. This should produce the plot comparing forecast versus actual, as shown in Figure 12.9.

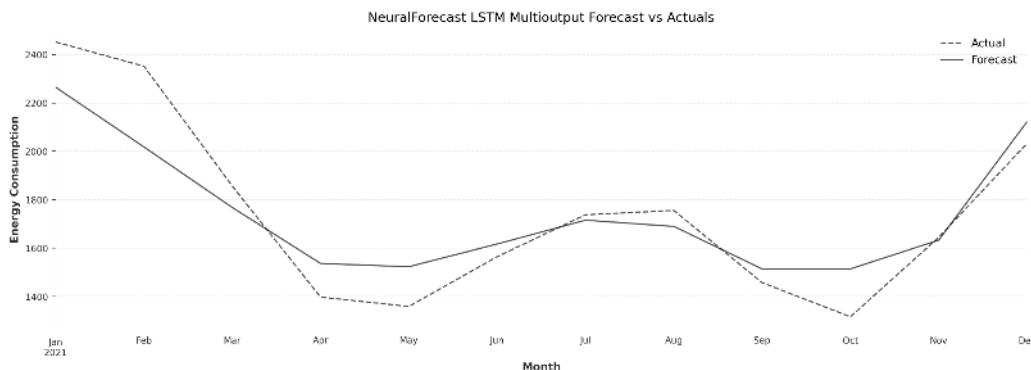


Figure 12.9: NeuralForecast LSTM forecast (LSTM encoder with MLP decoder) against actual (test)

How it works...

In this recipe, you trained an LSTM model on the *Monthly Energy Consumption* dataset and produced a forecast using three libraries: Keras (TensorFlow), Darts (PyTorch), and NeuralForecast (PyTorch).

Figure 12.10 summarizes the overall steps performed in the recipe, which include data preprocessing (windowing, splitting, and scaling), model fitting, forecasting, and evaluation. The diagram shows an example of transforming our time series into a windowing sequence (here, illustrated with 5 time steps, $x_1, \dots, x_5$, for visual clarity, though in our Python code in the recipe, we used 10 time steps).

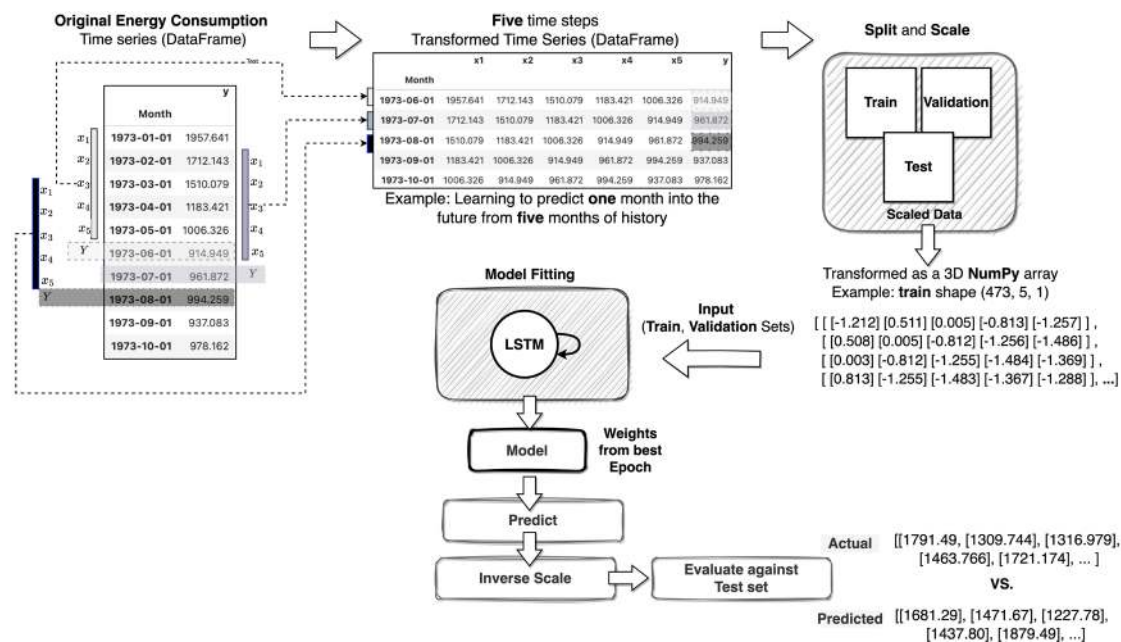


Figure 12.10: Summary of the overall process of training an LSTM model in the Forecasting with LSTM recipe

While the forecast plots from Darts (*Figure 12.8*), and NeuralForecast (*Figure 12.9*) show visually similar performance in capturing the energy consumption patterns, you will notice the MAE and RMSE scores are different (*Table 12.1*).

Model	MAE	RMSE	Last Epoch - Early Stopping	Best Epoch
keras_lstm_model (one-step)	143.5149	163.1068	Epoch 63	Epoch 48
darts_lstm_model (recursive/ autoregressive)	115.5418	161.0847	Epoch 119	Epoch 104
darts_blocklstm_model (direct multioutput)	137.6819	183.7500	Epoch 47	Epoch 31
nf_lstm_model	117.4197	146.7713	Epoch 30 (Step 539 / 18)	Epoch 15 (Step 270 / 18)

Table 12.1: Comparing MAE and RMSE scores from the three models (Keras, Darts, and NeuralForecast)

The varying early stopping points (*Table 12.1*) indicate different convergence patterns.

Although we aimed to align several high-level configurations (such as the input window, number of LSTM layers, learning rate, early stopping patience, and the 12-period evaluation window hold-out set for testing), distinct factors contribute to these differences:

- LSTM layer architecture and implementation:** One key difference lies in the LSTM unit configuration. Our Keras model uses 32 units in the first LSTM layer and 16 in the second layer. Keras provides you with granular control over each layer's configuration, so you can modify the number of units, for example, reduce the units in the second LSTM layer to create a funnel effect. In contrast, both Darts' `RNNModel` (with `n_rnn_layers=2` and `hidden_dim=32`) and NeuralForecast's LSTM (with `encoder_n_layers=2` and `encoder_hidden_size=32`) apply consistent 32 units to both LSTM layers.

NeuralForecast adds two MLP decoder layers, each 128 units by default, after the LSTM stack (see *Figure 12.11*). This architectural feature, unique to NeuralForecast in this comparison, can help the model learn more complex relationships when predicting multiple future time steps at once (multi-output forecast). You can disable this by setting `decoder_layers=0`. Furthermore, as illustrated in *Figure 12.11*, the dropout strategy differs: Keras applies explicit Dropout layers after each LSTM layer, while both Darts and NeuralForecast leverage PyTorch's internal LSTM dropout, which, by default, applies dropout between stacked layers but not after the final LSTM layer's output.

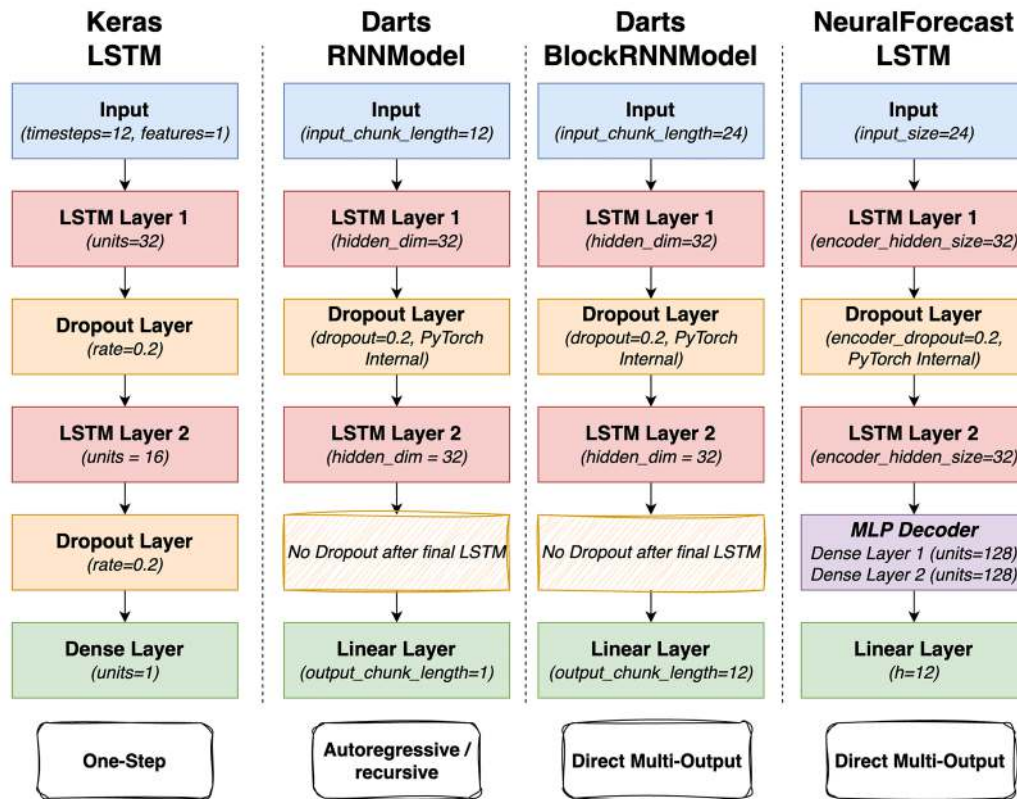


Figure 12.11: Comparing Keras, Darts, and NeuralForecast LSTM architectures implemented in the recipe

- Forecasting strategy:** NeuralForecast's LSTM implementation is designed for multi-output forecasting and provides flexibility (direct or recursive). This is controlled through the `recurrent=False` parameter for direct and `True` for recursive. By default, it will produce the entire forecast window (in our case, 12 periods) in a single forward pass. This is different from the recursive approach used in our Keras and Darts `RNNModel` examples (where the model predicts one step at a time and then feeds that prediction back into itself for the next step). In Darts, `BlockRNNModel` offered a similar mechanism to NeuralForecast but without the MLP layers. In *Chapter 11*, we discussed the different strategies, such as *recursive*, *direct*, *multi-output*, and *direct-recursive* (DirRec).
- Library-specific implementations:** Beyond explicit parameters, Keras (using the TensorFlow backend), NeuralForecast, and Darts (which leverages PyTorch Lightning) have their own low-level implementations for data handling, gradient computation and application, optimization routines, and precise execution of training loops and predictions. These internal differences can lead to variations in the training path and the final converged model, even when high-level parameters appear similar.
- Scaling:** While all models use standardization, NeuralForecast handles scaling internally with its `scaler_type` parameter.

- **Learning rate scheduling:** While Keras and Darts easily support ReduceLROnPlateau scheduling (monitoring validation loss to reduce learning rate when improvement plateaus), NeuralForecast uses StepLR by default, which reduces learning at fixed intervals. Although NeuralForecast can be configured to use ReduceLROnPlateau through the `lr_scheduler` and `lr_scheduler_kwargs` parameters, it does not support the `monitor` parameter required by ReduceLROnPlateau (as we used in our Keras and Darts implementations). Attempting to use ReduceLROnPlateau directly will result in a `MisconfigurationException` error, indicating that a `monitor` parameter must be specified in the scheduler configuration. To work around this limitation, you would need to subclass the model class (e.g., the LSTM class) and override the `configure_optimizers` method to properly configure the scheduler with its required `monitor` parameter. For a complete example, see the *Nixtla documentation*: https://github.com/Nixtla/neuralforecast/blob/main/nbs/docs/tutorials/configure_optimizers.ipynb.

For maximum control, Keras (or vanilla PyTorch) provides more granular control over all architecture and training details. Darts strikes a good balance between flexibility and productivity. NeuralForecast is the easiest to implement (for multi-output forecasting); however, it does obscure some of the internal workings (though it still exposes a variety of configuration options for most forecasting tasks).

There's more...

When training deep learning models with early stopping, understanding the difference between the *best* model (selected based on validation performance) and the *last* model (from the final training epoch) is crucial. Each library handles model persistence and loading differently, which directly impacts your forecasting results.

Recall that in Darts, we retrieved the weights of the best model, evaluated based on validation scores using the `load_from_checkpoint(model_name="energy_lstm", best=True)` method. This is different from Keras. In Keras, the `lstm_model` model will have its weights automatically restored to those of the epoch that achieved the best performance on the monitored metric (`val_loss`, in our case). This is due to `restore_best_weights=True` in `EarlyStopping`. So, calling `lstm_model.predict()` will inherently use these best weights (stored in memory) without needing an extra loading step, as we had to do in Darts.

In Keras, we also implemented `ModelCheckpoint` to persist the weights for future reference. Hence, you can also manually load the weights of the best model using the `load_weights` method:

```
keras_lstm_model.load_weights('./logs/best_model.keras')
```

In Darts, let's say you were to use the `predict` method directly from `darts_lstm_model`, as shown here:

```
pred = darts_lstm_model.predict(n=len(val))
last_model_pred = scaler.inverse_transform(pred)
```

This would produce a different forecast since it will be using the model from the **last epoch** (the model in memory after `fit()`). In most cases, the model from the last epoch may not be the optimal model and may perform worse when compared to the best model from a checkpoint, as it may start to overfit.

Inside the `darts_logs` folder, there is the `checkpoints` folder. Darts makes it pretty easy to persist checkpoints without needing to implement the `ModelCheckpoint` class; all you have to do is set `save_checkpoints=True`. Inside that folder, you may see two files. In my case, I have a `.ckpt` file for the best epoch (`best-epoch=48-val_loss=0.11.ckpt`) and another for the last epoch (`last-epoch=63.ckpt`). When you load from a checkpoint, it will be reading the `best-epoch=48-val_loss=0.11.ckpt` file, while if you use `predict` directly, it will be using the `last-epoch=63.ckpt` file. The `.ckpt` file is a type used to store parameters (weights) of a trained model.

In `NeuralForecast`'s `nf_lstm_model` model, the behavior is similar to Darts, where using `predict` will leverage the model from the last epoch. Using the best model is more involved, as you will need to manually load the learned weights from the `.ckpt` file. As you'll recall from our implementation, `NeuralForecast` stored its checkpoints in the `nf_tensorboard_log/LSTM` folder. Inside, you'll find a `version_0` subfolder (or `version_1`, `version_2`, etc., if you retrain the model). The checkpoint file (e.g., `epoch=269-step=270.ckpt`) contains the model's weights. Start by updating our model parameters:

```
# Rebuild parameters with early stopping disabled
reload_params = nf_lstm_params.copy()
reload_params.update({
    'max_steps': 0,                                # Prevent training
    'early_stop_patience_steps': -1, # Disable NeuralForecast's
                                           # auto EarlyStopping
})
reload_params.pop('callbacks', None) # Remove any callback references
```

We just updated our parameters; by setting `max_steps: 0`, we prevent any additional training, since we don't want to train our model again. Instead, we want to load the model. Setting `early_stop_patience_steps: -1` ensures that no new early stopping criteria are applied during the loading phase.

Load model weights using the `load_from_checkpoint` method:

```
checkpoint_path = 'nf_tensorboard_log/LSTM/version_0/checkpoints/epoch=269-
step=270.ckpt'
nf_best_model = NF_LSTM.load_from_checkpoint(checkpoint_path, **reload_params )
```



If `load_from_checkpoint` fails, verify that the `.ckpt` file path matches your training run's `version_xx` folder. The exact checkpoint filename (e.g., `epoch=269-step=270.ckpt`) will vary based on your training run. Look for the `.ckpt` file in your `version_xx/checkpoints/` folder.

We continue as we did before with `NeuralForecast`:

```
nf_loaded = NeuralForecast(models=[nf_best_model], freq='MS')
nf_loaded.fit(df=train_nf, val_size=0)
```

Unlike Darts, NeuralForecast requires a data context to make predictions, even when loading from a checkpoint. This is why we need to call `fit()` after loading, but with a crucial difference: we set `max_steps=0` in our parameters to prevent any actual training from occurring. The `fit()` call only establishes the data context needed for prediction while preserving our loaded model weights. To make our forecast, as we did before, use the `predict` method using the loaded model:

```
nf_best_preds = nf_loaded.predict()
```

In this recipe, you were introduced to three different libraries (Keras, Darts, and NeuralForecast). The following is a summary of their differences:

Library	Best Model Loading	Default Predict Uses	Forecasting Strategy	Output Layer Architecture
Keras	Automatic if <code>EarlyStopping(restore_best_weights=True)</code> ; else, manual via <code>load_weights()</code>	Best weights if using <code>EarlyStopping</code> and <code>restore_best_weights=True</code> ; else, last epoch	Recursive (autoregressive)	Linear (Dense) for scalar output
Darts	Manual via <code>load_from_checkpoint(best=True)</code>	Model from the last epoch	<code>RNNModel</code> : Recursive <code>BlockRNNModel</code> : Direct	<code>RNNModel</code> : Linear <code>BlockRNNModel</code> : Sequential (FCN)
Neural-Forecast	Manual via <code>load_from_checkpoint()</code>	Model from the last epoch	Configurable: Direct or Recursive	MLP (two-layer by default)

Table 12.2: Comparison table that summarizes some of the differences in Keras, Darts, and NeuralForecast

Recall that the `RNNModel` and `BlockRNNModel` classes provide access to the three RNN variants (vanilla RNN, LSTM, and GRU) through the model parameter ("RNN", "LSTM", or "GRU"). Notice that when predicting using NeuralForecast, the output was already transformed (inverse transformed), and we did not need to scale back the data, which is different from Darts and Keras, in which we had to apply inverse transformation.

See also

- To explore additional parameters and configuration options for Darts' `RNNModel` class, refer to their documentation: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.rnn_model.html
- To explore additional parameters and configuration options for Darts' `BlockRNNModel` class, refer to their documentation: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.block_rnn_model.html
- To explore additional parameters and configuration options for NeuralForecast's LSTM class, refer to their documentation: <https://nixtlaverse.nixtla.io/neuralforecast/models.lstm.html>

Handling multivariate time series and exogenous variables

In the previous recipe, *Forecasting with LSTM*, we explored LSTM implementation across different libraries, including Keras, Darts, and NeuralForecast. However, our focus was based on **univariate forecasting**. In this recipe, you will extend that foundation by diving into **multivariate forecasting**.

Multivariate forecasting can unlock deeper insights by modeling interactions between multiple time-dependent variables. Energy consumption, for instance, doesn't exist in isolation; it is influenced by external factors such as temperature, humidity, and precipitation. By incorporating these factors as **covariates** (also called **exogenous variables**), we can improve our forecasting performance beyond what univariate models can achieve.

The key to successful multivariate modeling lies in selecting impactful variables that influence our target variable. This process often involves exploration and experimentation to find the right combination of covariates. You should always benchmark your results against a univariate baseline to ensure the added complexity provides meaningful improvements.

Different libraries handle multivariate time series differently, and in this recipe, you will explore both `BlockRNNModel` from Darts and `LSTM` from NeuralForecast. Many models within Darts and NeuralForecast support multivariate time series as well as exogenous variables (covariates), and you can explore their capacities directly from their documentation:

- **Darts model capabilities:** <https://unit8co.github.io/darts/#forecasting-models>
- **NeuralForecast model capabilities:** <https://nixtlaverse.nixtla.io/neuralforecast/docs/capabilities/overview.html>

When working with covariates, it is critical to understand their natures, as they generally fall into two categories:

- **Past covariates** are known up to the forecasting point. For example, using historical economic indicators to forecast stock prices, where future indicators are unknown (you will only have historical data, but no future knowledge).
- **Future covariates** are known in advance, including throughout the forecasting horizon. For example, using weather forecasts to model energy consumption, since meteorological predictions are often available for future days and can extend beyond our forecast period.

This is where framework differences become critical. `BlockRNNModel`, due to its architecture, is limited to using **past covariates only**, while other models might support both types. This architectural constraint will shape our modeling approach in this recipe and demonstrate why understanding each framework's unique capabilities matters for your forecasting success.

How to do it...

You will be using the **City of Zurich Electricity Consumption** dataset available from the Darts library. This dataset combines electricity consumption data with weather measurements from Zurich, spanning from 2015 to 2022. We'll resample the original 15-minute data to daily averages for our forecasting task.

For this recipe, we'll focus on predicting household and small business electricity consumption using weather-related covariates. Specifically, we will use the following:

- The `Value_NE5` target variable: Daily average household and **SME (Small and Medium Sized Enterprises)** electricity consumption (kWh)
- Covariates (weather features):
 - `T [°C]`: Daily average temperature
 - `StrGlo [W/m2]`: Daily average global solar irradiation
 - `RainDur [min]`: Daily total precipitation duration
 - `Hr [%Hr]`: Daily average relative humidity

These weather variables were picked based on an intuitive sense as covariates since temperature drives heating/cooling demand, solar irradiation affects energy generation, precipitation influences overall consumption patterns, and humidity impacts comfort levels requiring climate control adjustments.

The following steps are similar to our previous recipe, *Forecasting with LSTM*, with the main differences being the inclusion of covariates and using different input and output window sizes.

Using BlockRNNModel from Darts

1. Start by importing the necessary libraries for this recipe:

```
from darts.metrics import rmse, mae
from darts.dataprocessing.transformers import Scaler
from darts.models import BlockRNNModel
from darts.datasets import ElectricityConsumptionZurichDataset

import pandas as pd
from pytorch_lightning.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
import torch
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = [14, 7]
```


2. Load the *Zurich Electricity Consumption* dataset. The data is recorded in 15-minute intervals, and we will resample it to a daily frequency. The final dataset consists of daily average energy consumption and daily average weather measurements. The loaded dataset will be a Darts `TimeSeries` object. We will also create a pandas `DataFrame` version for other uses, including compatibility with `NeuralForecast`:

```
zurich = (
    ElectricityConsumptionZurichDataset()
    .load()
    .resample(freq="D", method='mean')
    .astype(np.float32)
)

# convert to a pandas DataFrame
zurich_df = zurich.to_dataframe()
zurich_df.info()
```

```
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2800 entries, 2015-01-01 to 2022-08-31
Freq: D
Data columns (total 10 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   Value_NE5             2800 non-null  float32
 1   Value_NE7             2800 non-null  float32
 2   Hr [%Hr]              2800 non-null  float32
 3   RainDur [min]         2800 non-null  float32
 4   StrGlo [W/m2]         2800 non-null  float32
 5   T [°C]                2800 non-null  float32
 6   WD [°]                2800 non-null  float32
 7   WVs [m/s]             2800 non-null  float32
 8   WVv [m/s]             2800 non-null  float32
 9   p [hPa]               2800 non-null  float32
dtypes: float32(10)
memory usage: 131.2 KB
```

3. We will focus on a few meaningful covariates. Define the target variable and the covariates as Python lists:

```
cov = ["T [°C]", "StrGlo [W/m2]", "RainDur [min]", "Hr [%Hr]"]
target = ["Value_NE5"]
```

4. You can plot all five variables (target and covariates) to observe their behavior:

```
zurich_df[cov+target].plot(subplots=True, figsize=(10,10));
```

This will generate five subplots, one for each variable, as shown here:

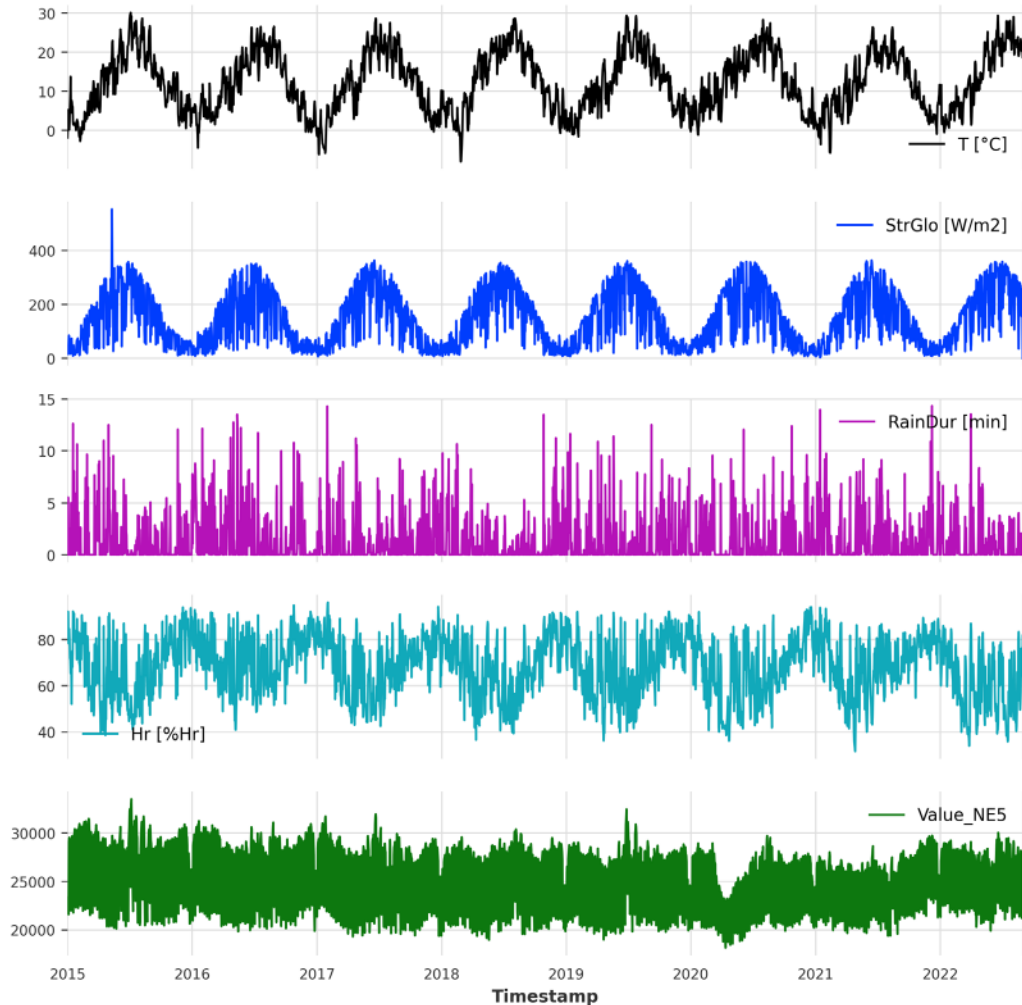


Figure 12.12: Daily average electricity consumption and weather covariates in Zurich (2015–2022)

The plots in *Figure 12.12* allow you to observe overall trends, variability, and potential relationships between electricity consumption (Value_NE5) and weather covariates over time.

- While the plots in *Figure 12.12* show the overall time series, you can get a clearer picture of the seasonal patterns. You can do this using the `month_plot` and `quarter_plot` functions from the `statsmodels` library:

```
from statsmodels.graphics.tsaplots import month_plot, quarter_plot

# Resample target to monthly/quarterly for clearer seasonality
monthly_target = zurich_df[target].resample('ME').mean()
quarterly_target = zurich_df[target].resample('QE').mean()

month_plot(monthly_target, ylabel='kWh')
plt.title('Monthly Seasonality: Avg. Electricity Consumption')
plt.show()
```

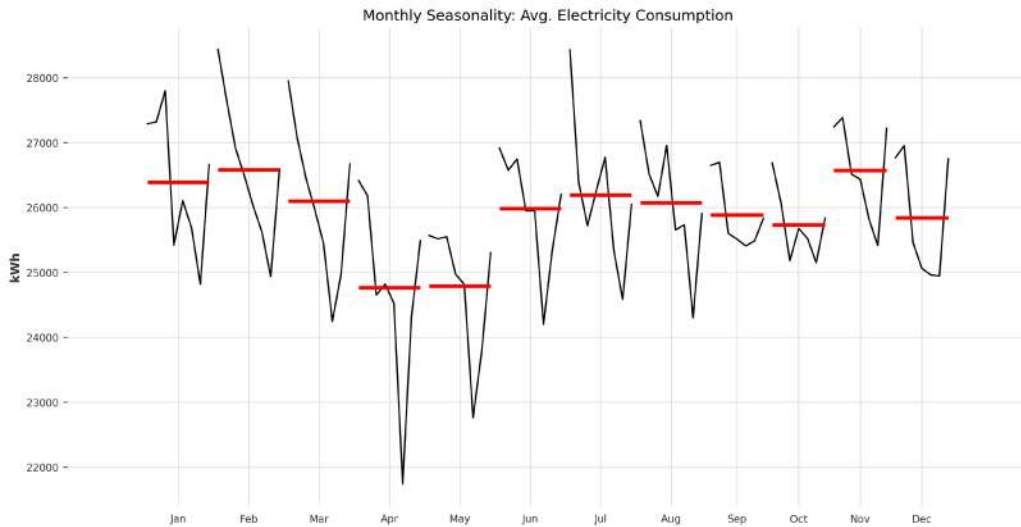


Figure 12.13: Monthly seasonality in electricity consumption using `month_plot`

The plot in *Figure 12.13* aggregates data from the same month across all years (e.g., all Januarys, all Februarys, etc.) to reveal the underlying seasonal cycle. The horizontal lines (red lines) indicate the grand mean for each month.

- You can plot the quarterly patterns using `quarter_plot`:

```
quarter_plot(quarterly_target, ylabel='kWh')
plt.title('Quarterly Seasonality: Avg. Electricity Consumption')
plt.show()
```

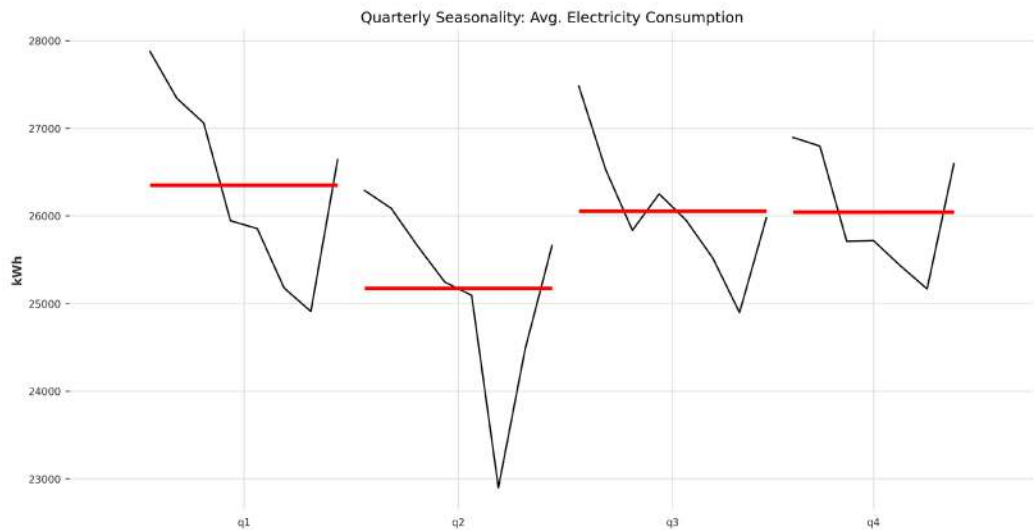


Figure 12.14: Quarterly seasonality in electricity consumption using `quarter_plot`

The plot in *Figure 12.14* aggregates data from the same quarter across all years (e.g., all Q1s, all Q2s, etc.) to reveal the underlying seasonal cycle. The horizontal lines (red lines) indicate the grand mean for each quarter.

The two plots (*Figure 12.13* and *Figure 12.14*) reveal the following:

- `month_plot` in *Figure 12.13* shows the electricity consumption peaks during the cold winter months, from November through March, most likely driven by increased heating and lighting demand.
 - `quarter_plot` in *Figure 12.14* reveals a similar seasonal pattern, with Q1 (January–March) exhibiting the highest consumption, which generally decreases as the quarter progresses. The average line (horizontal red line) is relatively elevated, suggesting higher demand during winter.
7. Split the time series data into train, validation, and test sets while maintaining temporal order using the `split_ts` utility function:

```
from utils import split_ts
```

The `split_ts` function ensures that we maintain chronological order when splitting time series data.

8. You will separate the target variables (`Value_NE5`) from the covariates (weather features):

```
zurich_cov = zurich[cov]
zurich_en = zurich[target]
```

9. Use the `split_ts` function to split each dataset (`zurich_en` and `zurich_cov`) into training, validation, and test sets. Our goal is to forecast 30-day energy consumption, so we will hold out the last 30 days as our test set:

```
# Create splits maintaining temporal order
d_zen_train, d_zen_val, d_zen_test = split_ts(zurich_en, 30, 160)

d_cov_train, d_cov_val, d_cov_test = split_ts(zurich_cov, 30, 160)
```

Always split your data before any scaling or transformation to prevent data leakage.

10. Apply scaling using the `Scaler` wrapper class from Darts with `StandardScaler` from scikit-learn. Recall from the previous recipe that if no scaler is provided, it defaults to `MinMaxScaler` from scikit-learn. You will perform scaling on the target time series and then on the covariates. Fit the scaler only on the training set, then transform the validation and test sets. In other words, we apply scaling using parameters derived only from the training set to prevent data leakage and ensure realistic model evaluation:

```
scaler_target = Scaler(StandardScaler())

d_zen_train_sc = scaler_target.fit_transform(d_zen_train)
d_zen_val_sc = scaler_target.transform(d_zen_val)
d_zen_test_sc = scaler_target.transform(d_zen_test)

# Covariates time series
scaler_cov = Scaler(StandardScaler())

d_cov_train_sc = scaler_cov.fit_transform(d_cov_train)
d_cov_val_sc = scaler_cov.transform(d_cov_val)
d_cov_test_sc = scaler_cov.transform(d_cov_test)
```

11. This approach ensures that scaling parameters are derived only from the training data, preserving the integrity of your validation and test evaluations.
12. Now, define the `BlockRNNModel` parameters, including the `EarlyStopping` callback, `lr_scheduler`, and model architecture. We will reuse the LSTM setup from the first recipe, *Forecasting with LSTM*. The adjustments would be `input_chunk_length=120` and `output_chunk_length=30`:

```
PATIENCE=15
trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)

model = BlockRNNModel(
    model="LSTM",
    input_chunk_length=120, # (4 months) captures seasonal patterns
```

```

        output_chunk_length=30, # 30 matches forecast horizon
        n_rnn_layers=2,
        hidden_dim=32,
        dropout=0.2,
        batch_size=32,
        n_epochs=500,
        optimizer_kwargs={"lr": 0.001},
        model_name="Zur_BlockRNN",
        log_tensorboard=True,
        random_state=42,
        force_reset=True,
        save_checkpoints=True,
        lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
        lr_scheduler_kwargs= lr_scheduler_kwargs,
        pl_trainer_kwargs=trainer_kwargs
    )

```

13. Fit the model with the scaled target data and covariates (past covariates). You will need to provide validation data for early stopping and learning rate scheduling for both target and covariates:

```

model.fit(
    series=d_zen_train_sc,
    past_covariates=d_cov_train_sc,
    val_series=d_zen_val_sc,
    val_past_covariates=d_cov_val_sc,
    verbose=True)

```

14. After training, load the model checkpoint with the best validation loss using the `load_from_checkpoint` method:

```

zen_best_model = model.load_from_checkpoint(
    model_name="Zur_BlockRNN", best=True
)

```

15. To forecast using the `predict` method, you will need to provide `past_covariates`. To generate a forecast for the test period, the model needs the preceding data as a starting point. We will provide it with the full history from the combined training and validation sets. Append the validation data to the training data for both the target and covariates, then forecast the next 30 days using the best model. Since our model has an `input_chunk_length` value of 120 days, it will use the most recent 120 days from this combined dataset as context for generating the 30-day forecast:

```

d_zen_train_val_sc = d_zen_train_sc.append(d_zen_val_sc)
d_cov_train_val_sc = d_cov_train_sc.append(d_cov_val_sc)

```

```
zen_forecast_sc = zen_best_model.predict(
    n=len(d_zen_test_sc),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc
)
```

16. Since the predicted output is scaled, apply the **inverse transformation** to convert back to the original scale:

```
zen_forecast_inv = scaler_target.inverse_transform(zen_forecast_sc)
```

17. You can plot the forecasted values against the actual test set:

```
plt.figure(figsize=(16, 5))
d_zen_test.plot(label='Actual', linestyle='--', alpha=0.65)
zen_forecast_inv.plot(label='Forecast', color='k',
                      linestyle='--', alpha=0.65)
plt.title('Darts Zurich Energy Forecast vs Actuals (Test Set)')
plt.xlabel('Day')
plt.ylabel('Energy Consumption')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

This should produce the following plot:

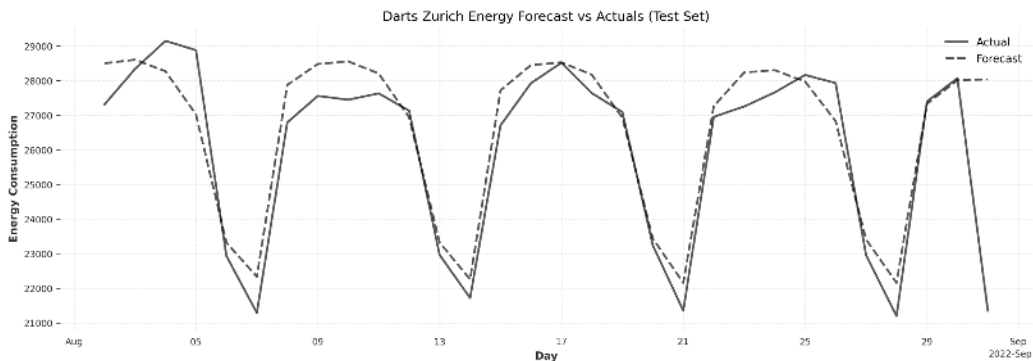


Figure 12.15: Darts Zurich Energy 30-day forecast vs. actual (test set)

18. Compute the MAE and RMSE to assess the model's predictive accuracy:

```
rmse_val = rmse(d_zen_test, zen_forecast_inv)
mae_val = mae(d_zen_test, zen_forecast_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
```

```
>>
Darts Test Set MAE: 835.9444
Darts Test Set RMSE: 1435.1460
```

Using LSTM from NeuralForecast

In this subrecipe, you will explore how to use the NeuralForecast library to build and evaluate an LSTM model with covariates (exogenous variables):

1. Start by importing the necessary libraries for this recipe:

```
from neuralforecast import NeuralForecast
from neuralforecast.models import LSTM as NF_LSTM
from neuralforecast.losses.pytorch import MAE, MSE
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.callbacks import ModelCheckpoint
from sklearn.metrics import mean_absolute_error, mean_squared_error

logger = TensorBoardLogger("nf_tensorboard_log", name="LSTM_Multi")
```

2. Let's prepare the `zurich_df` DataFrame to be suitable for NeuralForecast. This includes renaming the `Timestamp` column to `ds`, renaming our target column to `y`, and adding a `unique_id` column:

```
zurich_nf_df = zurich_df[target+cov].reset_index()

zurich_nf_df.rename(
    columns={'Timestamp': 'ds', 'Value_NE5': 'y'},
    inplace=True)
zurich_nf_df['unique_id'] = 'ZurichEnergy'
```

3. We will split our data in a similar manner to how we did it in Darts. The difference is that NeuralForecast handles the validation splitting internally. So, you will only need to split into training and test (hold-out) sets. NeuralForecast also automates scaling on training and inverse scaling the results when forecasting:

```
TEST_LENGTH=30

nf_zurich_test = zurich_nf_df.iloc[-TEST_LENGTH:]
nf_zurich_train = zurich_nf_df.iloc[:-TEST_LENGTH]
```


4. Configure the NeuralForecast LSTM model using a similar setup to BlockRNNModel in the previous subrecipe. A key difference is that we do not need to separate our target and covariates (exogenous variables). NeuralForecast simplifies preprocessing by specifying our exogenous variables by providing a list of column names in the `hist_exog_list` parameter:

```
import math

WINDOW_SIZE=120
batch_size = 32
steps_per_epoch = math.ceil(len(nf_zurich_train) / batch_size)

# Add checkpoint callback
checkpoint_callback = ModelCheckpoint(
    monitor="valid_loss",
    mode="min",
    save_top_k=1)

# NeuralForecast LSTM parameters
PATIENCE = 15
nf_lstm_params = {
    'h': TEST_LENGTH,
    'input_size': WINDOW_SIZE,
    'encoder_n_layers': 2,
    'encoder_hidden_size': 32,
    'encoder_dropout': 0.2,
    'max_steps': steps_per_epoch * 500,
    'batch_size': batch_size,
    'learning_rate': 0.001,
    'num_lr_decays': PATIENCE//3,
    'early_stop_patience_steps': PATIENCE,
    'val_check_steps': steps_per_epoch,
    'scaler_type': 'standard',
    'random_seed': 42,
    'logger': logger,
    'enable_checkpointing': True,
    'callbacks': [checkpoint_callback],
    'hist_exog_list': cov
}

# Instantiate the NeuralForecast LSTM model
nf_lstm = NF_LSTM(**nf_lstm_params)
```

```
# Define the models to be used in NeuralForecast
models = [nf_lstm]

# Create the NeuralForecast object
nf_lstm_model = NeuralForecast(models=models, freq='D')
```

5. Fit the model on the training set, specifying the validation size to match the Darts configuration:

```
nf_lstm_model.fit(df=nf_zurich_train, val_size= 160)
```

6. After training, use the predict method. You will need to supply future exogenous data for the test period:

```
fut_cov = nf_zurich_test.drop(columns=['y'])
nf_pred = nf_lstm_model.predict()
```

7. Plot the forecast against the actual values:

```
# Prepare the actual test data for plotting
nf_zurich_test.set_index('ds', inplace=True)

plt.figure(figsize=(16, 5))
nf_zurich_test['y'].plot(label='Actual', linestyle='--', alpha=0.65)
nf_pred.set_index('ds')[
    'LSTM'].plot(label='Forecast', color='k', alpha=0.65)
plt.title('NeuralForecast Zurich Energy Forecast vs Actuals (Test Set)')
plt.xlabel('Month')
plt.ylabel('Energy Consumption')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

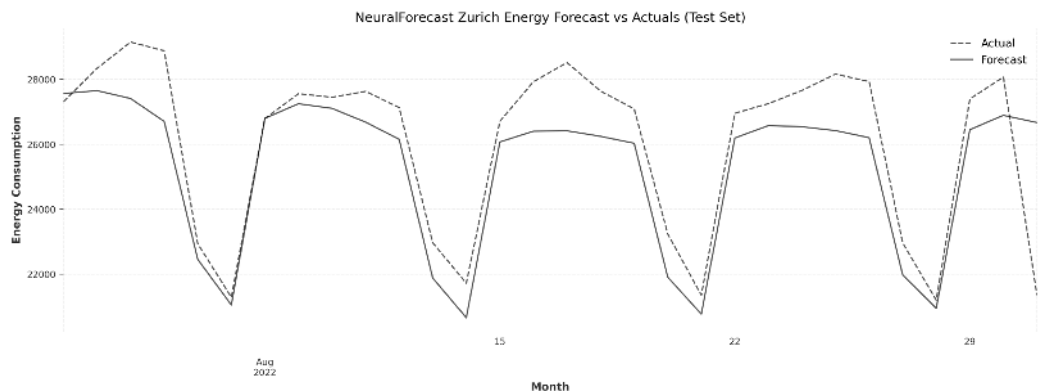


Figure 12.16: NeuralForecast LSTM Zurich Energy 30-day forecast vs. actuals (test set)

8. Calculate MAE and RMSE to assess the model's accuracy:

```
mae_nf = mean_absolute_error(nf_zurich_test['y'], nf_pred['LSTM'])
rmse_nf = np.sqrt(mean_squared_error(nf_zurich_test['y'], nf_pred['LSTM']))

print(f"NeuralForecast Test Set MAE: {mae_nf:.4f}")
print(f"NeuralForecast Test Set RMSE: {rmse_nf:.4f}")

>>
NeuralForecast Test Set MAE: 1121.0948
NeuralForecast Test Set RMSE: 1473.5518
```

How it works...

In this recipe, you explored how to use `BlockRNNModel` from Darts and `LSTM` from NeuralForecast for multivariate time series forecasting. Unlike traditional univariate models, which only learn from the target variable's history, multivariate models incorporate covariates (also called exogenous variables) at each time step. This allows the model to capture external influences and often leads to improved forecasting performance.

Both `BlockRNNModel` and `LSTM` support multi-output forecasting, something we explored in the first recipe, *Forecasting with LSTM*.

Both Darts and NeuralForecast support covariates, but they handle them differently:

- **Darts:**
 - **Accepts only past covariates:** `BlockRNNModel` is designed to explicitly support past covariates whose values are known up to the forecast point (i.e., historical data only). This is in contrast to `RNNModel`, which can handle both past and future covariates.
 - **No forecasting of covariates:** `BlockRNNModel` does not predict or use future values of covariates during forecasting. All input to the model comes from the past. For example, the model learns how historical weather patterns influence future consumption.
- **NeuralForecast:**
 - **Flexible covariate support:** NeuralForecast's `LSTM` model can incorporate historical, future, and static exogenous variables (covariates).
 - **Historical covariates** via `hist_exog_list`: These are variables whose values are only available and known up to the forecast point, just like `BlockRNNModel` from Darts.
 - **Future covariates** via `futr_exog_list`: These are variables whose values are known for the entire forecast horizon. This means you need to provide the actual future values (for example, obtaining weather forecasts) alongside the target series. When future covariates are provided, the `LSTM` model directly consumes these values within the prediction window, allowing it to anticipate changes in the target variable driven by known upcoming events or conditions.

- **Static covariates** via `stat_exog_list`: This refers to variables that do not change over time for a given series (e.g., city, product category, or store).

There's more...

In our main example, we purposely used `hist_exog_list` to ensure a fair comparison with Darts' past-covariate-only approach in `BlockRNNModel`. However, `NeuralForecast`'s strength lies in its ability to use future covariates when available.

`NeuralForecast`'s LSTM model supports future covariates through `futr_exog_list`. Before we dive in, we need to generate our future covariates, and we will create a function that will help us generate these covariates. Remember, the weather data in our dataset represents *actual historical* weather measurements, not forecasts. Using actual weather measurements as future covariates would introduce data leakage, as we would be using information that wouldn't be available in a real forecasting scenario. This makes the weather measurements (actual historical) suitable as past covariates for `hist_exog_list`, but not suitable to use as future covariates. The `add_time_covariates` function will allow us to create time-based features that are inherently predictable:

```
def add_time_covariates(df, ds_col='ds'):
    """
    Given a DataFrame with a DateTime column,
    returns a new DataFrame with added time-related future covariates.

    Returns:
        pd.DataFrame: DataFrame with additional time covariate columns.
    """
    df = df.copy()

    # Feature engineering
    df['hour'] = df[ds_col].dt.hour
    df['dayofweek'] = df[ds_col].dt.dayofweek
    df['month'] = df[ds_col].dt.month
    df['is_weekend'] = df[ds_col].dt.dayofweek.isin([5, 6]).astype(int)

    return df
```

The function creates simple features that are perfect for future covariates:

- **Hour, day of week, month:** Cyclical patterns that are always known in advance
- **Weekend indicator:** Derived from the day of the week, captures behavioral changes

You could enhance this function to include holidays and other known time-related events.

The `add_time_covariates` function takes a `DataFrame` and the name of the datetime columns, and it will return a new `DataFrame` with additional time-based features. We'll run this twice, once for the training data and again for the test data to ensure consistency:

```
nf_zurich_train_fc = add_time_covariates(nf_zurich_train, 'ds')
nf_zurich_test_fc = add_time_covariates(nf_zurich_test.reset_index(), 'ds')
```

We are resetting the index for `nf_zurich_test` because in the previous exercise, we set `ds` as an index. `NeuralForecast` needs `ds` to be a column and not an index.

Create a Python list of our future covariates that we will use with the `futr_exog_list` parameter:

```
fc_cov = ['hour', 'dayofweek', 'month', 'is_weekend']
```

Let's rerun the same model, but this time, we will add `futr_exog_list`:

```
# NeuralForecast LSTM parameters
nf_lstm_params_fc = {
    'h': TEST_LENGTH,
    'input_size': WINDOW_SIZE,
    'encoder_n_layers': 2,
    'encoder_hidden_size': 32,
    'encoder_dropout': 0.2,
    'max_steps': steps_per_epoch * 500,
    'batch_size': batch_size,
    'learning_rate': 0.001,
    'num_lr_decays': PATIENCE//3,
    'early_stop_patience_steps': PATIENCE,
    'val_check_steps': steps_per_epoch,
    'scaler_type': 'standard',
    'random_seed': 42,
    'logger': logger,
    'enable_checkpointing': True,
    'callbacks': [checkpoint_callback],
    'hist_exog_list': cov,
    'futr_exog_list': fc_cov
}

# Instantiate the NeuralForecast LSTM model
nf_lstm_fc = NF_LSTM(**nf_lstm_params_fc)

# Define the models to be used in NeuralForecast
models_fc = [nf_lstm_fc]

# Create the NeuralForecast object
```

```
nf_lstm_model_fc = NeuralForecast(models=models_fc, freq='D')

nf_lstm_model_fc.fit(df=nf_zurich_train_fc, val_size= 160)
```

After training, use the `predict` method. You will need to supply future exogenous data for the test period. Extend the `fc_cov` list to include the `unique_id` and `ds` columns required by `NeuralForecast`:

```
fc_cov.extend(['unique_id', 'ds'])

fut_cov_df = nf_zurich_test_fc.loc[:, fc_cov] # extract future covariates
nf_pred_fc = nf_lstm_model_fc.predict(futr_df=fut_cov_df) # supply future
covariates

# Calculate metrics
mae_nf = mean_absolute_error(nf_zurich_test['y'], nf_pred_fc['LSTM'])
rmse_nf = np.sqrt(mean_squared_error(nf_zurich_test['y'], nf_pred_fc['LSTM']))

print(f"NeuralForecast Test Set MAE: {mae_nf:.4f}")
print(f"NeuralForecast Test Set RMSE: {rmse_nf:.4f}")
>>
NeuralForecast Test Set MAE: 717.4011
NeuralForecast Test Set RMSE: 1374.3146
```

The performance improvement in MAE and RMSE scores demonstrates the value of future covariates when available. Time-based future covariates are more reliable than weather forecasts or economic indicators. Calendar information is always available and accurate for any future horizon, making this approach more practical. The 36% improvement in MAE demonstrates that even simple, always-available future covariates can dramatically enhance forecasting performance.

See also

- To explore additional parameters and configuration options for Darts' `BlockRNNModel` class, refer to their documentation: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.block_rnn_model.html
- To explore additional parameters and configuration options for `NeuralForecast`'s LSTM class, refer to their documentation: <https://nixtlaverse.nixtla.io/neuralforecast/models.lstm.html>

Hyperparameter tuning for deep learning forecasters

In *Chapter 11*, we explored the fundamentals of hyperparameter tuning for machine learning models using grid search. Building high-performance deep learning is not just about designing the right architecture; it's also about fine-tuning the many parameters that control how the model learns and adapts to your data.

From the previous recipes, you have already experienced that deep learning forecasters such as LSTM are packed with hyperparameters that require careful tuning, including *learning rate*, *number of layers*, *hidden units*, *dropout rates*, *batch size*, and more. Each of these choices can impact how well your model learns patterns and generalizes to unseen data. The core challenge is in finding the optimal combination of these hyperparameter values without having to manually test different configurations.

This is where Optuna comes into play, a powerful and flexible optimization library designed to automate this search process efficiently. What makes Optuna particularly appealing is its ability to prune unpromising trials early, saving you valuable computational time. Both Darts and NeuralForecast seamlessly integrate with optimization backends such as Optuna, allowing you to tune your models with just a few extra lines of code. It's also worth noting that *sktime*, the library we extensively used in *Chapter 11* for building machine learning models, also supports Optuna through `ForecastingOptunaSearchCV`.

This recipe builds directly on the multivariate time series forecasting examples from the previous recipe, where you worked with both Darts and NeuralForecast. We will tune the Darts `BlockRNNModel` class and the NeuralForecast LSTM class to see how each library handles the tuning process. While the implementation details will differ slightly, the core concepts remain the same, giving you a solid foundation for optimizing any deep learning forecaster you build in the future.

Getting ready

You will need to install the Optuna library for this recipe. You can install Optuna using `pip install`:

```
pip install optuna
```

How to do it...

Both Darts and NeuralForecast make the process of hyperparameter tuning straightforward. NeuralForecast supports hyperparameter tuning using two powerful optimization frameworks: **Ray Tune** and **Optuna**. Darts takes a more flexible approach, giving you three options to play with: Optuna, Ray Tune, and the classic grid search. While grid search might feel old-school, it's still incredibly useful when you want complete control over the search process or when working with smaller parameter spaces.

Using Optuna with Darts

Darts offers a flexible and direct integration with Optuna. You can pretty much use any Darts model within a custom objective function that you will create. This approach gives you complete control over the tuning loop, allowing you to define the exact search space and logic for each trial.

In this section, you will build an objective function from scratch to find the best hyperparameters for `BlockRNNModel`. The function will pretty much wrap the same configurations used in the previous recipe, *Handling multivariate time series and exogenous variables*, with a few adjustments for tuning.

This recipe builds on the previous one and assumes that we have the libraries imported, have created our `Scaler` object (`scaler_target`), and have the `d_zen_train_sc`, `d_zen_train_sc`, `d_cov_train_sc`, `d_cov_val_sc`, and `d_zen_test` datasets:

1. Start by importing the Optuna library:

```
import optuna
```

2. You will define the objective function as required by Optuna: the function will be called by Optuna for a set number of trials and should include the hyperparameter space. You will notice the objective function wraps the code from the previous section with these notable modifications:
 - Define the hyperparameter search space: Within the objective function, you specify how Optuna samples hyperparameters by using methods such as `suggest_float` and `suggest_categorical`. This process is known as hyperparameter sampling, and the values returned from these methods are referred to as suggested values for each trial.
 - Flexible hyperparameter selection: In this example, three hyperparameters are optimized (`hidden_dim`, `dropout`, and `lr`), but you can easily adjust this approach to handle additional hyperparameters or modify the value ranges as needed. For example, we can tune the `patience` parameter for early stopping, or `n_rnn_layers` for model architecture.
 - Specify suggestion methods for each parameter: For each hyperparameter, you will need to use the appropriate suggestion method available by Optuna (e.g., `suggest_float` or `suggest_categorical`), which defines how values are sampled during optimization.
3. Now, let's define our objective function:

```
def objective(trial):
    # Suggest hyperparameters
    PATIENCE=15
    hidden_dim = trial.suggest_categorical("hidden_dim", [16, 32, 64])
    dropout = trial.suggest_float("dropout", 0.0, 0.3, step=0.1)
    lr = trial.suggest_categorical("lr", [1e-4, 1e-3, 1e-2])

    pl_trainer_kwargs = make_trainer_kwargs(PATIENCE)
    lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)

    model_name = f"LSTMHyperTune_trial_{trial.number}"
    # Model instantiation
    model = BlockRNNModel(
        model="LSTM",
        input_chunk_length=120,
        output_chunk_length=30,
        n_rnn_layers=2,
        hidden_dim=hidden_dim,
        dropout=dropout,
        batch_size=32,
        n_epochs=500,
        optimizer_kwargs={"lr": lr},
        model_name=model_name,
```



```

        random_state=42,
        force_reset=True,
        save_checkpoints=True,
        loss_fn=torch.nn.L1Loss(), # to use MAE loss
        lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
        lr_scheduler_kwargs= lr_scheduler_kwargs,
        pl_trainer_kwargs=pl_trainer_kwargs
    )

    # Fit the model
    model.fit(
        series=d_zen_train_sc,
        past_covariates=d_cov_train_sc,
        val_series=d_zen_val_sc,
        val_past_covariates=d_cov_val_sc,
        verbose=False
    )

    score = model.trainer.callback_metrics['val_loss'].item()

    return score

```

4. Overall, the structure and core model setup remain unchanged from our previous recipe. The main difference is how we now define certain parameters:
 - Hyperparameter integration: Instead of hardcoding values for `hidden_dim`, `dropout`, and `lr`, now we use the hyperparameter suggestions that we defined. So, in each trial, Optuna dynamically samples these values (based on a sampler strategy we will define next), allowing for automated hyperparameter tuning.:
 - `hidden_dim`: Categorical values (i.e., [16, 32, 64])
 - `dropout`: Float values between 0.0 and 0.3 with a step of 0.1 (i.e., [0.0, 0.1, 0.2, 0.3]).
 - `lr`: Categorical learning rates (i.e., [1e-4, 1e-3, 1e-2])
 - Loss function update: By default, Darts uses `MSELoss` (`torch.nn.MSELoss`). In this version, we will update it to be based on `MAE` (`torch.nn.L1Loss`). This can, in some cases, improve model robustness.

5. After defining the objective function, we set up and run the Optuna optimization process by creating a study and specifying a sampler:

```
sampler = optuna.samplers.TPESampler(seed=42)
study = optuna.create_study(direction="minimize", sampler=sampler)
study.optimize(objective, n_trials=10)

print("Best trial:")
print(study.best_trial.params)
print("Best Validation MAE:", study.best_value)
```

The `optuna.create_study` function initializes a new Optuna study, which helps manage our tuning experiments and acts like our workspace. Here, we set the direction of the optimization, which, in our case, is "minimize", because our goal is to find the combination of hyperparameters that produces the lowest MAE score (this is what our objective function is returning).

In addition to `direction`, the study takes a `sampler`, which tells Optuna which strategy to use for selecting the combination of hyperparameters to try. While it defaults to `TPESampler`, we defined it explicitly as `optuna.samplers.TPESampler(seed=42)` with a random seed for reproducibility. `TPESampler` stands for **T**ree-**s**tructured **P**arzen **E**stimator **S**ampler, a popular Bayesian optimization algorithm that learns from past trials. Optuna also offers other samplers such as `RandomSampler`, `GridSampler`, and `CmaEsSampler`. For a full list, check the Optuna documentation: <https://optuna.readthedocs.io/en/stable/reference/samplers/index.html>.

The `optimize` method runs the objective function for `n_trials=10` times. In each trial, `TPESampler` suggests a new combination of hyperparameters, and study keeps track of the results for each trial to find the best set of hyperparameters.

To illustrate this, we used `model_name = f"LSTMHyperTune_trial_{trial.number}"` to dynamically create a different model name for each trial. Darts will create a separate folder for each model (trial) inside the `darts_logs` directory.

6. During optimization, Optuna logs trial results, which display the selected hyperparameter values, the best trial so far, and the score of the best trial. Here is an example of the last trial:

```
Trial 9 finished with value: 0.33913829922676086 and parameters:
{'hidden_dim': 64, 'dropout': 0.0, 'lr': 0.01}. Best is trial 6 with
value: 0.23227311670780182.
Best trial:
{'hidden_dim': 32, 'dropout': 0.2, 'lr': 0.01}
```

7. The print statements at the end will show the best hyperparameters and MAE:

```
Best trial:
{'hidden_dim': 32, 'dropout': 0.2, 'lr': 0.01}
Best Validation MAE: 0.23227311670780182
```



This can be a time-consuming process and will vary depending on your hardware. For example, on my machine, the search took about eight minutes to complete using the GPU on a MacBook Pro (M1 Max). It may run a little faster or slower for you, depending on your specific setup.

8. Once the trials are completed, you can access the set of hyperparameter values that produced the best score (the best trial's hyperparameters):

```
best_params = study.best_trial.params
best_params
>>
{'hidden_dim': 32, 'dropout': 0.2, 'lr': 0.01}
```

9. You can use these optimal values to train our final model. In the following example, we will update our model configuration using these values, and then fit the model:

```
PATIENCE=15
trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)

best_model = BlockRNNModel(
    model="LSTM",
    input_chunk_length=120,
    output_chunk_length=30,
    n_rnn_layers=2,
    hidden_dim=best_params['hidden_dim'], # from Optuna
    dropout=best_params['dropout'],      # from Optuna
    batch_size=32,
    n_epochs=500,
    optimizer_kwargs={"lr": best_params['lr']}, # from Optuna
    model_name="LSTMBestModel",
    random_state=42,
    force_reset=True,
    save_checkpoints=True, # Set True if you want to save checkpoints now
    loss_fn=torch.nn.L1Loss(),
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs=lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs)
```

10. Now, you can fit the model:

```
# Fit the model with transformed target data and covariates
best_model.fit(
    series=d_zen_train_sc,
    past_covariates=d_cov_train_sc,
    val_series=d_zen_val_sc,
    val_past_covariates=d_cov_val_sc,
    verbose=True)
```

11. After training, load the model checkpoint with the best validation loss using the `load_from_checkpoint` method. This ensures you are using the model state from the epoch with the lowest validation error:

```
zen_best_model = best_model.load_from_checkpoint(
    model_name="LSTMBestModel", best=True
)
```

12. The remaining code should be familiar and similar to the code we used in the previous recipe. Here, we make our forecast and evaluate our results against the unseen test data. Combine the train and validation splits:

```
d_zen_train_val_sc = d_zen_train_sc.append(d_zen_val_sc)
d_cov_train_val_sc = d_cov_train_sc.append(d_cov_val_sc)
```

13. Forecast on the test set, and evaluate the results:

```
zen_forecast_sc = zen_best_model.predict(
    n=len(d_zen_test_sc),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc)
zen_forecast_inv = scaler_target.inverse_transform(zen_forecast_sc)
rmse_val = rmse(d_zen_test, zen_forecast_inv)
mae_val = mae(d_zen_test, zen_forecast_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 782.6993
Darts Test Set RMSE: 1419.1954
```

14. Since hyperparameter tuning can be time-consuming, it's a good idea to save the final model for future use using the `save` method:

```
zen_best_model.save("darts_tuned_model.pt")
```

15. To load the model, you can use the `load` method from the `BlockRNNModel` class:

```
model_loaded = BlockRNNModel.load("darts_tuned_model.pt")
```

Using Optuna with NeuralForecast

NeuralForecast provides hyperparameter tuning through its *Auto* model classes, such as `AutoLSTM`, `AutoNHITS`, and `AutoTFT`. You can find a complete list of *Auto* models in their official documentation here: <https://nixtlaverse.nixtla.io/neuralforecast/models.html>.

Generally, each core model in NeuralForecast has a corresponding *Auto* version that handles Optuna (or Ray) integration internally. In this section, you will be using the `AutoLSTM` class.

This recipe will continue from the previous recipe and assumes we have the libraries imported from before, including the `nf_zurich_train` and `nf_zurich_test` datasets. You will also be referencing the list of covariate column names created earlier under the `cov` variable:

1. Start by importing the additional libraries for this section:

```
import optuna
from neuralforecast.auto import AutoLSTM

# Use this to disable training prints from optuna
optuna.logging.set_verbosity(optuna.logging.WARNING)
```

2. You will use the same parameters from the previous recipe, *Handling multivariate time series and exogenous variables* (using NeuralForecast with historical exogenous variables):

```
import math

TEST_LENGTH=30
WINDOW_SIZE=120

batch_size = 32
steps_per_epoch = math.ceil(len(nf_zurich_train) / batch_size)

# NeuralForecast LSTM parameters
PATIENCE = 15
nf_lstm_params = {
    'input_size': WINDOW_SIZE,
    'encoder_n_layers': 2,
    'encoder_hidden_size': 32,
    'encoder_dropout': 0.2,
    'max_steps': steps_per_epoch * 500,
    'batch_size': batch_size,
    'learning_rate': 0.001,
```

```

    'num_lr_decays': PATIENCE//3,
    'early_stop_patience_steps': PATIENCE,
    'val_check_steps': steps_per_epoch,
    'scaler_type': 'standard',
    'random_seed': 42, #
    'hist_exog_list': cov #as defined earlier
}

```

3. NeuralForecast handles things a bit differently when it comes to integration with Optuna. First, you will define a `config_lstm` function. Unlike the objective function in Darts that returns a score, this function's role is to define the hyperparameter search space and return a dictionary of configurations for each trial:

```

def config_lstm(trial):
    # Start with a copy of the base parameters
    configs = {**nf_lstm_params}
    # Update with the hyperparameters to be tuned by Optuna
    configs.update({
        "learning_rate": trial.suggest_categorical(
            "learning_rate", [1e-4, 1e-3, 1e-2]
        ),
        "encoder_hidden_size": trial.suggest_categorical(
            "encoder_hidden_size", [16, 32, 64]
        ),
        "encoder_dropout": trial.suggest_float(
            "encoder_dropout", 0.0, 0.3, step=0.1
        )
    })

    return configs

```

The function will return the configuration or hyperparameter values selected as a dictionary. This is different from Darts' objective function, in which we returned the score of the trial run.

4. Instantiate the `AutoLSTM` class and pass the `config_lstm` function and the sampler you created in the *Darts* section:

```

hp_model = AutoLSTM(
    h=TEST_LENGTH,
    loss=MAE(),
    config=config_lstm,
    search_alg=sampler, # defined in the Darts section
    backend='optuna',
    num_samples=10
)

```

We are reusing `sampler = optuna.samplers.TPESampler(seed=42)` defined earlier. The `num_samples=10` parameter sets the number of trials, similar to `n_trials` in the *Darts* recipe, telling Optuna how many hyperparameter combinations to test.

5. Create a `NeuralForecast` object and fit it. `NeuralForecast` can train multiple models (e.g., LSTM, RNN, AutoLSTM) by passing a list of models. In our example, we only have our single model, `hp_model`, an AutoLSTM model:

```
nf_hp = NeuralForecast(models=[hp_model], freq='D')
nf_hp.fit(df=nf_zurich_train, val_size= 160)
```



As with *Darts*, hyperparameter tuning can take a while. For instance, fitting took about 20 minutes using the GPU on a MacBook Pro (M1 Max). Your time may vary depending on hardware specs, especially since `NeuralForecast`'s *Auto* models include more steps under the hood.

Similar to *Darts*, `NeuralForecast` also creates a folder for each tuning trial, typically inside a `lightning_logs` folder. In each trial's folder, it stores a `hparams.yaml` file. This YAML file records the configuration and hyperparameter values used for that particular run. Because the format is plain text, you can open and inspect it in any text editor to see exactly what settings were used for each trial.

6. Once training is complete, you can access the best hyperparameter values from the best trial:

```
nf_hp.models[0].results.best_trial.params
>>
{'learning_rate': 0.001, 'encoder_hidden_size': 16, 'encoder_dropout':
0.3}
```

Notice the results are different from *Darts*: `{'hidden_dim': 64, 'dropout': 0.2, 'lr': 0.0001}`.



It's normal for the best hyperparameters with `NeuralForecast` to differ from those found using *Darts*. Implementation details, default settings, and search procedures vary between libraries, so don't be concerned if your results aren't an exact match.

7. You can review the results of all 10 trials as a `pandas DataFrame` for more detailed analysis:

```
results = nf_hp.models[0].results.trials_dataframe()
```

The resulting `DataFrame` includes trial duration, selected parameters, and performance metrics across all 10 trials.

8. NeuralForecast simplifies prediction. You do not need to retrain the model; calling the predict method will automatically use the best model found during the tuning process:

```
Y_hat_df_optuna = nf_hp.predict()
# Calculate metrics
mae_nf = mean_absolute_error(nf_zurich_test['y'], Y_hat_df_
optuna['AutoLSTM'])
rmse_nf = np.sqrt(mean_squared_error(nf_zurich_test['y'], Y_hat_df_
optuna['AutoLSTM']))

print(f"NeuralForecast Test Set MAE: {mae_nf:.4f}")
print(f"NeuralForecast Test Set RMSE: {rmse_nf:.4f}")
>>
NeuralForecast Test Set MAE: 782.5295
NeuralForecast Test Set RMSE: 1273.3864
```

9. Since hyperparameter tuning can be time-consuming, you can save the final model for future use using the save method:

```
nf_hp.save("nf_tuned_model" , overwrite=True)
```

10. To load the model, you can use the load method from the NeuralForecast class:

```
nf_loaded = NeuralForecast.load("nf_tuned_model")
```

How it works...

In machine learning, there are generally two types of parameters:

- **Model parameters:** These are values that the model learns on its own during training, such as the weights and biases inside the LSTM layers.
- **Hyperparameters:** These are configuration settings that you define before model training begins. They influence the model's structure and how it learns, which in turn influences the model parameters. This includes **learning rate**, **dropout rate**, and **number of hidden units**.

Hyperparameter tuning is an essential part of optimizing deep learning models, where the goal is to find the optimal combination of hyperparameter values. Traditionally, finding hyperparameters involved manual guesswork or brute-force methods such as **grid search**, which tries every possible combination. These methods can be very time-consuming and inefficient at times, especially with a large search space.

This is where Optuna comes in. It is platform-agnostic and automates the search process and efficiently navigates through large and complex search spaces. As you saw in the Darts and NeuralForecast implementations, the process revolves around three key concepts:

- **Study:** An Optuna study keeps track of every experiment (trial), including the hyperparameters tested and the resulting scores. For longer experiments, you can persist the study to a **relational database (RDB)** backend (e.g., SQLite, PostgreSQL, or MySQL). This is achieved by setting the storage parameter inside the `create_study` method, which takes a SQLAlchemy URL (refer to *Chapter 2* and *Chapter 4* for a refresher on SQLAlchemy).
- **Trial:** This is a single experiment (run) that involves training a model on a set of specific hyperparameter values and evaluating its performance to get a score. In our recipe, we chose MAE as the evaluation metric.
- **Sampler:** This is the critical part, as it defines the algorithm used to pick the next combination of hyperparameter values based on insights from outcomes of previous trials. This allows Optuna to learn which regions of the search spaces are most promising.

In the recipe, we used `TPESampler` from Optuna, a Bayesian optimization algorithm for efficient hyperparameter tuning. It builds a probabilistic algorithm that models the relationship between hyperparameters and scores (learns and adapts from previous trial results in real time) to distinguish between good and poor performing value ranges. To learn about the algorithm, you can refer to the paper *Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance*, which is available at <https://arxiv.org/pdf/2304.11127>.

In our recipe, for both Darts and NeuralForecast, we were optimizing (minimizing) one target metric (e.g., `val_loss`). By default, Optuna uses `TPESampler` for single-objective optimization when calling the `create_study` method. If you were performing multiple objectives, for example, two or more metrics (e.g., accuracy, latency, or `val_loss`) that you are trying to optimize for (minimize or maximize), then Optuna will default to `NSGAIISampler` instead. To explicitly use `TPESampler` for multi-objective search, you will need to pass it explicitly as we did in our code example. Setting the sampler's seed parameter ensures reproducible results.

When saving a tuned model, the libraries store the necessary artifacts to recreate it.

Saving with NeuralForecast will create a folder containing the following:

- `AutoLSTM_0.ckpt` (`modelName_suffix.ckpt`): This stores the model learned weights, parameters, and our set of hyperparameters as a PyTorch Lightning checkpoint
- `configuration.pkl`: A pickle file with configuration attributes
- `dataset.pkl`: A pickle file containing the dataset object used for fitting

When saving with Darts, for most models, it uses standard pickle. For Torch-based models (such as `BlockRNNModel`), it will create PyTorch Lightning checkpoints (`.ckpt`), similar to NeuralForecast.

It is worth mentioning that both Darts and NeuralForecast support other advanced tuning libraries, such as **Ray Tune** (<https://docs.ray.io/en/latest/tune/index.html>) for hyperparameter tuning.

Finally, as you saw, NeuralForecast offered a more streamlined approach with its `AutoLSTM` class, but Darts provided more fine-grained control over each trial by having you build the objective function from scratch.

There's more...

Optuna provides several powerful visualizations to help you inspect and understand the results of your tuning trials. It offers two backends for this:

- **Interactive** visualizations via the Plotly backend, which can be accessed through `optuna.visualization`
- **Static** visualizations via the Matplotlib backend, which can be accessed through `optuna.visualization.matplotlib`

In order to use the interactive visualization, you will need to install Plotly:

```
pip install plotly
```

Let's explore some of these interactive visualizations for both Darts and NeuralForecast:

- **Optimization history plot** (`plot_optimization_history`): Shows how the model's objective value (such as MAE) changes across each trial during hyperparameter tuning
- **Parameter importance plot** (`plot_param_importances`): Ranks the impact of each hyperparameter on the objective metric, highlighting which parameters are most influential for model performance
- **Timeline plot** (`plot_timeline`): Visualizes the start and end times of each trial, helping detect scheduling bottlenecks or unusually long-running trials in the optimization process
- **Slice plot** (`plot_slice`): Illustrates the relationship between each hyperparameter value and the objective value for all trials, revealing how specific hyperparameter choices affect results

In our Darts implementation, you can directly use the study object created:

```
from optuna.visualization import (plot_optimization_history,
                                  plot_param_importances,
                                  plot_timeline,
                                  plot_slice)

plot_optimization_history(study).show()
plot_param_importances(study).show()
plot_timeline(study).show()
plot_slice(study).show()
```

For NeuralForecast, you first need to access the study object that AutoLSTM saved for us:

```
nf_study = nf_hp.models[0].results
```

Then, use that for the visualization:

```
plot_optimization_history(nf_study).show()
plot_param_importances(nf_study).show()
plot_timeline(nf_study).show()
plot_slice(nf_study).show()
```

In Figure 12.17, the optimization history and parameter importance for the Darts study are shown. The top chart indicates that *Trial 0* was our best trial, setting the *best value* benchmark represented by the red line. The bottom plot shows that *lr* (learning rate) had the highest influence on the score, followed by *hidden_dim*, and finally, *dropout*.



Figure 12.17: Optuna visualization plots for the Darts hyperparameter tuning study

See also

- To learn more about hyperparameter optimization options in NeuralForecast, visit their official documentation: https://nixtlaverse.nixtla.io/neuralforecast/docs/capabilities/hyperparameter_tuning.html
- For a deeper dive into Darts' optimization user guide (including Optuna and Ray Tune), visit their official documentation: https://unit8co.github.io/darts/userguide/hyperparameter_optimization.html#hyperparameter-optimization-using-gridsearch
- To learn more about Optuna's capabilities, check out their official documentation: <https://optuna.readthedocs.io/en/stable/index.html>
- If you are interested in learning about Ray Tune, check out their **Getting Started** page: <https://docs.ray.io/en/latest/ray-overview/getting-started.html>

Forecasting with TCNs

In the previous recipes, we covered core deep learning concepts and techniques for time series forecasting, including early stopping, learning rate schedulers, and the use of covariates (exogenous variables)—both past and future covariates. We also discussed the importance of hyperparameter tuning and demonstrated this using tools such as Optuna.

To understand these libraries and concepts, we relied on LSTM networks due to their simplicity. We used an iterative approach to understand both model behavior and implementation details using libraries such as Darts and NeuralForecast. We achieved this by starting from basic to more advanced LSTM configurations.

Now, we are ready to explore more advanced architecture. However, it is important to note that just because there are more modern and recent innovations in model architecture, it does not mean they are better. For example, we should not discard things such as LSTMs. Classical deep learning models such as LSTM continue to be widely used for their proven ability to model complex, non-linear temporal dynamics. This applies to other models we have covered so far, from statistical models such as **Exponential Smoothing** and **SARIMA**, to machine learning models such as **XGBoost**.

This brings us back to a theme I have been emphasizing throughout: there is no one-size-fits-all solution, and this is even more true with time series forecasting. The choice, performance, and suitability of a model depend on many factors, such as domain context, forecasting objective, complexity of the data, and the nature of the time series process behind the data.

There are numerous instances where classical models can perform just as well as more advanced deep learning or machine learning models, and there are also numerous cases where deep learning models outperform and shine compared to statistical and traditional approaches. The difference is not only in the model or model architecture, but the nature of the data being used. Time series forecasting follows an iterative process with distinct **exploration** and **exploitation** phases: you begin by exploring various techniques (experimenting with feature engineering, trying different models), and once you discover what works (your “Aha!” moment), you shift to exploitation, optimizing that approach to get the maximum performance possible.

In this recipe, you will start the journey of exploring other models beyond LSTMs, beginning with TCNs. A TCN is a type of deep learning architecture designed for sequence modeling, leveraging causal and dilated convolutions to capture both short- and long-range dependencies in time series data.

Recall that LSTMs process data sequentially, with each hidden state depending on the previous state. When working with a large dataset and a long history, this process of sequential data processing can be slow and computationally intensive. In our previous implementations, we used two LSTM layers, but if our dataset was much larger and we stacked more LSTM layers to capture more complex patterns, the sequential processing time can become a bottleneck, and the training time would take a hit. This is where TCNs offer a compelling advantage due to their ability to capture long-term dependencies while processing all time steps in parallel, thus providing faster and better training stability than recurrent-type models.

While you might associate convolutions with image processing (which uses 2D convolutions), TCNs adapt the convolution concept for sequential data. This is achieved through 1D convolutions that are **causal** and **dilated**, allowing them to capture long-term dependencies and temporal patterns. The diagram in *Figure 12.18* illustrates how a TCN uses stacked layers of causal and dilated convolutions to achieve a large receptive field from an input sequence.

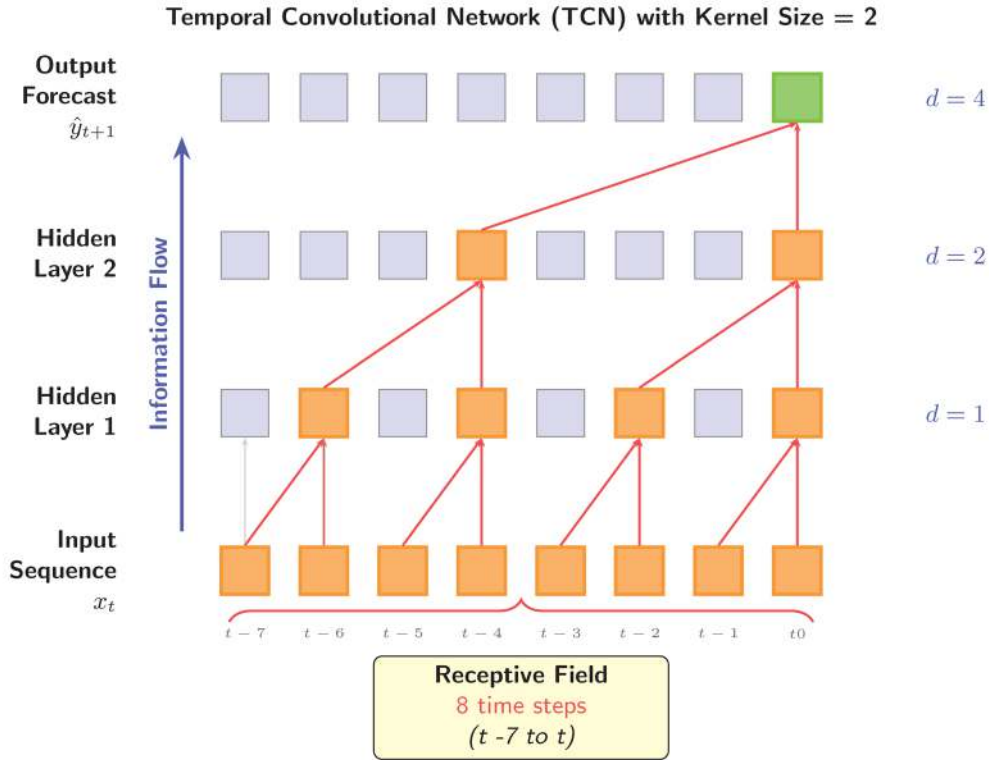


Figure 12.18: The architecture of a stacked TCN, illustrating how causal and dilated convolutions expand the model's receptive field

The network takes an input sequence (x_t) at the bottom (time steps from $t-7$ to t), and information flows upward through stacked hidden layers to produce an output forecast ($\hat{y}_{t+1}$). The represented architecture in the diagrams uses a kernel size of 2 and a dilation factor that doubles at each layer (1, 2, 4), resulting in a receptive field of 8 steps. This is calculated as $1 + (1 \times 1) + (2 \times 1) + (4 \times 1) = 8$, where each term represents the contribution of each layer's dilation factor multiplied by (kernel_size - 1). This can be written as follows:

$$RF = 1 + \sum_{i=1}^n (k - 1) \times d_i$$

Here, k is the kernel size and d is the dilation factor.

Each cell in the hidden layer gets its value from a **convolutional operation**. This is essentially computing the weighted sum (or dot product) of the inputs with a learned filter (the kernel) and then passing the result through a non-linear activation function (ReLU). The asymmetric left padding (zero padding) ensures that the model remains causal and the output sequence has the same length as the input sequence.

The key features that make TCNs effective include the following:

- **Causal convolutions:** Each prediction at a given time step only depends on current and past inputs. This is a crucial property for time series, as it prevents **information leakage** from future steps. This is enforced by adding **asymmetric zero padding** at the beginning of the input sequence (left side), ensuring that convolutions only access current and past time steps. The left padding is calculated as $(k - 1) \times d$. So, in our diagram, hidden layer 1 has $(2 - 1) \times 1 = 1$ zeros of padding, hidden layer 2 has $(2 - 1) \times 2 = 2$ zeros of padding, and the connection of the final output layer has $(2 - 1) \times 4 = 4$ zeros of padding.
- **Dilated convolutions:** Dilated convolutions introduce gaps between kernel elements, allowing the model to skip over input time steps at regular intervals (look at inputs that are further apart). As we stack layers, we typically increase the dilation factor (e.g., 1, 2, 4, 8, ...), which causes the receptive field to also grow exponentially. This enables TCNs to see very far into the past with relatively few layers, allowing it to capture patterns over longer time horizons without increasing the model depth or parameters (a strategy popularized by **WaveNet** architectures).
- **Receptive field:** The receptive field represents the span of input time steps that influence a single output prediction. In the diagram, the orange-highlighted nodes at each layer show which time steps are being processed, while the receptive field spans 8 historical time steps ($t-7$ to t) that contribute to generating the forecast at time $t+1$ (shown in green at the top layer).
- **Residual connections:** Not shown in the diagram for simplicity, TCNs also use residual connections to create shortcut paths by adding the input of a layer to its output (a strategy popularized by **ResNet** architectures). It helps prevent the vanishing or exploding gradients problem and stabilizes training compared to recurrent networks. Note that each **residual block** typically contains two dilated causal convolution layers with normalization, ReLU activations, dropout or weight-norm, and a skip connection from input to block output. See *Figure 12.19*.

TCNs offer a powerful alternative to recurrent models such as LSTM, especially when training speed and stability are important. Their ability to model long-term dependencies without recurrent feedback loops makes them highly scalable and practical for modern time series forecasting tasks. In this recipe, we'll implement TCNs using Darts, building on the multivariate forecasting skills you developed in the previous recipe, *Handling multivariate time series and exogenous variables*.

Getting ready

Before starting, ensure that you have the following:

- All core libraries imported (e.g., torch, EarlyStopping from pytorch_lightning, and rmse and mae from Darts)
- The Scaler object (scaler_target) created
- The following preprocessed datasets:
 - **Training:** d_zen_train_sc (target variable, scaled), d_cov_train_sc (covariates, scaled)
 - **Validation:** d_zen_val_sc, d_cov_val_sc, d_zen_train_val_sc, and d_cov_train_val_sc
 - **Test:** d_zen_test (unscaled, reserved for final evaluation)

If you completed the previous recipes, these prerequisites should already be in place—no need to repeat setup code.

How to do it...

In this recipe, you will implement a TCN using Darts' TCNModel model. If you've followed our previous recipes, the process will feel familiar (whether using Darts or NeuralForecast). Both libraries streamline the time series forecasting workflow, making it easy to swap models with minimal code changes.

Let's dive in and build a TCN model, leveraging past covariates to predict future values, and evaluate its performance:

1. Start by importing the additional components for this recipe, mainly the TCNModel class:

```
from darts.models import TCNModel
```

2. You will use the same configurations we used in previous recipes for early stopping and learning rate scheduler:

```
PATIENCE=15
trainer_kwargs = make_trainer_kwargs(PATIENCE)
lr_scheduler_kwargs = make_lr_scheduler_kwargs(PATIENCE)
```

3. Create the TCNModel instance. You will configure the model with a 120-step input_chunk_length (meaning it looks at 120 past data points to make a prediction) and a 30-step output_chunk_length (our forecast horizon), similar to the previous recipes.
4. Key parameters such as kernel_size (filter width), num_filters (number of filters per layer), and dilation_base (skipping steps to longer patterns) control how the model learns. We will use the default parameter values as they are a good starting point. You can tweak them later for better performance. If you don't set num_layers (default is None), Darts automatically selects the minimum number of layers to ensure every output depends on the entire input window (full history coverage):

```

tcn_d_model = TCNModel(
    input_chunk_length=120,
    output_chunk_length=30,
    n_epochs=500,
    kernel_size=3, # width of the convolutional filter
    num_filters=3, # number of convolutional filters per layer
    dilation_base=2, # controls exponential dilation growth
    optimizer_kwargs={"lr": 0.001},
    weight_norm=True, # TCNModel default is False
    model_name="Zur_TCNModel",
    random_state=42,
    force_reset=True,
    save_checkpoints=True,
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs
)

```

5. There are a few key hyperparameters unique to `TCNModel`:
 - `kernel_size`: Controls the size (width) of the convolutional filters. Large values enable each filter to consider more consecutive time steps, which can help capture longer temporal patterns.
 - `num_filters`: Specifies the number of convolutional filters per layer. More filters allow the model to capture more patterns, but can increase computational cost.
 - `dilation_base`: Determines the exponential growth of the dilation factor across layers. A higher `dilation_base` increases the receptive field of the model more rapidly.
 - `weight_norm`: Indicates whether to use weight normalization, which helps stabilize training by normalizing the weights of convolutional layers.
6. Fit the model using past covariates. `TCNModel` inherits from `PastCovariatesTorchModel`, which means it only supports past covariates. This allows the model to leverage information available up to the prediction point to improve forecasts. This is similar to `BlockRNNModel` used earlier:

```

tcn_d_model.fit(
    series=d_zen_train_sc,
    past_covariates=d_cov_train_sc,
    val_series=d_zen_val_sc,
    val_past_covariates=d_cov_val_sc,
    verbose=True
)

```


7. Load the best model checkpoint with the lowest validation loss:

```
tcn_best_model = tcn_d_model.load_from_checkpoint(
    model_name="Zur_TCNModel", best=True
)
```

8. Generate the forecast and inverse transform the forecast to the original scale:

```
d_zen_train_val_sc = d_zen_train_sc.append(d_zen_val_sc)
d_cov_train_val_sc = d_cov_train_sc.append(d_cov_val_sc)

tcn_forecast_sc = tcn_best_model.predict(
    n=len(d_zen_test),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc)
tcn_forecast_inv = scaler_target.inverse_transform(tcn_forecast_sc)
```

9. Finally, evaluate model performance by computing RMSE and MAE to measure the forecast's accuracy against the test set:

```
rmse_val = rmse(d_zen_test, tcn_forecast_inv)
mae_val = mae(d_zen_test, tcn_forecast_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 744.4603
Darts Test Set RMSE: 1331.7764
```

10. The results are decent for our first run using default parameters (no hyperparameter tuning).

How it works...

You just implemented a TCN model using default parameter values and got decent forecasting results. The training was pretty fast, which is one of the key advantages of TCNs compared to recurrent models such as LSTMs.

In the recipe's introduction, we introduced TCN fundamentals such as dilated causal convolutions, receptive field, and asymmetric zero padding. To fully grasp how TCNs work, it's important to highlight another core innovation—the residual block and its skip connection:

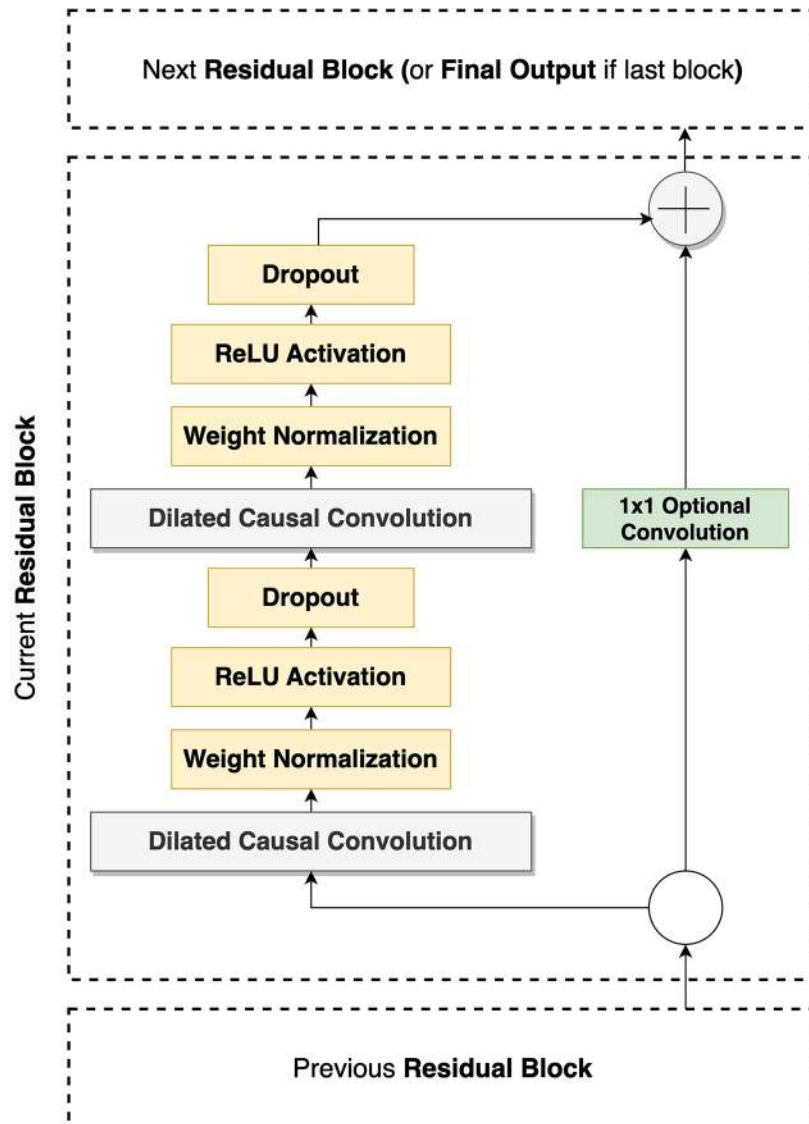


Figure 12.19: Architecture of a TCN residual block

The illustration in *Figure 12.19* shows the internal architecture of a TCN **residual block**. Each block contains a stack of two dilated causal convolutions, each followed by weight normalization (some implementations use batch normalization), ReLU activation, and dropout. A crucial skip connection (the arrow on the right) adds the block's input directly to its output, with an optional 1x1 convolution for dimension matching when needed. This residual path helps with gradient flow and enables the network to learn both simple and complex temporal patterns.

When we created `TCNModel`, we left `num_layers=None`, which allows Darts to automatically calculate the minimum number of layers (residual blocks) needed to ensure that every output timestamp can see the entire input window. In other words, the receptive field is large enough to cover our `input_chunk_length` value of 120.

Internally, Darts uses the following code to infer the number of layers:

```
import math
# Parameters from our model
input_chunk_length = 120
dilation_base = 2
kernel_size = 3

# Darts' internal calculation for num_layers
num_layers = math.ceil(
    math.log(
        (input_chunk_length-1) * (dilation_base-1) / (kernel_size-1) / 2 + 1,
        dilation_base,
    )
)

print(f"Calculated number of layers: {num_layers}")
>>
Calculated number of layers: 5
```

You can confirm that our model has five residual blocks using the following code:

```
len(tcn_best_model.model.res_blocks)
>>
5
```

A crucial concept for TCN is the **receptive field (RF)**, which determines how far back in time the model can “see” when making a prediction. For our model to be effective, we would like it to be as large as `input_chunk_length`. In the introduction section, we saw a general formula for TCN's receptive field. Darts' `TCNModel` class uses residual blocks, and each contains two convolutional layers, not just one.

This means we need to adapt the original formula to be more precise in order to calculate the receptive field:

$$RF = 1 + 2 \cdot (k - 1) \cdot \frac{d_b^L - 1}{d_b - 1}$$

Let's break this down:

- **RF:** Receptive field (in time steps)
- **L:** Number of layers (num_layers)
- **k:** Kernel size (kernel_size)
- **d_b:** Dilation base (dilation_base)

If we plug in our values (num_layers=5, kernel_size=3, and dilation_base=2), the receptive field (RF) would be 125 steps. This means that each output of the TCN at the top layer is able to see and is influenced by 125 preceding time steps in the input sequence. Any dependency or pattern that spans more than 125 past steps will not be fully captured by your network in its current configuration. This is greater than our input_chunk_length value of 120, so our model can see all the necessary history for every prediction it makes.

You can inspect the residual block at a specific layer. For example, to inspect the residual block at layer 2, you can use the following:

```
tcn_best_model.model.res_blocks[1]

>>
_ResidualBlock(
  (dropout1): MonteCarloDropout(p=0.2, inplace=False)
  (dropout2): MonteCarloDropout(p=0.2, inplace=False)
  (conv1): ParametrizedConv1d(
    3, 3, kernel_size=(3,), stride=(1,), dilation=(2,)
    (parametrizations): ModuleDict(
      (weight): ParametrizationList(
        (0): _WeightNorm()
      )
    )
  )
  (conv2): ParametrizedConv1d(
    3, 3, kernel_size=(3,), stride=(1,), dilation=(2,)
    (parametrizations): ModuleDict(
      (weight): ParametrizationList(
        (0): _WeightNorm()
      )
    )
  )
)
```

This should align with *Figure 12.18* in terms of the residual block architecture: we have two convolutional layers, two dropouts, and two weight normalizations.



In Darts, you will notice the dropout layer used is `MonteCarloDropout`. This is a special Darts layer used in training, and is designed to optionally remain active during prediction (inference) to help estimate forecast uncertainty. By default, this feature is off at inference, and in our recipe, it was not enabled; thus, the layer behaved exactly like a standard dropout layer, which is only active during training.

There's more...

Let's expand on the `MonteCarloDropout` (`mc_dropout`) layer we saw earlier to understand its role in probabilistic forecasting. Previously, you computed single-value metrics such as RMSE and MAE:

- **MAE:** The average of the absolute difference between the predicted and the actual values. It tells us, on average, how much our prediction is off from the actual. For example, an MAE of 747 means our forecast is, on average, 747 units away from the actual value.
- **RMSE:** The square root of the average of the squared difference between forecast and actual values. RMSE penalizes larger errors more due to squaring, so a high RMSE (e.g., 1333) indicates some large deviations in the forecast.

When using `TCNModel` with `num_samples` and `mc_dropout`, you are employing **Monte Carlo (MC)** sampling during inference. This means that at prediction time, dropout layers remain active (unlike standard inference, in which dropout layers are turned off), thus producing an ensemble of forecasts at prediction time. These forecasts represent different network configurations for each forward pass, thus creating different forecast variations representing the model's predictive uncertainty.

With this approach, you generate a distribution of possible future forecasts, rather than a single forecast. While you can still compute MAE and RMSE on the mean or median of MC dropout forecasts, these metrics cannot evaluate the quality of prediction intervals or quantify forecast uncertainty. This is where **quantile loss** comes in.

Quantile loss measures how accurately your forecast predicts a specific quantile (e.g., 5th, 50th, or 95th percentile) of the distribution of possible outcomes. It helps us evaluate the quality of our prediction intervals and how well our forecasts are at a given quantile compared to actual observed values. For the 50th percentile (the median), the quantile loss is equivalent to MAE.

By using Monte Carlo dropout, you will be generating a full range of possible future outcomes. This can help you have a better understanding of the model as opposed to a single forecast. Let's see this through an example leveraging `mc_dropout` and `num_samples`:

```
# check activation function
tcn_prob = tcn_best_model.predict(
    n=len(d_zen_test),
    num_samples=200,
    mc_dropout=True, # Enable MC Dropout
    series=d_zen_train_val_sc,
```

```

past_covariates=d_cov_train_val_sc,
random_state=1)
tcn_prob_inv = scaler_target.inverse_transform(tcn_prob)
tcn_prob_inv.shape
>>
(30, 1, 200)

```

Our forecast TimeSeries object has a shape of (timesteps, components, samples). This means it contains 30 time steps (our 30-day forecast horizon), 1 component (the energy series), and 200 individual forecast scenarios generated by the model (each reflects a different MC dropout scenario).

You can visualize the spread of your forecasts by plotting all samples or, more commonly, by plotting prediction intervals (e.g., the area between the 5th and 95th percentiles):

```

plt.figure(figsize=(16, 5))
d_zen_test.plot(label='Actual', linestyle='--', alpha=0.65)
tcn_prob_inv.plot(label='Forecast',
                  color='k',
                  linestyle='--',
                  alpha=0.25,
                  low_quantile=0.05, # default value for lower bound
                  high_quantile=0.95, #default value for upper bound
                  central_quantile=0.5 # default value (median forecast))
plt.title('Darts Zurich Energy TCN Probabilistic Forecast vs Actuals (Test Set)')
plt.xlabel('Day')
plt.ylabel('Energy Consumption')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()

```

Figure 12.20 illustrates the TCN model's probabilistic forecast on the test set and the 90% prediction interval against actual energy consumption values.

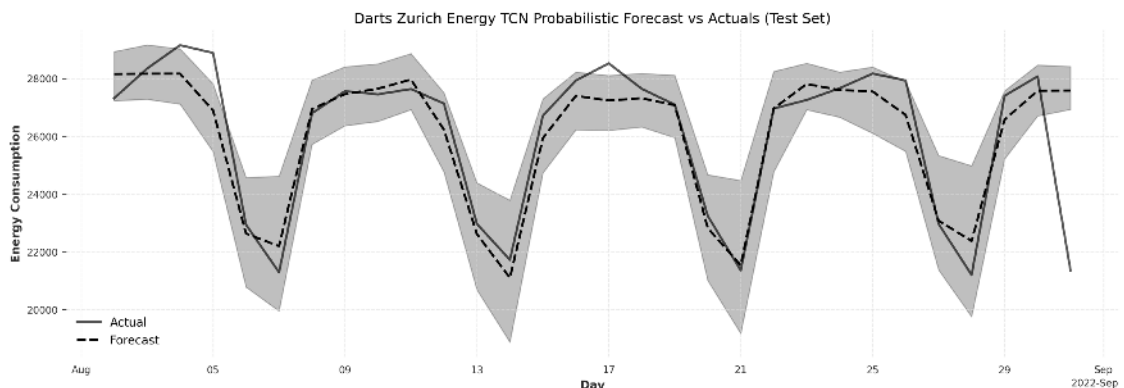


Figure 12.20: TCN model forecast with Monte Carlo dropout

The dashed forecast line represents the **median** predictions (central quantile for all 200 forecasts). The shaded area reflects uncertainty (the spread) derived from the 200 prediction simulations using Monte Carlo dropout. Remember, this is happening during prediction time, and we are not retraining the model. The Monte Carlo dropout is what is giving us these variations in forecasting.

Darts provides **quantile loss (QL)** and **mean quantile loss (MQL)**. We will compute the MQL for different quantiles:

```
from darts.metrics import mql
# Calculate the loss for the lower bound (5th percentile)
loss_p05 = mql(d_zen_test, tcn_prob_inv, q=0.05)

# Calculate the loss for the median (50th percentile)
# This value will be identical to your MAE score
loss_p50 = mql(d_zen_test, tcn_prob_inv, q=0.50)

# Calculate the loss for the upper bound (95th percentile)
loss_p95 = mql(d_zen_test, tcn_prob_inv, q=0.95)

print(f"Average Quantile Loss at 5th percentile: {loss_p05:.4f}")
print(f"Average Quantile Loss at 50th percentile (MAE): {loss_p50:.4f}")
print(f"Average Quantile Loss at 95th percentile: {loss_p95:.4f}")
>>
Average Quantile Loss at 5th percentile: 516.3424
Average Quantile Loss at 50th percentile (MAE): 755.3407
Average Quantile Loss at 95th percentile: 237.9668
```

The MAE of our median forecast is 755.34, providing a baseline measure of central tendency accuracy. However, we gain additional insight by comparing the interval bounds (opposite tails). The QL for the lower bound (5th percentile) is 516.34, which is more than double the loss of the upper bound (95th percentile) at 237.97. This indicated that the model's lower quantile is overestimating, meaning actual values fall below the predicted 5th percentile more often than they should.

In a well-calibrated model (meaning its confidence levels are reliable), the QLs of the lower and upper bounds (e.g., 5th and 95th percentiles) should be approximately equal, indicating balanced prediction intervals.

Our current results indicate an imbalance (asymmetry), which suggests opportunities for model improvement, more specifically, in better calibrating the lower tail of the forecast distribution.

See also

- To learn more about TCN, you can read the paper titled *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*: <https://arxiv.org/pdf/1803.01271>

- To learn more about `TCNModel`, you can read the official documentation here: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.tcn_model.html

Forecasting with Transformers (N-HiTS)

In *Chapter 10*, you explored models such as the **Theta model**, which decomposes a time series into short-term and long-term trends. In *Chapter 11*, you were introduced to **ensemble learning** approaches, such as **gradient boosting** and **XGBoost**, where models are stacked and each new model tries to correct the mistakes of the previous ones. These foundational concepts, decomposition and residual learning, inspired some of the most popular neural approaches in time series forecasting.

One innovation is N-BEATS, which decomposes a time series using neural networks stacked in sequence. Each block receives the residuals from the previous block, allowing the model to iteratively refine its prediction. A key feature in N-BEATS is that each block produces dual outputs:

- **Forecast (future expansion):** Predicting future values based on a predefined forecast horizon.
- **Backcast (backward expansion):** Attempts to reconstruct the input signal. This helps the model better learn from historical data by minimizing reconstruction error.

Building on the foundation of N-BEATS is N-HiTS, which introduces two major innovations:

- **Multi-rate data sampling:** Unlike N-BEATS, which relies on **fully connected (FC)** blocks, each N-HiTS block combines **MaxPooling** (to downsample and reduce the signal's resolution, allowing the model to focus on dominant patterns) with a **multi-layer perceptron (MLP)**. This structure allows each block in the stack to specialize in different time series components at various scales and frequencies. The pooling kernel size controls how much the signal is downsampled, and each block uses a different kernel size, allowing different blocks in the stack to capture patterns at multiple temporal resolutions. Like N-BEATS, each block produces both a **forecast** and a **backcast**.
- **Hierarchical interpolation:** Since different blocks in N-HiTS generate forecasts at varying frequencies, the model needs a way to blend and assemble these multi-scale outputs into a single, cohesive forecast. N-HiTS accomplishes this through **hierarchical interpolation** by **upsampling** the lower-resolution (coarser) forecasts and sequentially integrating them with the higher-resolution (finer) forecasts. This process creates sharper and more detailed predictions by blending information from different levels. While backcasts are primarily used for internal learning, it's the hierarchical merging of forecasts that yields a more accurate and detailed overall output.

Together, these advances make N-HiTS one of the most effective and flexible architectures for time series forecasting, particularly for datasets with complex, multi-scale patterns. One of the great advantages of N-BEATS and N-HiTS is that they are **global models** that can be trained across multiple time series.

In this recipe, you'll implement N-HiTS using the Darts library (`NHiTSModel`).

Getting ready

This recipe builds on the earlier recipe, *Handling multivariate time series and exogenous variables*. Before starting, ensure that you have the following:

- All core libraries imported (e.g., torch, EarlyStopping from pytorch_lightning, and rmse and mae from Darts)
- The Scaler object (scaler_target) created
- The following preprocessed datasets:
 - **Training:** d_zen_train_sc (target variable, scaled) and d_cov_train_sc (covariates, scaled)
 - **Validation:** d_zen_val_sc, d_cov_val_sc, d_zen_train_val_sc, and d_cov_train_val_sc
 - **Test:** d_zen_test (unscaled, reserved for final evaluation)

If you completed the previous recipes, these prerequisites should already be in place—no need to repeat setup code.

How to do it...

1. Start by importing the additional components for this recipe, mainly the NHiTSModel class:

```
from darts.models import NHiTSModel
```

2. You will use the same configurations we used in previous recipes for early stopping and learning rate scheduler:

```
PATIENCE=15
early_stopper = EarlyStopping(
    monitor="val_loss",
    patience=PATIENCE,
    min_delta=0.001,
    mode='min',
    verbose=True)

trainer_kwargs = {
    "callbacks": [early_stopper],
    "enable_progress_bar": True,
    "accelerator": "auto"}

lr_scheduler_kwargs = {
    "mode": "min",
    "monitor": "val_loss",
    "factor": 0.2,
    "patience": PATIENCE//3,
```

```
"min_lr": 1e-6
}
```

3. Create the `NHiTSModel` instance. You will configure the model with a 120-step `input_chunk_length` (meaning it looks at 120 past data points to make a prediction) and a 30-step `output_chunk_length` (our forecast horizon), similar to the previous recipes.

Key parameters such as `num_blocks`, `num_stacks`, `num_layers`, `pooling_kernels_sizes`, and `n_freq_downsample` were kept using default parameter values. They make for a good starting point. You can tweak them later for better performance (e.g., hyperparameter tuning). If you don't explicitly set `pooling_kernels_sizes` and `n_freq_downsample` (their default is `None`), Darts automatically computes their values based on `input_chunk_length` and `output_chunk_length`:

```
nhits_d_model = NHiTSModel(
    input_chunk_length=120,
    output_chunk_length=30,
    n_epochs=500,
    num_blocks=1, # NHiTSModel default
    num_stacks=3, # NHiTSModel default
    num_layers=2, # NHiTSModel default
    pooling_kernel_sizes=None, # NHiTSModel default
    n_freq_downsample=None, # NHiTSModel default
    optimizer_kwargs={"lr": 0.001},
    model_name="Zur_NHiTSModel",
    random_state=42,
    force_reset=True,
    save_checkpoints=True,
    lr_scheduler_cls=torch.optim.lr_scheduler.ReduceLROnPlateau,
    lr_scheduler_kwargs= lr_scheduler_kwargs,
    pl_trainer_kwargs=trainer_kwargs
)
```

4. There are a few key hyperparameters unique to `NHiTSModel`:
 - `num_blocks` (default is 1): Number of blocks per stack (controls block depth/complexity).
 - `num_stacks` (default is 3): Number of hierarchical decomposition levels (stack depth).
 - `num_layers` (default is 2): Number of hidden layers in each block's MLP.
 - `n_freq_downsample` (default is `None`): Downsampling factor for each block.
 - `pooling_kernels_size` (default is `None`): Pooling kernel size per block. Larger kernels smooth out high-frequency details to focus on long-term trends, while smaller kernels preserve fine-grained patterns for short-term forecasting.

5. Fit the model using past covariates. `NHITSModel` inherits from `PastCovariatesTorchModel`, which means it only supports past covariates. This allows the model to leverage information available up to the prediction point to improve forecasts. This is similar to `BlockRNNModel` used earlier:

```
nhits_d_model.fit(
    series=d_zen_train_sc,
    past_covariates=d_cov_train_sc,
    val_series=d_zen_val_sc,
    val_past_covariates=d_cov_val_sc,
    verbose=True
)
```

6. Load the best model checkpoint with the lowest validation loss:

```
nhits_best_model = nhits_d_model.load_from_checkpoint(
    model_name="Zur_NHITSModel", best=True
)
```

7. Generate the forecast and inverse transform the forecast to the original scale:

```
d_zen_train_val_sc = d_zen_train_sc.append(d_zen_val_sc)
d_cov_train_val_sc = d_cov_train_sc.append(d_cov_val_sc)

nhits_forecast_sc = nhits_best_model.predict(
    n=len(d_zen_test),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc)

nhits_forecast_inv = scaler_target.inverse_transform(nhits_forecast_sc)
```

8. Finally, evaluate model performance by computing RMSE and MAE to measure the forecast's accuracy against the test set:

```
rmse_val = rmse(d_zen_test, nhits_forecast_inv)
mae_val = mae(d_zen_test, nhits_forecast_inv)
print(f"Darts Test Set MAE: {mae_val:.4f}")
print(f"Darts Test Set RMSE: {rmse_val:.4f}")
>>
Darts Test Set MAE: 672.2498
Darts Test Set RMSE: 1221.2898
```

9. The results are decent for our first run using default parameters (no hyperparameter tuning).

How it works...

You just implemented an N-HiTS model using default parameter values and got better scores compared to TCN in the previous recipe, with the added benefit of faster training. This makes N-HiTS even more attractive for practical forecasting.

Earlier, we discussed the innovations introduced by N-BEATS and further improved by N-HiTS. Here, we will dive deeper using *Figure 12.21*.

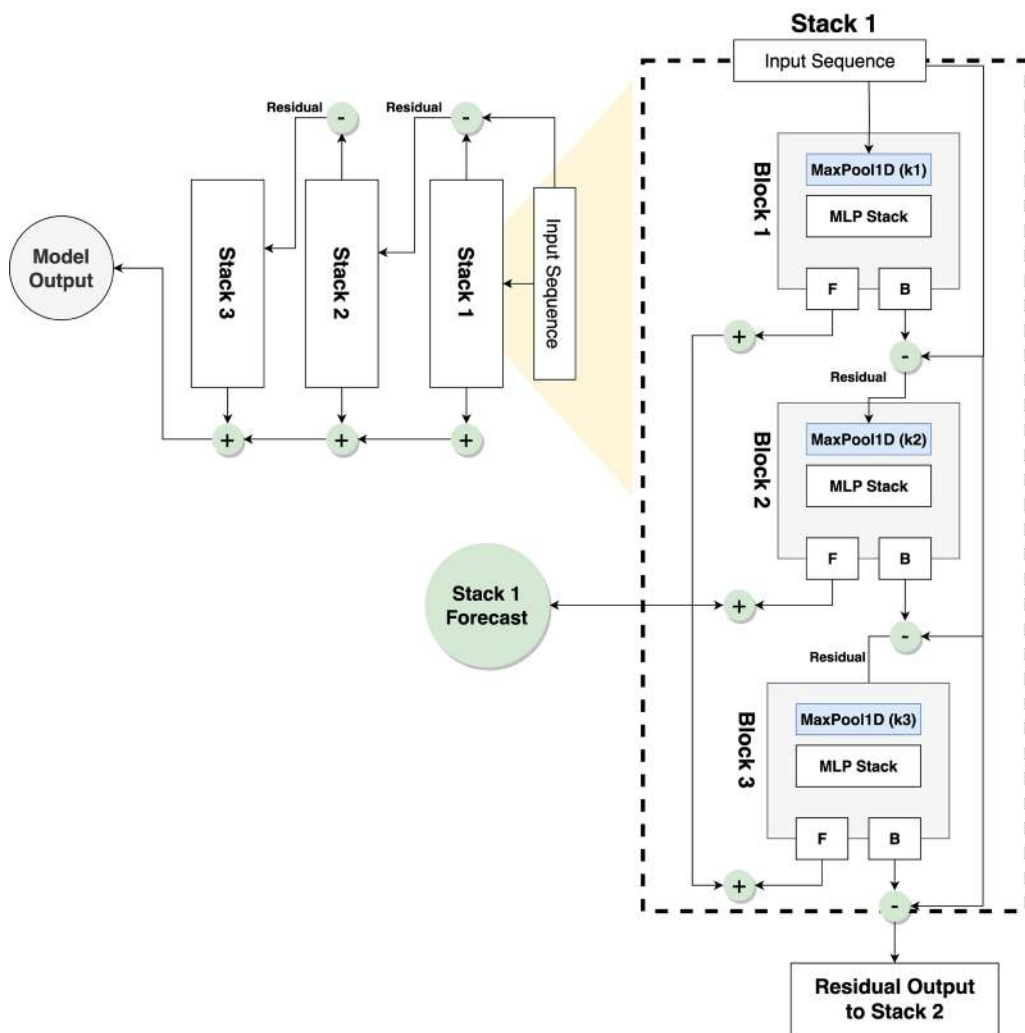


Figure 12.21: N-HiTS architecture showing hierarchical stacks with residual connections and forecast aggregation

The architecture in *Figure 12.21* represents three stacks, each with three blocks. In our implementation, we used three stacks, each with one block, but the illustration shows a more general case to help explain the core ideas.

On the left of the diagram, you see a high-level architecture: a sequence of stacks (*Stack 1 to Stack 3*). Each stack produces a backcast and a forecast. The backcast (backward expansion) is an attempt to reconstruct the input it received, which is then used to compute the residual by subtracting it from the input values. This residual, which represents patterns not yet captured by the current stack, is passed as input for the next stack. The idea with this residual connection is to allow the next stack to focus on capturing insights and patterns that were probably missed by the prior stack. As each stack learns, it produces a forecast (future expansion), which is then combined to produce a global forecast (the output). This process allows each stack, which focuses on different aspects of the input sequence, to contribute to the final output.

Now, inside each stack, a similar process happens internally between the blocks that make up the stack. On the right side of *Figure 12.21*, since it is part of the first stack, the original time series sequence is the input, which goes through a layer that combines `MaxPool1D` with an MLP. Each block produces a backcast (backward expansion) and a forecast (future expansion). The backcast of one block gets passed as a residual to the next block within the same stack, ensuring that each subsequent block focuses on patterns not yet learned by its predecessor. The forecasts from all blocks within the stack are combined to form the stack's overall forecast. The backcast output from the last block in a stack becomes the stack's overall backcast, ready to be passed as a residual to the subsequent main stack.

The innovation within N-HiTS lies in how it handles the signal's scale and frequencies. First, each block utilizes `MaxPool1D` with potentially different kernel sizes (`k1`, `k2`, `k3`, etc.). The `pooling_kernel_sizes` parameter essentially resamples (downsamples) the input data differently for each stack. This variation in input granularity forces different stacks to focus on distinct frequency components.

Second, the MLP in each block does not directly predict all future points. Instead, it predicts a smaller set of coefficients. The `n_freq_downsample` parameter controls how many of these coefficients are produced relative to `output_chunk_length`.

When N-HiTS attempts to reconstruct the forecasts (to get back to the full `output_chunk_length`), N-HiTS uses hierarchical output interpolation. The combination of multi-rate input sampling and hierarchical output interpolation is what makes N-HiTS computationally efficient and accurate, similar to a blend of residual and ensemble learning.

When we left `pooling_kernel_sizes=None`, Darts used automatic kernel pooling size calculation based on the following:

```
import numpy as np
input_chunk_length = 120
num_blocks = 1
num_stacks = 3
max_v = max(input_chunk_length // 2, 1)
pooling_kernel_sizes = tuple(
    (int(v),) * num_blocks
    for v in max_v // np.geomspace(1, max_v, num_stacks)
)
pooling_kernel_sizes
```

```
>>
((60,), (7,), (1,))
```

The `pooling_kernel_sizes` parameter controls how much the input is pooled (downsampled) at each block in each stack. Looking at the code output, *Stack 1* pools very aggressively (kernel size 60), *Stack 2* moderately (kernel size 7), and *Stack 3* not at all (kernel size 1), which means each stack analyzes the data at a distinct level of smoothness. This directly contributes to the **multi-rate sampling**, allowing different stacks to focus on varying frequency components.

Similarly, for `n_freq_downsample=None`, Darts used automatic downsampling coefficients calculation based on the following:

```
output_chunk_length = 30
max_v = max(output_chunk_length // 2, 1)
n_freq_downsample = tuple(
    (int(v),) * num_blocks
    for v in max_v // np.geomspace(1, max_v, num_stacks))
n_freq_downsample
>>
((15,), (3,), (1,))
```

The `n_freq_downsample` parameter controls how much the outputs are downsampled before interpolation for each block in each stack. More specifically, it sets the dimensionality of the outputs (the number of coefficients) produced by the MLP in each block, before these are interpolated back to the full forecast.

In this example, *Stack 1* produces only 15 coefficients for its forecast (heavily downsampled), *Stack 2* produces 3, and *Stack 3* produces 1 (no downsampling). After these coefficients are predicted by each block, they are upsampled (interpolated) back to the original output length using linear interpolation.

Darts will use these values to introduce variety in each stack, as shown:

Stack	Blocks per Stack	Pooling Kernel Size	n_freq_downsample
1	1	60	15
2	1	7	3
3	1	1	1

Table 12.3: Auto-calculated pooling and downsampling parameters in *NHiTSModel*

There's more...

In addition to forecasting models, it is critical to know how our model would have performed historically. Instead of just a single test evaluation, historical forecasting helps us better understand how consistent our model is at different time periods, and how well the model can handle different conditions or seasonal patterns through the different slices in time. Generally, it offers a more robust performance assessment than just a single holdout test.

Darts offers **backtesting** tools such as `historical_forecasts()` and `backtest()`, which we will explore here.

The `historical_forecasts()` method can simulate how the trained model would have predicted at many points in the past, moving forward through our data in a sliding manner. The core idea here is to see how our fixed, unchanging model would have performed if it had been deployed at various historical moments, without ever being retrained on new data. This helps us pinpoint periods of struggle, where our model's fixed parameters (already trained) can adapt to different inputs.

This is achieved by calling the `historical_forecasts()` method from your model:

```
nhits_historical_forecasts_sc = nhits_best_model.historical_forecasts(
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc,
    start=len(d_zen_train_sc), # where to start
    forecast_horizon=30,      # Predict 30 steps ahead
    stride=1,                 # Slide window by 1 each time
    retrain=False,
    last_points_only=False # Get forecasts for all points, not just the last
)
```

Here, Darts starts after our training set, slides 1 step each time, with a forecast horizon of 30 days.

The returned `nhits_historical_forecasts_sc` object is a list of `TimeSeries` objects, with one forecast window for each rolling origin (a total of 131 full 30-step forecasts):

```
len(nhits_historical_forecasts_sc)
>>
131
```

If you set `last_points_only=True`, you would only get the final point from each window: a list of 131 items, where each item represents the final point in the window.

You can calculate rolling-origin evaluation; in other words, you calculate the MAE for each window and then compute the average of these individual scores:

```
from darts.metrics import mae

# Reverse the scaling for forecasts and targets
nhits_historical_forecasts_inv = [scaler_target.inverse_transform(forecast)
                                for forecast in nhits_historical_forecasts_sc]

# Gather ground truth for comparison
# Let's get the corresponding "actual" future values for each forecast window
actuals = []
for i, forecast in enumerate(nhits_historical_forecasts_sc):
    # The forecast starts at the same time as forecast
```

```

actual_values = d_zen_train_val_sc.slice(
    forecast.start_time(), forecast.end_time()
)
actuals.append(scaler_target.inverse_transform(actual_values))

# Compute MAE for each window, then average
window_maes = [mae(a, f) for a, f in zip(actuals,
                                         nhits_historical_forecasts_inv)]

average_mae = sum(window_maes) / len(window_maes)

print(f"Average rolling MAE over all windows: {average_mae:.2f}")
>>
Average rolling MAE over all windows: 983.42

```



The rolling MAE (983.42) is higher than our single test evaluation (672.25) because it includes predictions made on different validation periods, some of which may be more challenging to forecast. The average rolling MAE gives a more stable and representative measure of its overall performance across various historical periods.

Darts also provides the `backtest()` method, which does a similar thing as `historical_forecasts()` by internally running a rolling-origin forecasting. The difference is that instead of returning actual forecasts, it computes and returns the errors between the forecasts and the true values at each window:

```

mae_scores = nhits_best_model.backtest(
    metric=mae,
    reduction=None,
    last_points_only=False,
    stride=1,
    start=len(d_zen_train_sc),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc,
    forecast_horizon=30,
    retrain=False
)

```

The `backtest()` method is more efficient when you only need performance metrics instead of the actual forecast values.

Note that we are using scaled data for `backtest()` since our model was trained on scaled inputs. If you use unscaled data, the results will not be good. Maintaining this consistency between training and evaluation is crucial for the model to perform correctly. This means the resulting `mae_score` value will be in scaled units, which is acceptable for computing and comparing model performance across different validation windows.

Like `historical_forecasts()` with `last_points_only=False`, `backtest()` also returns a list:

```
len(mae_scores)
>>
131
```

Each value in `mae_scores` is the MAE for one of the 131 forecast windows.

Since we have a distribution of MAE scores, you can plot them using a histogram to visualize the variability of the model's performance:

```
from darts.utils.statistics import plot_hist
plot_hist(mae_scores, bins=20)
```

This should produce the following plot:

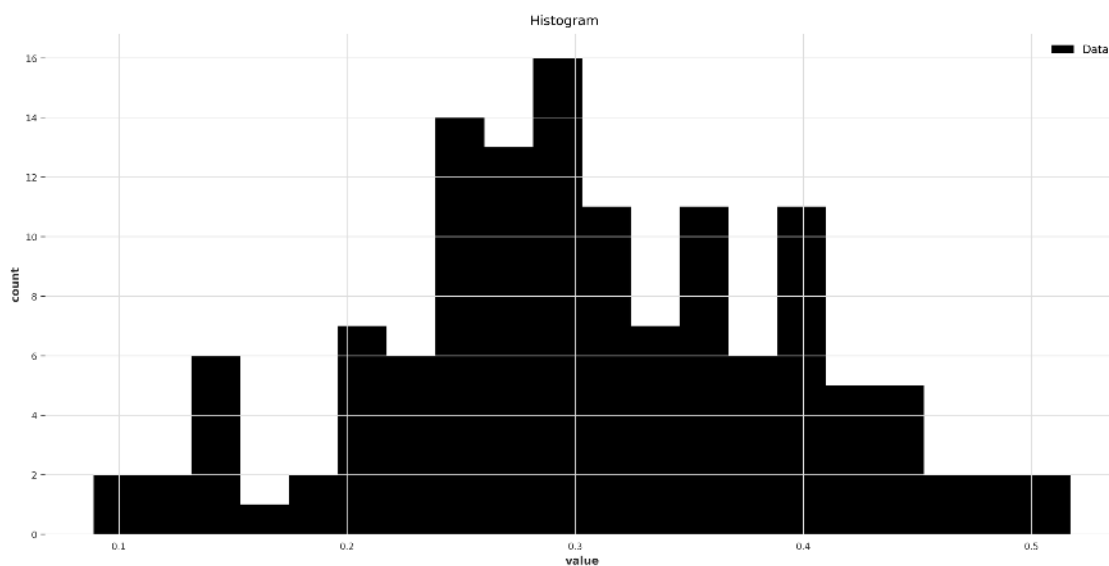


Figure 12.22: Distribution of MAE scores across 131 rolling forecast windows showing model performance variability

The histogram in Figure 12.22 helps identify whether the model performs consistently (narrow distribution) or has high variability (wide distribution) across different time periods.

If you want `backtest()` to return just a single aggregate error value, then you can use the `reduction` parameter and provide an aggregation method. In this example, we will use mean aggregation (`np.mean`):

```
avg_score = nhits_best_model.backtest(
    metric=mae,
    reduction=np.mean,
    last_points_only=True,
    stride=1,
    start=len(d_zen_train_sc),
    series=d_zen_train_val_sc,
    past_covariates=d_cov_train_val_sc,
    forecast_horizon=30,
    retrain=False
)
avg_score
>>
np.float32(0.30740514)
```

The average scores provide a more concise summary of the model's overall performance over all historical windows. The average backtest MAE of 0.302 is in scaled units, which provides a standardized measure of model performance.

Another neat feature in Darts is that it provides several plotting options, including `plot_residual_analysis`. We have explored residual analysis and its importance in *Chapters 10* and *11*. Residual analysis helps us ensure that there are no patterns in the residuals that our model may have missed (random residuals suggest a good model fit). Given that N-HiTS internally uses backcasting (as residuals), it makes natural sense to evaluate the residuals:

```
from darts.utils.statistics import plot_residuals_analysis
plot_residuals_analysis(
    nhits_best_model.residuals(
        retrain=False,
        series=d_zen_train_val_sc,
        past_covariates=d_cov_train_val_sc
    )
)
```

This will produce three insightful subplots: a time plot of the residuals, a histogram of their distribution (ideally bell-shaped), and an **autocorrelation function (ACF)** plot to check for any remaining patterns.

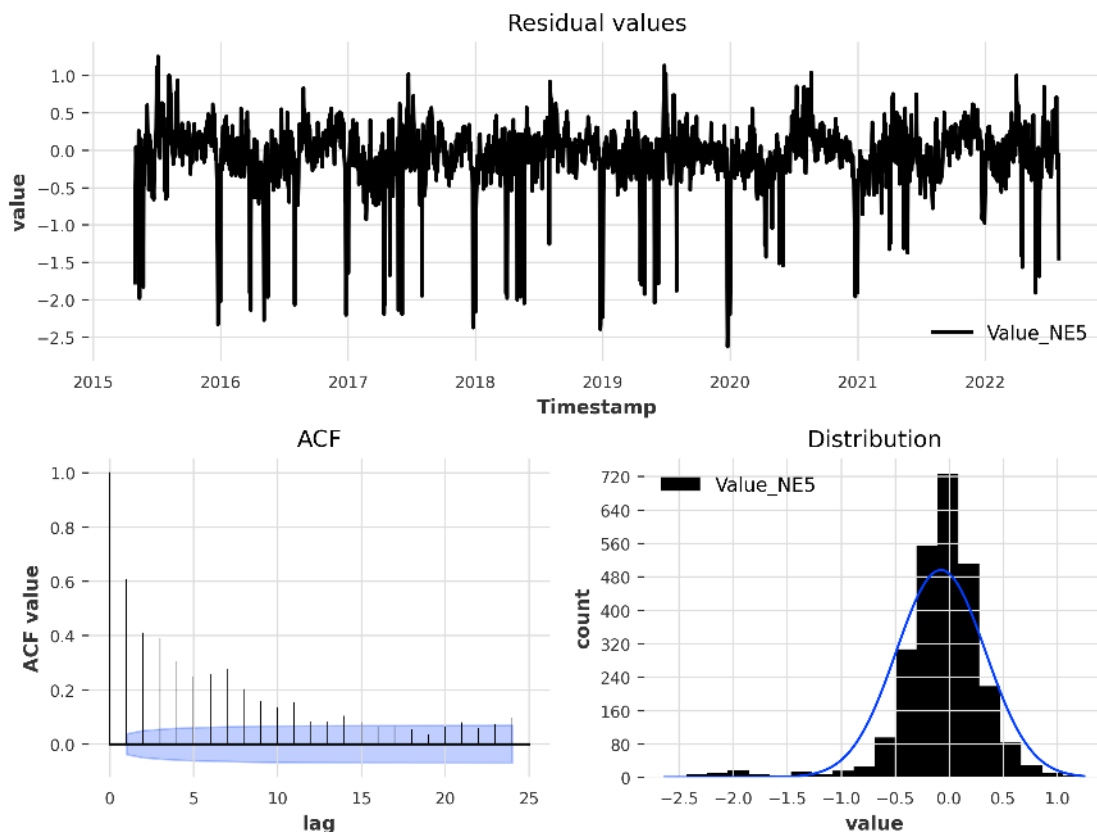


Figure 12.23: N-HiTS residual analysis showing (top) residuals over time, (bottom left) ACF plot for temporal correlation, and (bottom right) residual distribution for normality assessment

See also

- To learn more about N-HiTS, you can read the paper titled *N-HiTS: Neural Hierarchical Interpolation for Time Series Forecasting*: <https://arxiv.org/pdf/2201.12886>
- To learn more about Darts' NHiTSModel class, you can read the official documentation here: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.nhits.html

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.gg/jSmGeJ4rP5>



13

Outlier Detection Using Unsupervised Machine Learning

In *Chapter 7*, you explored parametric and non-parametric statistical techniques to spot potential outliers. These methods are simple and interpretable, yet quite effective.

Outlier detection remains challenging, mainly due to the ambiguity surrounding what constitutes an outlier in your specific data or the problem that you are trying to solve. This ambiguity arises because what is considered an outlier can vary depending on the context, such as the presence of trends, seasonality, or domain-specific patterns. For example, though commonly used, the statistical thresholds from *Chapter 7* are somewhat arbitrary rather than definitive rules. This is why domain knowledge or access to **Subject Matter Experts (SMEs)** is vital to making the proper judgment when identifying time series outliers.

The following table summarizes the outlier detection techniques for different anomaly types that were discussed in *Chapter 7*:

Anomaly Type	Definition	Techniques	Key Characteristics
Point Outliers	Individual data points that deviate significantly from the global distribution without considering the temporal context	Tukey's Method (IQR) z-score and Modified z-score STRAY	Compare against global statistics Treat observation as independent Does not consider temporal relationships Flags extreme values regardless of context
Contextual Outliers	Data points that are anomalous specifically within their local temporal context, but might be normal in the global distribution	Hampel Filter	Uses a sliding window approach Compare points to local neighborhood statistics Consider temporal proximity Can detect seasonally unusual values

Collective Outliers	Groups or sequences of data points that together form an anomalous pattern, even if individual points might not be extreme	Matrix Profile	Analyze subsequences rather than individual points Detect unusual patterns over time Identify anomalous periods or regimes Can detect subtle pattern changes

Table 13.1: Summary of outlier detection techniques used in Chapter 7

Point outlier detection is suitable when you are looking for obvious outliers irrespective of temporal patterns, where these outliers manifest as extreme individual values. It's a good choice when simplicity and interpretability are priorities. **Contextual outlier detection** is appropriate when local temporal context matters (e.g., weekly patterns), as the same value might be normal in one context but anomalous in another. This approach works well when you need to account for trends or seasonality. **Collective outlier detection** is best when you are interested in unusual patterns or sequences. These outliers or anomalies manifest as changes in behavior over multiple time points. This is a good choice when individual points might not be extreme, but their combination (collectively) is unusual.

While statistical methods are simple and interpretable, machine learning techniques can capture more complex patterns in the data, especially when dealing with high-dimensional or non-linear relationships. These advanced techniques offer additional flexibility when statistical approaches fall short, particularly in complex time series with multiple variables or intricate patterns. The right technique depends on your specific use case and the type of anomalies you want to detect in your time series data.

In this chapter, you will be introduced to a handful of machine learning and deep learning-based methods for outlier detection. Most of the machine learning techniques for outlier detection are considered *unsupervised* outlier detection methods, such as **K-Nearest Neighbors (KNN)**, **Local Outlier Factor (LOF)**, **Copula-Based Outlier Detection (COPOD)**, **Isolation Forests (iForest)**, and **AutoEncoder**, to name a few. **Unsupervised machine learning** algorithms identify patterns or anomalies in data without explicit instructions.

Generally, outliers (or anomalies) are considered rare occurrences within your dataset. In a large collection of data, they typically constitute only a small fraction of observations—perhaps around 1% or even less. This rarity makes outliers inherently challenging to detect, requiring specialized methods designed to find these unusual patterns or observations. Later in the chapter, you will see this concept referred to as the **contamination** percentage. The contamination parameter essentially represents the expected proportion of outliers in your dataset. This parameter is used by the machine learning algorithms to adjust their sensitivity when identifying these unusual data points.

After investigating outliers, you will have a historical set of labeled data, allowing you to leverage semi-supervised outlier detection techniques. Unlike the unsupervised methods we'll focus on in this chapter (which detect patterns without prior labels), semi-supervised approaches use a small amount of labeled data (known outliers) to improve detection accuracy on new observations. This chapter focuses on unsupervised outlier detection, which requires no labeled examples to get started.

In this chapter, you will be introduced to the PyOD library, described as “*a comprehensive but easy-to-use Python library for detecting anomalies in multivariate data.*” The library offers an extensive collection of implementations for popular and emerging algorithms in the field of outlier detection, which you can read about here: <https://github.com/yzhao062/pyod>. Even though PyOD was not specifically designed for time series, the library provides efficient implementations that can be adapted to temporal data through techniques we’ll explore in this chapter.

You will be using the same New York taxi dataset to make it easier to compare the results between the different machine learning methods in this chapter and the statistical methods from *Chapter 7*.



In the literature, you will encounter both *anomaly detection* and *outlier detection*—terms often used interchangeably. While the methods to identify these unusual observations are similar, the difference typically lies in the context and the actions after identification. Throughout this chapter, both *outlier* and *anomaly* may be used to refer to the same underlying concept of finding unusual patterns in time series data, so you can consider these terms equivalent as you read.

The recipes that you will encounter in this chapter are as follows:

- Detecting outliers using distance-based algorithms
- Detecting outliers using clustering-based algorithms
- Detecting outliers using probabilistic and statistical algorithms
- Detecting outliers using kernel-based algorithms
- Detecting outliers using ensemble method algorithms
- Detecting outliers using deep learning-based algorithms (AutoEncoder)

Technical requirements

You can download the Jupyter notebooks and datasets required from the GitHub repository:

- **Jupyter notebooks:** https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/blob/main/code/Ch13/Chapter_13.ipynb
- **Datasets:** <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch13>



Creating a new Python environment when installing new libraries such as PyOD is always a good idea. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup recipe* from *Chapter 0* on GitHub. The chapter covers two methods: using conda and venv.

You can install PyOD with either pip or conda. For a pip install, run the following command:

```
pip install pyod
```


To prepare for the outlier detection recipes, start by loading the libraries that you will be using throughout the chapter:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
from utils import plot_outliers

plt.rcParams["figure.figsize"] = [12, 5]
```

You will be working with the same dataset used in *Chapter 7*. You can refer to the *Technical Requirements* section of *Chapter 7* for more information regarding the dataset.

Load the `nyc_taxi.csv` data into a pandas DataFrame, as it will be used throughout the chapter:

```
file = Path("../datasets/Ch13/nyc_taxi.csv")
nyc_taxi = pd.read_csv(file,
                      index_col='timestamp',
                      parse_dates=True)

nyc_taxi.index.freq = '30min'
```

Store the known dates containing outliers, also known as ground truth labels:

```
nyc_dates = [
    "2014-11-01",
    "2014-11-27",
    "2014-12-25",
    "2015-01-01",
    "2015-01-27"
]
```

The `plot_outliers` function is designed to visualize time series data with highlighted outliers, making it easier to compare the results of different outlier detection methods.

As you proceed with the outlier detection recipes, the goal is to see how the different techniques capture outliers and compare them to the ground truth labels, as follows:

```
tx = nyc_taxi.resample('D').mean()
known_outliers = tx.loc[nyc_dates]
plot_outliers(known_outliers, tx, 'Known Outliers', labels=True)
```

The preceding code should produce a time series plot with *X* markers for the known outliers:

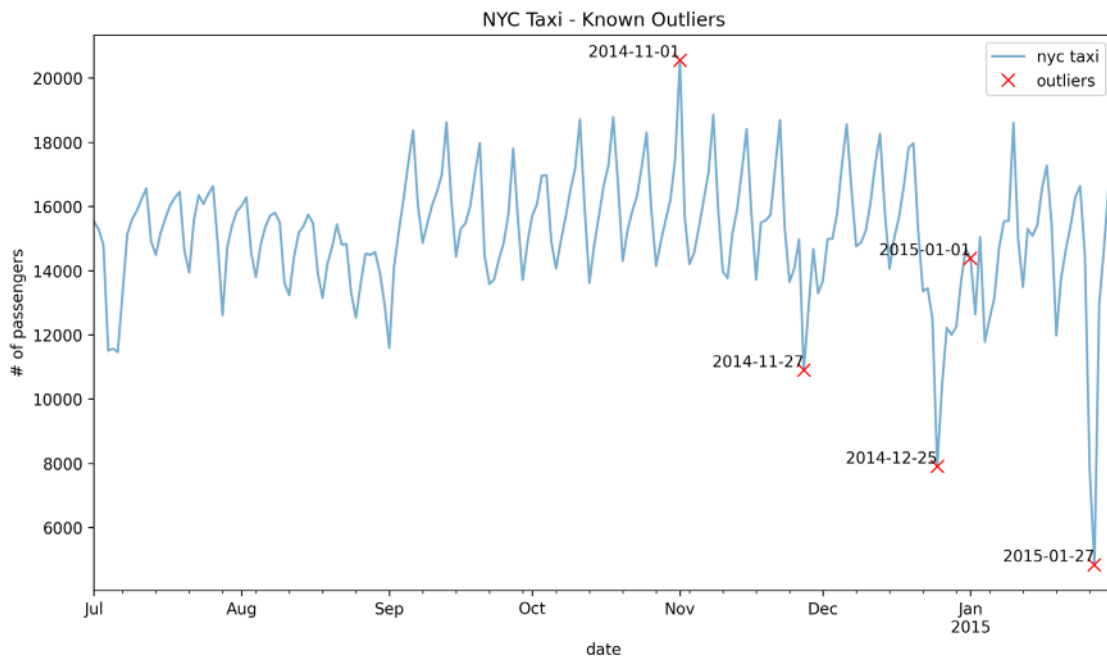


Figure 13.1: Plotting the NYC taxi data after downsampling with ground truth labels (outliers)

Understanding considerations for time series outlier detection

Outlier detection in time series has its unique challenges compared to static data, primarily due to its inherent temporal dependencies between consecutive data points. Temporal dependencies mean that the value at any given time point is often influenced by previous values, thus making it important to consider the sequence of data points rather than treating them as independent observations. The presence of trend and seasonality in time series data can further complicate outlier detection. As you may recall, outliers in time series can manifest in different forms: **point outliers** (individual extreme values), **contextual outliers** (values unusual within a specific time period), and **collective outliers** (groups of points exhibiting abnormal behavior).

Several outlier detection algorithms were not explicitly designed for time series data; this includes the algorithms available in **PyOD** and the **scikit-learn** library. They are not designed to capture temporal patterns in the data or to account for seasonality and trends.

Using these algorithms without any transformations to the data or the inclusion of a sliding window means you are focusing on capturing extreme outlier values regardless of the temporal context. In other words, using these algorithms without transformation essentially performs **point outlier** detection. If our data exhibits trend and seasonality, one approach is to decompose the time series into its constituent components (trend, seasonality, and residuals) and apply outlier detection on the residuals (after removing trend and seasonality).

Applying these algorithms to the residuals, after removing predictable patterns, allows us to focus on outliers that deviate from the expected temporal pattern, potentially catching both point and **contextual outliers**.

When transforming the data by including **lagged features**, or adding other time-dependent features such as cyclical encoding, seasons, and so on, you are adding temporal context as additional features. This allows the algorithm to consider the relationship of a point to its surrounding points in time, and therefore capture **contextual outliers**.

There are libraries such as **aeon** that have implementations based on PyOD algorithms that are more compatible with time series. Even better, **aeon** includes the **PyODAdapter** class, which wraps any PyOD algorithm to make them context aware and suitable for time series data by incorporating a sliding window (`window_size` and `stride` parameters). Sliding windows involve dividing the time series into overlapping segments, allowing the algorithm to analyze patterns over a fixed time period. This approach is particularly useful for detecting anomalies that span multiple time points, such as collective outliers.

In this chapter, you will explore PyOD for **point outliers** (using the algorithms as is), using PyOD after **decomposition** to start to address **contextual outliers**. You will then move into more focused contextual outlier detection by enhancing our time series with temporal features through feature engineering, such as adding lagged features, and lastly, using **PyODAdapter**.



To avoid repetition, the detailed implementation of these transformation techniques will be provided in the first recipe. In subsequent recipes, we will then focus on using PyOD as is for point outlier detection.

The following are the techniques you will explore in the first recipe:

- **Detecting point outliers:** This is the simplest approach, where outliers are identified based on extreme values without considering temporal context
- **Detecting outliers after decomposition:** By removing trend and seasonality, we can focus on residuals to identify anomalies that deviate from expected patterns
- **Detecting contextual outliers with a sliding window:** This approach considers temporal context by analyzing patterns over a fixed time period
- **Detecting contextual outliers with feature engineering:** Adding temporal features (e.g., lagged values, cyclical encoding, holidays) helps the algorithm understand the context better
- **Detecting contextual outliers with PyODAdapter:** This method adapts PyOD algorithms for time series data by incorporating sliding windows

Several PyOD algorithms are wrappers to scikit-learn with additional functionality. These include **GMM** (Gaussian Mixture Model), which wraps **GaussianMixture**; **IForest** (Isolation Forest), which wraps **IsolationForest**; **LOF** (Local Outlier Factor), which wraps **LocalOutlierFactor**; and **OCSVM** (One-Class SVM Detector), which wraps **OneClassSVM**.



Most PyOD algorithms are designed for unsupervised learning scenarios where no labeled examples are available. However, some algorithms can be adapted to semi-supervised or supervised settings when labeled data is available. In this chapter, we will focus on unsupervised learning algorithms.

PyOD offers 45 outlier detection methods, including 12 modern neural models (PyTorch). The PyOD algorithms can be classified as follows:

Category	Learning Type	Key Algorithms	Best For
Statistical and distance-based	Unsupervised	KNN, LOF, ABOD, HBOS, ECOD, COPOD	Identifying observations that deviate from their neighbors or from statistical distributions
Clustering-based	Unsupervised	CBLOF	Finding observations that don't belong to any cluster or form small clusters
Linear model-based	Unsupervised	PCA, MCD, OCSVM (linear kernel)	Detecting observations that deviate from learned linear relationships or boundaries
Ensemble methods	Unsupervised	IForest, FeatureBagging, LSCP, XGBOD	Combining multiple detectors for improved robustness and accuracy
Deep learning-based	Unsupervised	AutoEncoder, DeepSVDD	Learning complex non-linear representations for identifying anomalies
Semi-supervised	Semi-supervised	DevNet, XGBOD	Leveraging a small amount of labeled data to improve detection

Table 13.2: Categorization of some key algorithms in PyOD

Table 13.2 categorizes PyOD algorithms based on their learning type and key characteristics. This can help you choose the right algorithm for your specific use case, depending on whether you are looking for global outliers, local outliers, or complex patterns.

It is worth mentioning that another open source library, **DeepOD**, offers deep learning-based outlier detection that supports both tabular and time-series anomaly detection and offers a similar API to PyOD. You can learn more about DeepOD here: <https://deepod.readthedocs.io/en/latest/index.html>.



PyOD's methods for training and making predictions

Like scikit-learn, PyOD offers familiar methods for training your model and making predictions. The standard approach is using `model.fit()` for training followed by `model.predict()` for predictions. While `model.fit_predict()` is currently available, it's important to note that this method is deprecated since version 0.6.9 and will be removed in future releases..

In addition, PyOD provides these additional prediction methods: `predict_proba()` for probability scores, `decision_function()` for raw outlier scores, and `predict_confidence()` for model confidence in predictions.

In our first recipe, you will explore how to implement these approaches using PyOD's comprehensive suite of algorithms, starting with understanding how PyOD handles training and prediction. This foundational recipe will introduce fundamental concepts (e.g., `contamination`, `threshold_`, and `decision_scores_`) and demonstrate various adaptation techniques for time-series data, including using PyOD as is, applying algorithms to residuals, incorporating lagged and temporal features, and utilizing PyODAdapter.

Detecting outliers using distance-based algorithms

In this recipe, we focus on distance-based outlier detection algorithms, which identify anomalies by measuring the proximity between data points. Specifically, we will explore two powerful distance-based techniques: **K-Nearest Neighbors (KNN)** and **Local Outlier Factor (LOF)**. Both algorithms calculate distances between points to determine outliers, but with different approaches. KNN identifies **global outliers** by calculating the absolute distance to the *k*-nearest neighbors, while LOF identifies **local outliers** by comparing the local density (derived from distance measurements) of a point to its neighbors. This density-based approach of LOF makes it particularly effective at finding outliers in datasets with varying densities.

KNN for global outlier detection

The KNN algorithm is typically used in a supervised learning setting where prior results or outcomes (labels) are known. It can be used to solve classification or regression problems. The idea is simple; for example, you can classify a new data point, *Y*, based on its nearest neighbors. For instance, if *k*=5, the algorithm will find the five nearest data points (neighbors) by distance to point *Y* and determine its class based on the majority in the case of classification, or compute the average for regression.

KNN can be effectively adapted for unsupervised outlier detection. In this context, KNN identifies outliers by measuring the **distances** between a data point and its *k* nearest neighbors. Points that are significantly farther from their neighbors are more likely to be flagged as outliers. The algorithm works by doing the following:

- Finding the **k** nearest neighbors to each data point
- Calculating the average (or median) distance to these neighbors
- Identifying points with significantly larger distances as **global outliers**

In essence, the distance becomes the scoring mechanism to determine which points are outliers among the population; hence, it is a **proximity-based algorithm**.

The choice of distance metric is a crucial parameter that significantly affects how outliers are identified. PyOD's default is the **Minkowski** distance, but you can specify alternatives such as **Euclidean** (l2) or **Manhattan** (l1) distances using the `metric` parameter.

Once the distance is calculated, then you need to determine a method for aggregating these distances for the *k* neighbors (scoring). PyOD's KNN implementation supports three scoring methods through the `method` parameter:

- `'largest'` (default): Uses the distance to the *k*th neighbor as the outlier score
- `'mean'`: Uses the average distance to all *k* neighbors as the outlier score
- `'median'`: Uses the median of distances to all *k* neighbors as the outlier score

The choice of method affects how the algorithm aggregates the distance information from the neighbors.

KNN's strength lies in identifying points that are globally distant from the majority of the data. However, it may struggle with identifying outliers in datasets with varying densities or clusters, which is where LOF comes in.

LOF for local outlier detection

While KNN works well for global outliers, it may struggle with local outliers in dense regions. This is where the **Local Outlier Factor (LOF)** algorithm becomes valuable. Instead of relying on absolute distance between neighboring points, LOF uses the concept of local density as the basis for scoring data points and detecting outliers (considering density variations). The LOF is considered a **density-based** algorithm. The idea behind the LOF is that outliers will be further from other data points and more isolated, and thus will be in low-density regions.

Similar to KNN, the LOF algorithm relies on finding the *k* nearest neighbors and calculating a **distance** metric.

By combining these two approaches, you can address a broader range of outlier detection scenarios. Generally, you would choose KNN when you expect outliers to be far from the majority of data points, and LOF when you suspect outliers in low-density regions within clusters. Let's illustrate the difference between the two using a concrete example.

Imagine you are visiting your local coffee shop on a busy weekday morning. Inside, there's a line of customers waiting to order, and they're standing relatively close to each other. Outside in the parking lot, there's a person sitting in their car, waiting for the line to shorten before going in.

KNN perspective

The KNN algorithm measures distances between people. Someone waiting in their car in the parking lot would be considered an outlier because they're physically far from everyone else. KNN simply calculates the distance from each person to their *k* nearest neighbors and flags those with the largest distances as outliers. The person in the car will have a higher score because their distance to their nearest neighbor (people at the end of the line) is much greater than the typical distances between people standing in the line.

LOF perspective

LOF goes deeper by considering local density. The people in line represent a high-density area where everyone is similarly close to their neighbors. The person in the car is in a low-density area. What makes LOF special is that it considers **reachability**—from the perspective of people in line, the person in the car is not easily reachable, even though that person can see everyone in line. This **inverse reachability** concept (how far are you from your neighbor’s perspective, not just yours) helps LOF identify local outliers that KNN might miss. For example, if there were two distinct lines forming in the coffee shop with different densities, KNN might struggle to identify outliers within the less dense line, while LOF would recognize density variations in both local contexts.

This example illustrates two key principles:

- KNN is better at finding **global outliers**, identifying outliers that are distant from the main data distribution, regardless of local patterns
- LOF excels at finding outliers in datasets, identifying outliers within datasets with varying densities and clusters

How to do it...

In this recipe, you will explore four different techniques when using PyOD algorithms. In later recipes, we will stick with the **point outlier technique**, but the same techniques presented here can be reused across the remaining recipes.

Start by loading the necessary classes for the recipe:

```
from pyod.models.knn import KNN
from pyod.models.lof import LOF
```

You will work with the `tx` DataFrame, created in the *Technical requirements* section.

Detecting point outliers

The simplest and most straightforward way to use PyOD is as is. This, in essence, is effective at capturing point outliers—those that are extreme values compared to the rest—without any consideration of temporal dependencies, seasonality, or trend.

There are some key parameters you need to be familiar with before we start our recipe, as they help control the algorithm’s behavior. One of these parameters is `contamination`, a numeric (float) value representing the dataset’s fraction of outliers. This is a common parameter across all the different algorithms in PyOD. For example, a `contamination` value of `0.1` indicates that you expect 10% of the data to be outliers. The default value is `contamination=0.1`. The `contamination` value can range from `0` to `0.5` (or 50%). You will need to experiment with the `contamination` value, since the value influences the scoring threshold used to determine potential outliers, and how many of these potential outliers are to be returned. You will learn more about this in the *How it works...* section.

For example, if you suspect the proportion of outliers in your data to be 3%, then you can use that as the contamination value. You could experiment with different contamination values, inspect the results, and determine how to adjust the contamination level. We already know that there are 5 known outliers out of the 215 observations (around 2.3%), and in this recipe, you will use 0.03 (or 3%).

Other parameters, specific to KNN and LOF, include the following:

- **n_neighbors:** An integer representing the number of neighbors to use. For KNN, the default is `n_neighbors=5`, while for LOF the default is `n_neighbors=20`. Generally, you will want to experiment with different values of `k` (number of neighbors). You can fit KNN models with varying `k` values and compare the results to determine the optimal number of neighbors for the dataset.
- **metric:** A string representing the distance metric used for computing the distance (more on that in the *how it works...* section). The default for both KNN and LOF is the 'minkowski' distance, but it can be any distance metric from scikit-learn or the SciPy library.

The method is specific to KNN, which describes the aggregation method across the distances. It defaults to `method='largest'` but can be 'mean' or 'median'. In this recipe, you will use the mean—the average of all `k` neighbor distances.

1. Instantiate KNN and LOF with the updated parameters:

```
knn = KNN(contamination=0.03,
           method='mean',
           n_neighbors=5)
print(knn)
>>
KNN(algorithm='auto', contamination=0.03, leaf_size=30, method='mean',
    metric='minkowski', metric_params=None, n_jobs=1,
    n_neighbors=5, p=2, radius=1.0)

lof = LOF(contamination=0.03,
           n_neighbors=20)
print(lof)
>>
LOF(algorithm='auto', contamination=0.03, leaf_size=30,
    metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=20,
    novelty=True, p=2)
```

2. Fit KNN and LOF on the `tx` DataFrame:

```
knn.fit(tx)
lof.fit(tx)
```


3. Use the predict method to flag outliers by generating binary labels for each data point (1 if True or 0 if False). Store the results in a pandas Series:

```
knn_pred = pd.Series(knn.predict(tx), index=tx.index)
lof_pred = pd.Series(lof.predict(tx), index=tx.index)

print('Number of KNN outliers = ', knn_pred.sum())
print('Number of LOF outliers = ', lof_pred.sum())
>>
Number of KNN outliers = 6
Number of LOF outliers = 6
```

4. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from knn_pred
knn_outliers = knn_pred[knn_pred == 1]
knn_outliers = tx.loc[knn_outliers.index]
print(knn_outliers)
>>

              value
timestamp
2014-11-01  20553.500000
2014-11-27  10899.666667
2014-12-25   7902.125000
2014-12-26  10397.958333
2015-01-26   7818.979167
2015-01-27   4834.541667

# extract the outlier values and dates from lof_pred
lof_outliers = lof_pred[lof_pred == 1]
lof_outliers = tx.loc[lof_outliers.index]
print(lof_outliers)
>>

              value
timestamp
2014-11-01  20553.500000
2014-11-27  10899.666667
2014-12-25   7902.125000
2014-12-26  10397.958333
2015-01-26   7818.979167
2015-01-27   4834.541667
```

Both LOF and KNN produced the same results, given the parameters used. Note that results would vary if the parameters were changed (e.g., `n_neighbors`, `metric`). Overall, the results look promising; four out of the five known dates have been identified. Additionally, both algorithms identified the day after Christmas as well (January 26, 2015), which was when all vehicles were ordered off the street due to the North American blizzard.

5. Use the `plot_outliers` function created in the *Technical requirements* section to visualize the output to gain better insight:

```
plot_outliers(knn_outliers, tx, 'KNN')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except that the *X* markers are based on the outliers identified using the KNN algorithm:

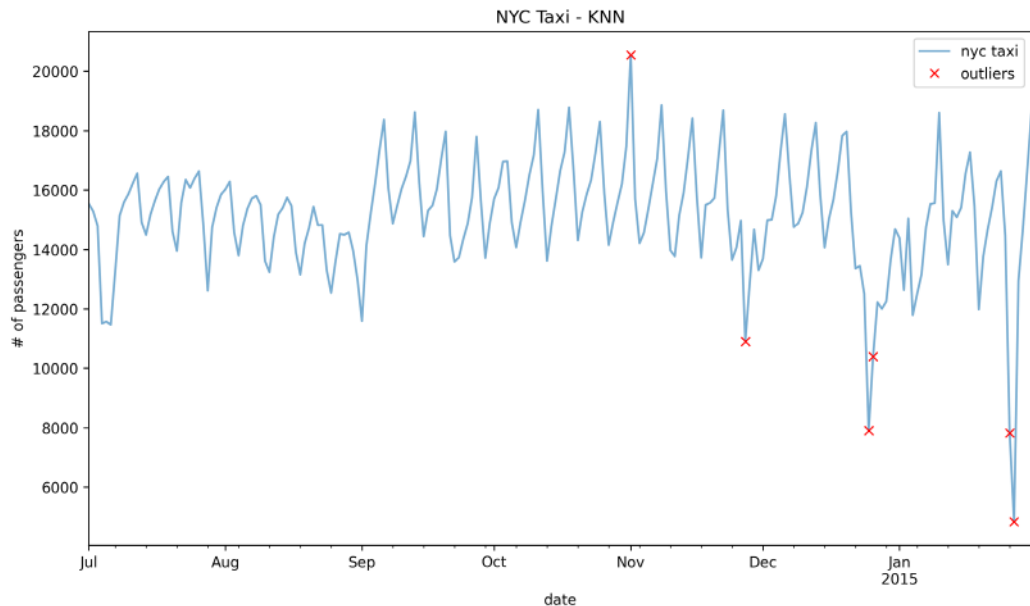


Figure 13.2: Markers showing the identified potential outliers using the KNN algorithm

You can plot the outliers from LOF using `plot_outliers(lof_outliers, tx, 'LOF')`, which should produce a similar plot to that shown in *Figure 13.2*.

Detecting outliers after decomposition

Time-series decomposition can be a useful preprocessing step before applying outlier detection algorithms. Decomposition involves breaking down the time series into its trend, seasonal, and residual components. By focusing on the residuals (the remaining variation after removing trend and seasonality), we can better identify outliers that significantly deviate from the expected patterns.

You were introduced to time-series decomposition in the *Decomposing time series data* recipe in Chapter 8. In this recipe, you will use **STL** from `statsmodels`:

1. Start by importing the STL class from `statsmodels`:

```
from statsmodels.tsa.seasonal import STL
```

2. Decompose the time series into seasonal, trend, and residual components. You will focus on capturing the residuals:

```
# For daily data with weekly seasonality
stl = STL(tx, seasonal=7)
result = stl.fit()
# Focus on residuals
residuals = result.resid.dropna().to_frame()

residuals.plot(title='Residuals after STL decomposition');
```

The `seasonal=7` parameter specifies weekly seasonality, which is appropriate for this daily taxi data. Choosing the right seasonality period is crucial for effective decomposition and outlier detection.

The preceding code should produce a residual plot, as shown in *Figure 13.3*:

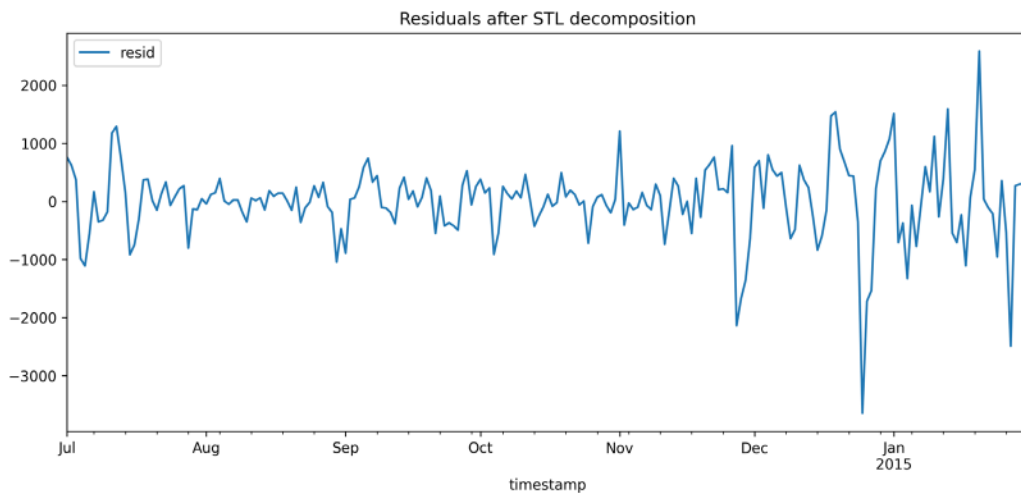


Figure 13.3: Residual plot after STL decomposition on the average daily passenger taxi data

The residual plot in *Figure 13.3* shows the remaining variation after removing trend and seasonality. Outliers in this plot represent deviations from the expected patterns.

3. Instantiate KNN and LOF with the same parameters as the previous section, then fit the models on the residuals, and use the `predict` method to identify outliers in the residuals:

```

knn_decomp = KNN(contamination=0.03, method='mean', n_neighbors=5)
lof_decomp = LOF(contamination=0.03, n_neighbors=20)

# Apply PyOD to residuals
knn_r = knn_decomp.fit(residuals)
lof_r = lof_decomp.fit(residuals)

knn_pred_r = pd.Series(knn_decomp.predict(residuals),
                       index=residuals.index)

lof_pred_r = pd.Series(lof_decomp.predict(residuals),
                       index=residuals.index)

>>
Number of KNN outliers after Decomposition = 4
Number of LOF outliers after Decomposition = 4

```

4. Filter the outputs to only show the outlier values and associated dates:

```

# extract the outlier values and dates from knn_pred_r
knn_outliers_r = knn_pred_r[knn_pred_r == 1]
knn_outliers_r = tx.loc[knn_outliers_r.index]
print(knn_outliers_r)

>>
           value
timestamp
2014-11-27  10899.666667
2014-12-25   7902.125000
2015-01-20  13759.416667
2015-01-27   4834.541667

# extract the outlier values and dates from lof_pred_r
lof_outliers_r = lof_pred_r[lof_pred_r == 1]
lof_outliers_r = tx.loc[lof_outliers_r.index]
print(lof_outliers_r)

>>
           value
timestamp
2014-11-27  10899.666667
2014-12-25   7902.125000
2015-01-20  13759.416667
2015-01-27   4834.541667

```

The code outputs the outlier values and their corresponding dates for both KNN and LOF. Both KNN and LOF identified the same four outliers because the residuals after decomposition are relatively simple, with clear deviations from the expected patterns. Notice how decomposition helped identify three known outliers (2014-11-27, 2014-12-25, and 2015-01-27), while also detecting 2015-01-20, which wasn't in our known outlier list.

5. Visualize the outputs to gain better insight using the `plot_outliers` function. The following example is for KNN:

```
plot_outliers(  
    knn_outliers_r, tx,  
    'KNN on residuals after decomposition - Original Taxi Plot',  
    labels=True  
)
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except that the X markers are based on the outliers identified using the KNN algorithm.

To better understand how these outliers appear in the residual component, we can visualize them directly on the residual plot:

```
res = residuals.loc[knn_pred_r[knn_pred_r == 1].index]  
res.columns = ['value']  
plot_outliers(  
    res, residuals,  
    'KNN on residuals after decomposition - Residual Plot',  
    labels=True  
)
```

This should display a similar plot to *Figure 13.1*, but instead of the markers being plotted on the original data, they are plotted on the residual plot:

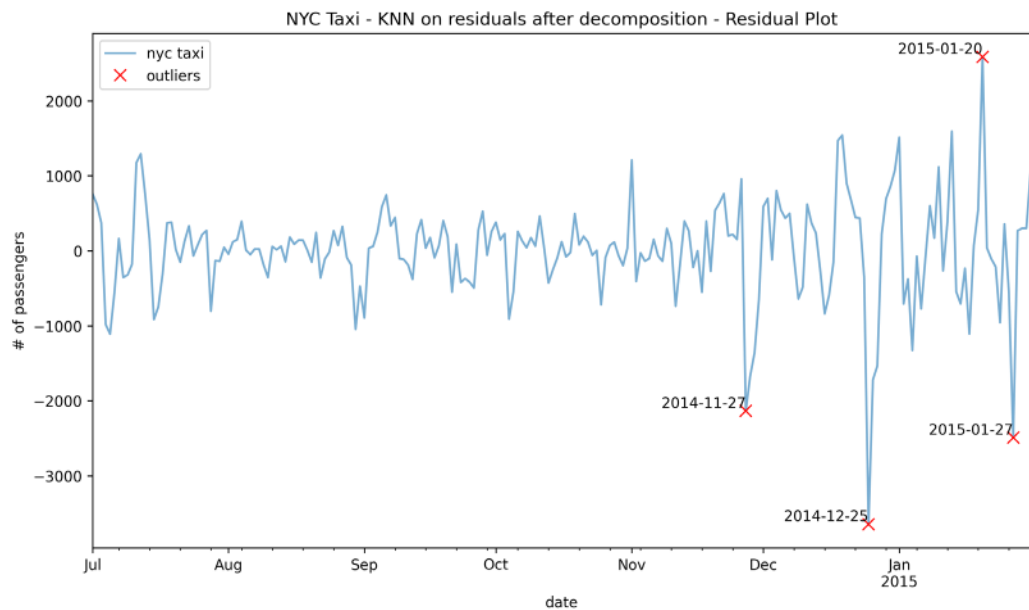


Figure 13.4: Residual plot with markers showing extreme residuals using the KNN algorithm

Figure 13.4 shows extreme residuals that represent significant deviations from expected behavior.

Applying outlier detection to residuals rather than raw data can be more effective because residuals represent the *unexpected* variation in your time series. By removing trend and seasonality through decomposition, you eliminate predictable patterns that might mask or distort outliers. For example, a value that appears normal in raw data might actually be abnormal when considering its expected seasonal pattern. The residuals highlight these deviations from expected behavior, making contextual outliers more apparent to detection algorithms that weren't designed to account for temporal patterns.

Detecting contextual outliers with a sliding window

Using PyOD as is on the time series proved to be effective in our case for identifying extreme data points (point outliers). The algorithms treated each observation independently and did not consider temporal ordering or relationships between consecutive observations.

To make the PyOD algorithm more suitable for time series data, we need to perform additional pre-processing. Recall in *Chapter 11*, you created the `generate_lagged_features` function to transform our time series into a tabular format suitable for supervised machine learning. You will use the same function, with minor changes, to accomplish the same thing: transforming our time series into a tabular data format with lagged features. The main difference is, since we are using it in an unsupervised context, we will not need to include the target column `y`.

1. You will implement the `create_sliding_window` function to transform the time series data into sliding windows. In essence, it is very similar to the `generate_lagged_features` function used in *Chapter 11* to add lagged features.

```
def create_sliding_windows(df, window_size):
    """Transform time series data into sliding windows for
       anomaly detection.
    """
    # Validate input
    if not isinstance(window_size, int) or window_size < 1:
        raise ValueError("Window size must be a positive integer")

    # Convert DataFrame to 1D array
    d = df.values.squeeze()

    # Create sliding windows using numpy's efficient implementation
    windows = np.lib.stride_tricks.sliding_window_view(
        d, window_shape=window_size)[-1]

    # Create column names for positions within the window
    cols = [f'pos_{i}' for i in range(window_size)]

    # Create DataFrame with windows
    windows_df = pd.DataFrame(windows, columns=cols,
                              index=df.index[window_size:])

    return windows_df
```

2. Apply the `create_sliding_windows` function to the `tx` DataFrame. Generally, you will need to experiment with the `window_size` parameter but for this activity we will go with a 7-day sliding window.

```
tx_sw = create_sliding_windows(tx, window_size=7)
print(tx_sw.shape)
>>
(208, 7)
```

A 7-day window is chosen to capture weekly seasonality patterns, as taxi ridership typically follows weekly cycles with different patterns on weekdays versus weekends. Different window sizes may be appropriate for other time series with different periodicity.

3. Instantiate KNN and LOF with the same parameters as the previous section, then fit the models on the transformed dataset (tx_sw), and use the predict method to highlight potential outliers.

```
knn_sw = KNN(contamination=0.03, method='mean', n_neighbors=5)
lof_sw = LOF(contamination=0.03, n_neighbors=20)

knn_sw.fit(tx_sw)
lof_sw.fit(tx_sw)

knn_pred_sw = pd.Series(knn_sw.predict(tx_sw), index=tx_sw.index)
lof_pred_sw = pd.Series(lof_sw.predict(tx_sw), index=tx_sw.index)

print('Number of KNN outliers = ', knn_pred_sw.sum())
print('Number of LOF outliers = ', lof_pred_sw.sum())
>>
Number of KNN outliers = 4
Number of LOF outliers = 6
```

Observe how we are starting to see a difference between KNN and LOF, unlike our point outliers in the previous section.

4. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from knn_pred_sw
knn_outliers_sw = knn_pred_sw[knn_pred_sw == 1]
knn_outliers_sw = tx.loc[knn_outliers_sw.index]
print(knn_outliers_sw)
>>
           value
timestamp
2015-01-28  12947.562500
2015-01-29  14686.145833
2015-01-30  16676.625000
2015-01-31  18702.479167

# extract the outlier values and dates from lof_pred_sw
lof_outliers_sw = lof_pred_sw[lof_pred_sw == 1]
lof_outliers_sw = tx.loc[lof_outliers_sw.index]
print(lof_outliers_sw)
>>
           value
timestamp
2014-12-26  10397.958333
2015-01-27   4834.541667
```


2015-01-28	12947.562500
2015-01-29	14686.145833
2015-01-30	16676.625000
2015-01-31	18702.479167

Interestingly, KNN did not identify any of the known outlier dates, but picked up on dates during the North American Blizzard. On the other hand, LOF picked 2015-01-27 (a known outlier date), as well as the day before and the day following that event. Surprisingly, LOF also captured the day after Christmas.

5. Visualize the outputs to gain better insight using the `plot_outliers` functions. The following example is for LOF:

```
plot_outliers(lof_outliers_sw, tx, 'LOF Sliding Window')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the X markers are based on the outliers identified using the LOF algorithm after applying a 7-day sliding window:

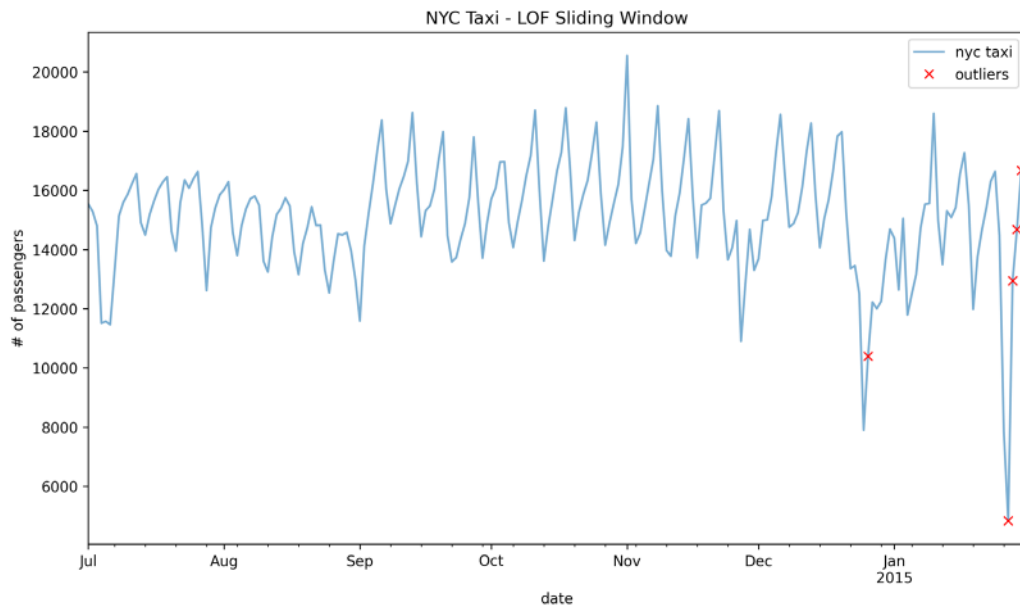


Figure 13.5: Markers showing the identified potential outliers using the LOF algorithm on transformed data

Notice how LOF identified a sequence of consecutive days (2015-01-27 through 2015-01-31) that were affected by the North American Blizzard, demonstrating how contextual outlier detection can identify anomalous periods rather than just individual points.

In this section, KNN and LOF produced different results when applied to the sliding window data. This difference arises because KNN evaluates absolute distances in the feature space created by the sliding window, while LOF's density-based approach is more sensitive to local patterns within these windows. This explains why LOF detected the known outlier (2015-01-27) that KNN missed, highlighting the value of using different algorithmic approaches when searching for contextual outliers.

Detecting contextual outliers with feature engineering

In *Chapter 11*, you were introduced to a feature engineering technique in the Forecasting with exogenous variables and ensemble learning in *sktime* recipe. You will be using the same techniques (adding cyclical encoding, holidays, etc.), but implemented as a function.

1. Start by creating the `add_time_features` function, which creates additional temporal-based features for the time series data. This function adds cyclical encoding (for weekly and monthly patterns), year, incremental counter for trend, U.S. holidays, month start/end indicators, and flags for weekends and working days.

```
import holidays

def add_time_features(df):
    """Add time-based and exogenous features to a
       daily time series dataset.
    """
    # Create a copy to avoid modifying the original
    result = df.copy()

    # Cyclical encoding for day of week (weekly seasonality)
    result['dow_sin'] = np.sin(2 * np.pi * result.index.dayofweek / 7)
    result['dow_cos'] = np.cos(2 * np.pi * result.index.dayofweek / 7)

    # Cyclical encoding for month (yearly seasonality)
    result['month_sin'] = np.sin(2 * np.pi * result.index.month / 12)
    result['month_cos'] = np.cos(2 * np.pi * result.index.month / 12)

    # Keep year for trend analysis
    result['year'] = result.index.year

    # Trend feature (simple incremental counter)
    result['time'] = np.arange(1, len(result)+1)

    # US holidays - fix the datetime comparison warning
    us_holidays = holidays.US(years=result.index.year.unique())
    result['is_holiday'] = result.index.map(
        lambda x: x in us_holidays).astype(int)
```

```

# Weekend feature
result['is_weekend'] = (result.index.dayofweek >= 5).astype(int)

# Month start/end features (can be important for taxi data)
result['is_month_start'] = result.index.is_month_start.astype(int)
result['is_month_end'] = result.index.is_month_end.astype(int)

# For NYC taxi data: check if it's a typical commuting day
result['is_commuting_day'] = (
    (~result['is_holiday'].astype(bool)) &
    (~result['is_weekend'].astype(bool))
).astype(int)

return result

```

2. You can apply the `add_time_features` function to the `tx` DataFrame and perform the same steps we did to evaluate the models. Alternatively, you can combine sliding windows with feature engineering for additional context. Let's do that:

```

# combine with sliding windows for more context
windows = create_sliding_windows(tx, 7)
features = add_time_features(tx)
# Join the windows with the features at the corresponding dates
combined = pd.merge(
    windows,
    features.drop(columns=['value']), # Exclude the target value
    left_index=True,
    right_index=True
)

print(combined.shape)
>>
(208, 18)

```

3. You will follow the same steps we did earlier: instantiate KNN and LOF, fit the model on the transformed dataset (`combined`), use the `predict` method to highlight potential outliers, and filter the outputs to only show the outlier values and associated dates:

```

knn_fe = KNN(contamination=0.03,
              method='mean',
              n_neighbors=5)

lof_fe = LOF(contamination=0.03,

```

```

n_neighbors=20)

knn_fe.fit(combined)
lof_fe.fit(combined)

knn_pred_fe = pd.Series(knn_fe.predict(combined),
                        index=combined.index)

lof_pred_fe = pd.Series(lof_fe.predict(combined),
                        index=combined.index)

# extract the outlier values and dates from knn_pred_fe
knn_outliers_fe = knn_pred_fe[knn_pred_fe == 1]
knn_outliers_fe = tx.loc[knn_outliers_fe.index]
print(knn_outliers_fe)
>>
           value
timestamp
2015-01-28  12947.562500
2015-01-29  14686.145833
2015-01-30  16676.625000
2015-01-31  18702.479167

# extract the outlier values and dates from lof_pred_fe
lof_outliers_fe = lof_pred_fe[lof_pred_fe == 1]
lof_outliers_fe = tx.loc[lof_outliers_fe.index]
print(lof_outliers_fe)
>>
           value
timestamp
2014-12-26  10397.958333
2015-01-27   4834.541667
2015-01-28  12947.562500
2015-01-29  14686.145833
2015-01-30  16676.625000
2015-01-31  18702.479167

```

The results are very similar to what we got earlier using only the sliding window. Depending on the data we are working with, adding additional context could change the output and how the model picks outliers.

Detecting contextual outliers with PyODAdapter

In this section, you will be introduced to the aeon library, which offers a “*scikit-learn compatible toolkit for time series machine learning tasks*.” The library offers several anomaly detection algorithms, including some of those from PyOD. Additionally, they have created a PyODAdapter class, which wraps any of the PyOD algorithms and makes them ready for time-series anomaly detection through a sliding window.

Before starting with the recipe, you will need to install aeon. You can install it using pip:

```
pip install aeon
```

1. Using PyODAdapter is very straightforward and follows the same steps you did before. Start by importing the library:

```
from aeon.anomaly_detection.series import PyODAdapter
```

Using the class is quite simple; you pretty much just wrap your PyOD classes with PyODAdapter. PyODAdapter takes two additional parameters: `window_size` and `stride`. The `window_size` parameter determines how many consecutive observations are included in each sliding window, similar to our manual implementation earlier. The `stride` parameter controls the step size between consecutive sliding windows. A stride of 1 means that the window moves one time step at a time, resulting in overlapping windows. A larger stride value would create non-overlapping windows, which can reduce computational complexity but may also miss some temporal patterns. By default, PyODAdapter uses a stride of 1 to ensure comprehensive coverage of the time series.

Since PyODAdapter takes care of the sliding window transformation, you will use the `tx` DataFrame without any preprocessing this time. You can always add additional features if you want, but for this run, we will only be utilizing PyODAdapter:

```
knn_aeon = PyODAdapter(
    KNN(contamination=0.03, method='mean', n_neighbors=5),
    window_size=7,
    stride=1)

lof_aeon = PyODAdapter(
    LOF(contamination=0.03, n_neighbors=20),
    window_size=7,
    stride=1)

knn_pred_aeon = pd.Series(knn_aeon.fit_predict(tx, axis=0),
                          index=tx.index)

lof_pred_aeon = pd.Series(lof_aeon.fit_predict(tx, axis=0),
                          index=tx.index)
```

2. Unlike our manual PyOD implementation, PyODAdapter doesn't return binary labels (0 or 1) but rather anomaly scores. You will need to determine the threshold for what constitutes an outlier. To get the top 3% outliers, you can use the **quantile** method to create a threshold, then identify outliers that exceed that threshold:

```
threshold_knn = knn_pred_aeon.quantile(0.97) # Top 3% as outliers
threshold_lof = lof_pred_aeon.quantile(0.97) # Top 3% as outliers

# Create binary labels
knn_outliers_aeon = knn_pred_aeon > threshold_knn
lof_outliers_aeon = lof_pred_aeon > threshold_lof
```

3. You now filter and print out the outlier dates and values for each result set:

```
knn_outliers_aeon_tx = tx[knn_outliers_aeon]
print(knn_outliers_aeon_tx)
>>
           value
timestamp
2015-01-25  14463.791667
2015-01-26   7818.979167
2015-01-27   4834.541667
2015-01-28  12947.562500
2015-01-29  14686.145833
2015-01-30  16676.625000
2015-01-31  18702.479167

lof_outliers_aeon_tx = tx[lof_outliers_aeon]
print(lof_outliers_aeon_tx)
>>
           value
timestamp
2015-01-25  14463.791667
2015-01-26   7818.979167
2015-01-27   4834.541667
2015-01-28  12947.562500
2015-01-29  14686.145833
2015-01-30  16676.625000
2015-01-31  18702.479167
```

Notice that the PyODAdapter results differ from our manual sliding window implementation. While both approaches use a window size of 7, PyODAdapter identified a longer continuous sequence of anomalies (seven consecutive days) compared to our manual implementation. This difference occurs because PyODAdapter applies a more sophisticated scoring mechanism that focuses on the anomaly score for each timestamp rather than for each window. It also maintains the original time series index and employs a specialized approach to map window-based anomaly scores back to individual data points.

These differences illustrate how specialized time series tools such as PyODAdapter offer additional refinements to the sliding window approach. While our manual implementation successfully identified the key anomalous period, including the known outlier date (2015-01-27), PyODAdapter extends the detection to include an additional preceding day, providing slightly more context around the anomalous event. Both approaches effectively captured the unusual pattern during the North American blizzard, demonstrating that even a straightforward implementation of sliding windows can yield valuable insights for time series anomaly detection.

Visualize the outputs to gain better insight using the `plot_outliers` functions. The following example is for KNN:

```
plot_outliers(knn_outliers_aeon_tx, tx, 'KNN PyODAdapter')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the x markers are based on the outliers identified using the KNN algorithm using PyODAdapter:

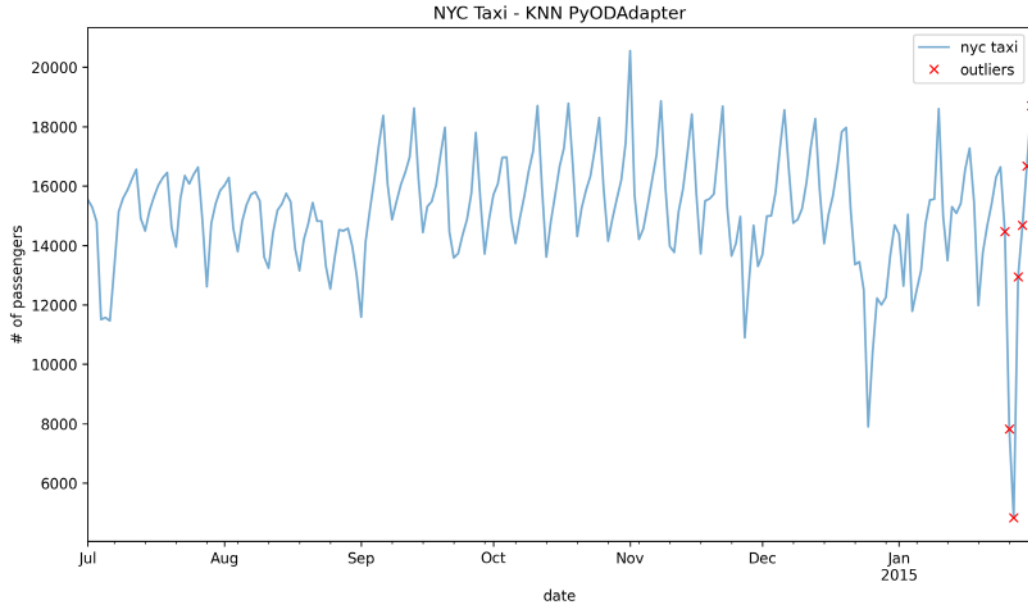


Figure 13.6: Markers showing the identified potential outliers using the KNN algorithm with PyODAdapter

Figure 13.6 shows the time-series data with markers indicating the outliers identified by KNN using PyODAdapter.

How it works...

Both KNN and LOF rely heavily on measuring distances between data points. The choice of distance metric can significantly impact your outlier detection results, as it fundamentally changes how “closeness” is evaluated. PyOD supports several distance metrics from **scikit-learn** and **scipy.spatial**. Here are a few of the popular ones:

- **Manhattan distance** ('minkowski' or 'l1'): Also known as *city block* or *taxicab distance*, measures the sum of absolute differences between coordinates. Useful when dealing with grid-like structures or when features represent different units/scales.

$$d(x, y) = \sum_{i=1}^n |x_i - y_i|$$

- **Euclidean distance** ('euclidean' or 'l2'): The straight-line distance between two points in Euclidean space. This is the square of the sum of squared differences between corresponding coordinates. It works well for continuous data in low-dimensional space where each dimension has similar scales.

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

- **Minkowski distance** ('minkowski'): A generalized metric that encompasses both Euclidean and Manhattan distance, and it's controlled by the parameter **p** in PyOD (e.g., p=1 gives us Manhattan distance while p=2 gives us Euclidean distance).

$$d(x, y) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}$$

- **Mahalanobis distance** ('mahalanobis'): Accounts for covariances in the data, thus measuring distance in terms of standard deviations from the mean. This can be useful when features are correlated.

$$d(x, y) = \sqrt{(x - y)^T \Sigma^{-1} (x - y)}$$

- **Cosine distance** ('cosine'): Measures the cosine of the angle between two vectors, ignoring magnitude. This is useful when you care about directional similarity rather than absolute distance.

$$d(x, y) = 1 - \frac{\sum_i^n x_i y_i}{\sqrt{\sum_i^n x_i^2} \sqrt{\sum_i^n y_i^2}}$$

- **Chebyshev distance** ('chebyshev'): Measures the maximum absolute difference across all dimensions. Useful when a large difference in any one feature is significant.

$$d(x, y) = \max_i |x_i - y_i|$$

In PyOD, you can specify the distance metric using the `metric` parameter. The `metric` parameter can take a string value or a callable function for custom distance calculations:

```
# Using Euclidean distance
knn = KNN(n_neighbors=5, metric='euclidean')

# Using Manhattan distance
lof = LOF(n_neighbors=20, metric='manhattan')

# Using Minkowski with custom p value
knn = KNN(n_neighbors=5, metric='minkowski', p=3)
```

For KNN and LOF, there are three popular algorithms for efficient neighbor searches: `ball_tree`, `kd_tree`, and `brute`, respectively. The default value is set to `algorithm="auto"`.

- **brute**: Uses brute force search by computing the distances between all pairs of points. This can be reasonable for small datasets, but it can become slower for larger ones.
- **kd_tree**: Uses a binary tree structure to organize points and works well in low dimensions.
- **ball_tree**: A more sophisticated tree structure that can perform better in higher dimensions than **kd_tree**.

The `auto` setting will use a decision rule to choose the most efficient algorithm based on the sparsity of your input, the number of neighbors being searched, and the dimensionality of your data.

In addition to algorithm selection, several other parameters affect search efficiency. When you print the model definition/parameters, you will notice the `leaf_size` parameter for both KNN and LOF. Here is an example for KNN:

```
print(knn)
>>
KNN(algorithm='auto', contamination=0.03, leaf_size=30, method='mean',
    metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=5, p=2,
    radius=1.0)
```

The `leaf_size` parameter (defaulting to 30) affects the performance of the tree-based neighbor search algorithms (KD-tree and Ball-tree). A larger `leaf_size` value leads to faster tree construction but slower queries, while a smaller value creates more computationally expensive trees that enable faster queries. This parameter becomes more important when working with larger datasets, as it can significantly impact computational efficiency without affecting the results.

PyOD assigns an internal score to each observation in your data using algorithm-specific calculations. These scores are stored in the `decision_scores_` attribute after fitting the model, showing how strongly each point is considered an outlier—higher values indicate greater abnormality:

```
knn_scores = knn.decision_scores_
```

You can convert this into a DataFrame:

```
knn_scores_df = (pd.DataFrame(
    knn_scores, index=tx.index, columns=['score']))
knn_scores_df
```



The `decision_scores_` attribute contains scores calculated during the `fit()` process and is used internally to determine the **threshold** for outlier classification. If you need outlier scores for new data that wasn't part of the training set, you should use the `decision_function()` method instead:

```
knn_new_scores = knn.decision_function(new_data)
```

After scoring all data points, PyOD determines a **threshold** value to classify points as outliers or normal. This threshold is directly determined by the **contamination** parameter you set, which represents your expectation of what percentage of your data consists of outliers. The relationship works like this:

- A higher contamination value lowers the threshold, resulting in more points being flagged as outliers
- A lower contamination value raises the threshold, resulting in fewer points flagged as outliers (more conservative detection)

You can get the threshold value using the `threshold_` attribute from the model after fitting it to the training data. Here is the threshold for KNN based on a 3% contamination rate:

```
knn.threshold_
>> 225.0179166666657
```

You could obtain the same value using the quantile method to obtain the top 3% outliers based on their scores:

```
# We use 0.97 since we're looking for the top 3% (100% - 3% = 97%)
knn_scores_df.quantile(0.97)
>>
score    225.017917
```

This is the value used to filter out the significant outliers. Here is an example of how you can reproduce that:

```
print(knn_scores_df[knn_scores_df['score'] > knn.threshold_])
```

The output is as follows:

	score
timestamp	
2014-09-27	225.604167
2014-11-01	1806.850000
2014-11-27	608.250000
2014-12-25	2441.250000
2014-12-26	1009.616667
2015-01-26	2474.508333
2015-01-27	4862.058333

Notice the observation on 2014-09-27 is slightly above the threshold, but it was not returned when you used the `predict` method. This is likely due to rounding differences or implementation details in how PyOD applies the threshold. If you use the **contamination** parameter directly to select the top 3% of observations, you can get a more consistent cutoff:

```
n = int(len(tx)*0.03)
print(knn_scores_df.nlargest(n, 'score').sort_index())
>>
```

	score
timestamp	
2014-11-01	1806.850000
2014-11-27	608.250000
2014-12-25	2441.250000
2014-12-26	1009.616667
2015-01-26	2474.508333
2015-01-27	4862.058333

Another helpful method is `predict_proba`, which returns the probability of being normal and the probability of being abnormal for each observation. PyOD provides two methods for determining these percentages: `linear` and `unify`. The two methods scale the outlier scores before calculating the probabilities. For example, in the case of `linear`, the implementation uses `MinMaxScaler` from `scikit-learn` to scale the scores (range between 0 and 1) before calculating the probabilities. The `unify` method uses the **z-score** (**standardization**) and the **Gaussian error function** (`erf`) from the `SciPy` library (`scipy.special.erf`).

Here is an example using the `linear` method to calculate the prediction probability. The probability ranges from 0 and 1, and in this example, we multiplied by 100 to make it easier to interpret as percentages:

```
knn_proba = knn.predict_proba(tx, method='linear')
knn_proba_df = (pd.DataFrame(
    np.round(knn_proba * 100, 3),
```

```

        index=tx.index,
        columns=['Proba_Normal', 'Proba_Anomaly'])
    )

```

The first column, `Proba_Normal`, indicates the probability that the observation is normal, while `Proba_Anomaly` shows the probability that it's an outlier. The values in each row sum to 100%:

```

print(knn_proba_df.nlargest(n, 'Proba_Anomaly'))
>>

```

	Proba_Normal	Proba_Anomaly
timestamp		
2015-01-27	27.337	72.663
2015-01-26	64.256	35.744
2014-12-25	64.599	35.401
2014-11-01	70.682	29.318
2014-12-26	84.341	15.659
2014-11-27	90.548	9.452

The `predict_confidence` assesses how consistently the model would make the same prediction if the training data were slightly perturbed. In other words, it informs us about the robustness of the predictions rather than the probability of a sample being an outlier. Higher values mean the model would likely make the same prediction even if the training data changed slightly.

In PyOD, normal observations typically receive a confidence score of exactly 1.0, indicating complete confidence in their classification. Only outliers receive confidence scores less than 1.0, with lower values indicating the model is less certain about these unusual observations. This makes `predict_confidence` particularly useful for identifying borderline cases among your detected outliers:

```

knn_conf = knn.predict_confidence(tx)
knn_conf_df = (pd.DataFrame(knn_conf,
                           index=tx.index,
                           columns=['confidence']))
print(knn_conf_df.nsmallest(n, 'confidence'))
>>

```

	confidence
timestamp	
2014-11-27	0.304893
2014-12-26	0.452453
2014-11-01	0.624149
2014-12-25	0.624149
2015-01-26	0.624149
2015-01-27	0.984580

Though the output from `predict_proba` and `predict_confidence` may seem similar, they come from different perspectives:

- Low confidence from `predict_confidence` indicates unstable prediction that would change if the training data were slightly different—a strong signal that a point is an outlier
- High anomaly probability from `predict_proba` directly measures how likely a point is to be an outlier by transforming the outlier scores into probabilities (either using simple min-max scaling with `method='linear'` or more sophisticated unifying scores with `method='unify'`)

There's more...

In the *Detecting contextual outliers with PyODAdapter* section, `window_size` and `stride` have a direct impact on the model output. This, of course, is in addition to other hyperparameters, such as the number of neighbors to consider (`n_neighbors`), distance metrics (`metric`), and the distance aggregation method (`method`), to name a few.

Let's illustrate this with a few examples on the impact of the method from KNN, and changing `window_size` in `PyODAdapter`.

Create a function to use with KNN and LOF to train the model on different scoring methods by updating the `method` parameter to either `mean`, `median`, or `largest` to examine the impact on the decision scores. Use the point outlier technique using the PyOD model as is:

Create the `knn_explore` function with the following parameters: `df`, `method`, `contamination`, and `k`:

```
def knn_explore(df, method='mean', contamination=0.03, k=5):

    m = KNN(contamination=contamination, method=method, n_neighbors=k)
    m.fit(df)

    decision_score = pd.DataFrame(m.decision_scores_,
                                  index=df.index, columns=['score'])

    n = int(len(df)*contamination)
    outliers = decision_score.nlargest(n, 'score')
    return outliers, m.threshold_
```

You can run the function using different aggregation methods, contamination, and `k` values to experiment.

Explore the different aggregation methods and how they yield different thresholds and decision scores, thus impacting which outliers are detected:

```
for method in ['mean', 'median', 'largest']:
    o, t = knn_explore(tx, method=method, contamination=0.05)
    print(f'Method= {method}, Threshold= {t}')
    print(o)
```

The preceding code should print out the top 10 outliers (contamination at 5%) for each method.

The order of the outliers is based on the score aggregation method, which also impacted the threshold value.

Lastly, let's explore the effect of different `window_size` values in `PyODAdapter` (note that `stride` defaults to 1) for the **LOF** algorithm:

```
plt.figure(figsize=(15, 10))
known_anomalies = pd.to_datetime(nyc_dates)
window_sizes = [1, 3, 7, 14, 28]

for i, window in enumerate(window_sizes, 1):
    detector = PyODAdapter(
        pyod_model=LOF(contamination=0.03, n_neighbors=5),
        window_size=window
    )

    scores = pd.Series(detector.fit_predict(tx, axis=0), index=tx.index)
    threshold = scores.quantile(0.97)
    outliers = scores > threshold
    outlier_dates = scores[outliers].index

    # Check against known anomalies
    detected = sum(anomaly in outlier_dates for anomaly in known_anomalies)

    plt.subplot(len(window_sizes), 1, i)
    plt.plot(scores)

    # Mark outliers with X
    plt.scatter(outlier_dates, scores[outliers], color='red', marker='x', s=50)

    plt.title(f"Window Size = {window}, threshold = {threshold:.3f}, "
             f"Total outliers found: {sum(outliers)}, "
             f"Detected {detected}/{len(known_anomalies)} known anomalies")
    plt.axhline(y=threshold, color='r', linestyle='--')

    # Mark known anomalies
    for date in nyc_dates:
        if pd.Timestamp(date) in scores.index:
            plt.axvline(x=pd.Timestamp(date), color='g', alpha=0.3)

plt.tight_layout()
plt.show()
```

This should produce the following five subplots:

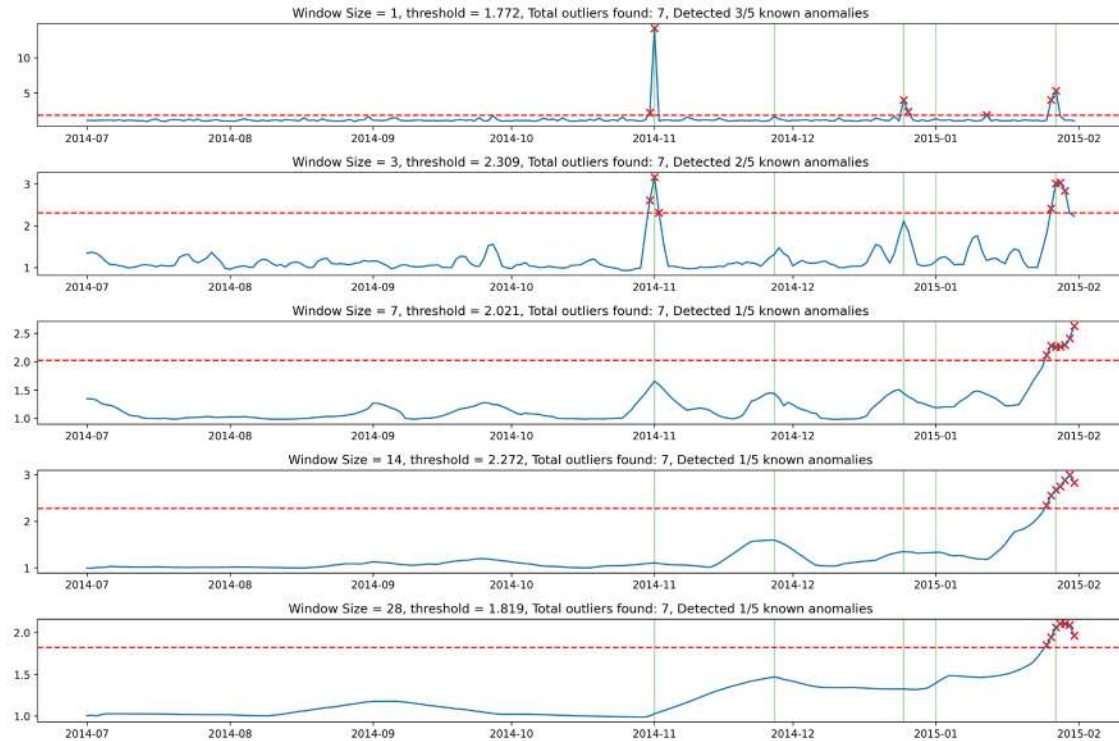


Figure 13.7: Comparing different window size values and their impact on outliers in LOF with PyODAdapter

In Figure 13.7, the continuous horizontal line represents the **anomaly scores** over time, the horizontal dashed line represents the **threshold** used to determine outliers (points above the threshold), the vertical lines represent **known outliers**, and the X marks the **outlier** identified by the LOF with the sliding window. Notice how larger window sizes produce progressively smoother anomaly score lines, reflecting how the algorithm considers more observations together when calculating anomaly scores with larger windows.

These visualizations reveal interesting patterns in how window size affects anomaly detection. With a small window size of 1, the model essentially performs point outlier detection, evaluating each observation in isolation without temporal context. This approach detected 3 out of 5 known anomalies but produced more volatile scoring.

As the window size increases, the detector shifts toward contextual outlier detection, where anomalies are identified based on their relationship to neighboring observations. The anomaly scores become smoother, focusing detection increasingly on the January 2015 blizzard period.

Notably, smaller windows (1 and 3) were more successful at detecting the November and December anomalies, while larger windows (7, 14, and 28) exclusively identified anomalies in the January 2015 period. This demonstrates the trade-off: smaller windows are more sensitive to sudden, short-lived anomalies, while larger windows better capture extended anomalous periods but might miss isolated events.

When applying PyODAdapter to your own data, consider experimenting with different window sizes based on the expected duration of anomalies in your domain and whether you're more interested in identifying point outliers (extreme values) or contextual outliers (unusual patterns).

See also

Check out the following resources:

- To learn more about **PyODAdapter**, visit the official documentation: https://www.aeon-toolkit.org/en/stable/api_reference/auto_generated/aeon.anomaly_detection.series.PyODAdapter.html#aeon.anomaly_detection.series.PyODAdapter
- To learn more about **LOF** in PyOD, visit the official documentation: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.lof>
- To learn more about **KNN** in PyOD, visit the official documentation: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.knn>

Detecting outliers using clustering-based algorithms

In this recipe, we focus on clustering-based outlier detection algorithms, which identify anomalies by examining how data points relate to clusters within the dataset. Specifically, we will explore the **Cluster-Based Local Outlier Factor (CBLOF)** algorithm. While distance-based methods such as KNN and LOF examine relationships between individual points, CBLOF first organizes data into clusters and then identifies outliers based on cluster properties. This approach determines outlier scores by considering both cluster size and the distance of points from their cluster centers. Instead of using a number of neighbors parameter (as in KNN and LOF), CBLOF introduces a new parameter—the number of clusters (`n_clusters`)—which determines how the data is partitioned before outlier detection begins. The default clustering method (`clustering_estimator`) in PyOD's implementation is k-means (KMeans), though other clustering algorithms can be substituted by any valid clustering algorithm from scikit-learn.

Building on our coffee shop analogy from the previous recipe, we can further understand how clustering-based methods, such as CBLOF, approach outlier detection differently from distance-based methods such as KNN and LOF. Recall that, in our analogy, we considered customers waiting in line at a coffee shop and a person sitting alone in their car in the parking lot. KNN identified the person in the car as an outlier based on absolute distance, while LOF considered local density variations to detect outliers in different contexts. Now, let's see how CBLOF would analyze this same scenario.

CBLOF perspective

CBLOF takes a different approach by first organizing people into groups (clusters). In our coffee shop scenario, it would first identify natural groupings: perhaps one cluster for people standing in line, another for those seated at tables, and a third for staff behind the counter. CBLOF would then determine which groups are “large” (like the line of customers) and which are “small” (like the lone person in their car).

While CBLOF's name might suggest a close relationship to LOF, their methodologies differ significantly. While LOF compares the local density of a point to its neighbors, CBLOF first creates a structural understanding of the data through clustering. It then assigns outlier scores based on two key factors:

- Whether a person belongs to a small group (higher outlier score)
- In the larger group, how far they are from the center of their assigned group (higher distance = higher outlier score)

The person in the car would be flagged as an outlier because they form a very small cluster (or a cluster of one) and are distant from the centers of the larger clusters. However, CBLOF would also identify other potential outliers that KNN and LOF might miss. For instance, if someone is technically part of the line but standing unusually far from the center of the line, CBLOF might flag them even if their neighbors are relatively close (which KNN might miss) and the local density is consistent (which LOF might miss).

CBLOF excels at identifying **cluster-based outliers**, detecting both points that don't belong to any major cluster and points that deviate significantly from the center of their assigned cluster.

How to do it...

CBLOF has different key clustering parameters that you need to be aware of, in addition to the standard contamination parameter (across all PyOD algorithms). The key parameters include `n_clusters`, `alpha`, and `beta`. The `alpha` and `beta` parameters help determine which clusters are large (normal) or small (potential outliers). The default values are `n_clusters=8`, `alpha=0.9`, and `beta=5`.

1. Start by importing the CBLOF class and instantiating it using the default parameter values, except for contamination, which you should change to 0.03:

```
from pyod.models.cblof import CBLOF
cblof = CBLOF(n_clusters=8,
               contamination=0.03,
               alpha=0.9,
               random_state=15,
               beta=5)
```

2. Fit `cblof` to the `tx` DataFrame, then use the `predict` method to flag outliers by generating binary labels for each data point (1 if True or 0 if False). Store the results in a pandas Series:

```
cblof.fit(tx)
# Flag outliers using CBLOF predict method
cblof_pred = pd.Series(cblof.predict(tx),
                       index=tx.index)
print('Number of CBLOF outliers = ', cblof_pred.sum())
>>
Number of CBLOF outliers = 7
```

3. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from cblof_pred
cblof_outliers = cblof_pred[cblof_pred == 1]
cblof_outliers = tx.loc[cblof_outliers.index]
print(cblof_outliers)
>>
```

	value
timestamp	
2014-11-01	20553.500000
2014-11-27	10899.666667
2014-11-28	12850.854167
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

Overall, the results look promising; CBLOF successfully identified four out of the five known dates (NYC Marathon, Thanksgiving, Christmas, and the North American Blizzard), but missed New Year's Day (January 1, 2015). Interestingly, the algorithm also identified three additional potential outliers: the day after Thanksgiving (November 28, 2014), the day after Christmas (December 26, 2014), and the day before the North American Blizzard (January 26, 2015).

These additional detections actually demonstrate an interpretable pattern—holiday effects often extend beyond just the single day, possibly indicating reduced taxi demand continues into surrounding days. People likely altered their travel plans the day before the severe blizzard as weather warnings were issued. This demonstrates how clustering-based methods can reveal temporal patterns that point-based approaches might miss.

4. Use the `plot_outliers` function created in the *Technical requirements* section to visualize the output to gain better insight:

```
plot_outliers(cblof_outliers, tx, 'CBLOF')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the X markers are based on the outliers identified using the CBLOF algorithm:

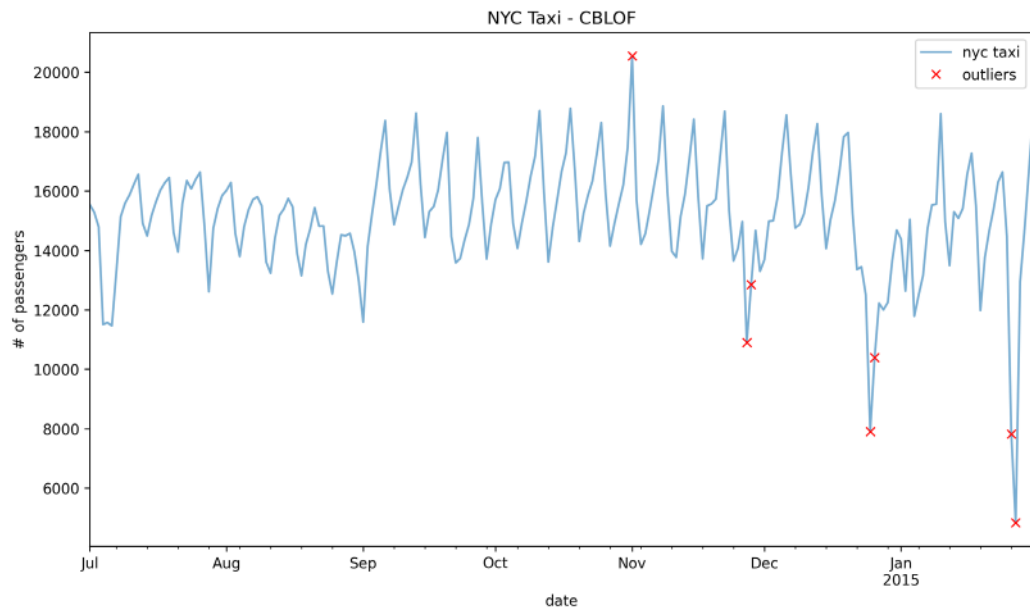


Figure 13.8: Markers showing the identified potential outliers using the CBLOF algorithm

How it works...

Cluster-Based Local Outlier Factor (CBLOF) is an unsupervised anomaly detection algorithm that identifies outliers by combining clustering techniques with local density analysis. Unlike point-by-point methods (e.g., KNN or LOF), it evaluates clusters' properties and their spatial relationships to detect anomalies.

Here are the high-level steps in the CBLOF algorithm:

1. **Data is grouped into clusters** where the number of clusters depends on the clustering estimator, which is typically K-means and its `n_clusters` parameter.
2. **Clusters are then classified into small and large:** large clusters represent normal data, while small clusters are potential outliers.
3. **Anomaly scores are calculated** for each data point based on its relationship to these clusters:
 - For points in small clusters (or isolated points), the outlier score depends on the distance to the center of the nearest large cluster. In other words, a small cluster is considered more anomalous if its centroid is relatively far from the centroid of the large clusters.
 - For points in large clusters, the score reflects their proximity to their own cluster's centroid.
4. **Observations with scores above the threshold** (user-defined through the contamination parameter) are then flagged as anomalies.

There are three parameters that control the classification of clusters (determining which are large versus small) that should be explored through hyperparameter tuning to get optimal results for your data:

- **n_clusters** (default=8): This dictates the number of clusters to be created in the initial clustering step. This gets passed to the clustering algorithm directly (e.g., KMeans from scikit-learn by default).
- **alpha** (default=0.9, range 0.5-1.0): Controls the minimum proportion of data that must be in large clusters (considered “normal”).
 - A larger value (e.g., 0.95) requires at least 95% of data to be in large clusters, resulting in fewer clusters being classified as “small,” which potentially reduces false positives but risks missing subtle anomalies.
 - A lower value (e.g., 0.70) requires at least 70% of data in large clusters, resulting in more data being classified as “small.” This increases sensitivity—flagging more potential anomalies (higher recall) but potentially increasing false positives.
- **beta** (default=5, range >1.0): This parameter provides another criterion for separating large and small clusters based on relative size differences.
 - A larger value (e.g., 10) requires more dramatic size differences between clusters, focusing on more extreme outlier groups.
 - A smaller value (e.g., 2) allows for more gradual transitions, potentially identifying more subtle anomalies.

In our recipe, we use the defaults (`n_clusters=8`, `alpha=0.9`, and `beta=5`), which work well for balanced datasets, but you should adjust these based on your specific data characteristics.

Additionally, PyOD offers utility functions that can be very useful. For a list of these utility functions, you can visit their page here: <https://pyod.readthedocs.io/en/latest/pyod.utils.html>.

For example, to quantitatively evaluate CBLOF’s performance (or any PyOD model), you can use PyOD’s `evaluate_print()` function, which produces an ROC AUC score and a precision @ rank *n* score.

```
from pyod.utils import evaluate_print
# Create binary ground truth labels (1 for known outliers, 0 for normal points)
y = tx.index.isin(known_outliers.index).astype(int)
evaluate_print('CBLOF', y, cblof.decision_scores_)
>>
CBLOF ROC:0.9333, precision @ rank n:0.6
```

The CBLOF model demonstrates promising detection capabilities. The ROC AUC of 0.9333 (where 1.0 is perfect separation) shows strong discriminative power between normal and anomalous patterns—this exceeds the 0.9 threshold considered excellent in anomaly detection research.

The precision@n score of 0.6 (where $n=5$ matches our ground truth count) means that among the top 5 points ranked as most anomalous, 3 are true anomalies. Interestingly, while CBLOF successfully identified 4 out of our 5 known outliers in total and detected 7 outliers overall, the precision score suggests that some of the additional dates it flagged (like the days after holidays) received higher anomaly scores than certain known anomalies. This highlights the importance of examining both the binary predictions and the underlying anomaly scores when applying these algorithms.

There's more...



For deeper inspection of results, you can access `decision_scores_`, `threshold_`, and `predict_proba`—attributes that are available across all PyOD models. For a refresher on interpreting these attributes and methods to incorporate temporal context (such as sliding windows, feature engineering, or using PyODAdapter), refer to the first recipe in this chapter, *Detecting outliers using distance-based algorithms*.

In the first recipe of this chapter, *Detecting outliers using distance-based algorithms*, you explored five different techniques for outlier detection using PyOD:

- Detecting **point** outliers (using PyOD as is)
- Detecting outliers on the residuals after decomposition (e.g., STL)
- Detecting **contextual** outliers using sliding window transformation
- Detecting **contextual** outliers with time-based feature engineering
- Detecting **contextual** outliers using PyODAdapter from the aeon library

Building on those five techniques, you can apply the same approaches (STL decomposition, sliding windows, etc.) with CBLOF. However, the aeon library offers additional specialized algorithms worth exploring—particularly their **KMeansAD** implementation, which naturally combines clustering with temporal awareness. Unlike PyODAdapter, which wraps existing PyOD algorithms, KMeansAD is purpose-built for time series with its own implementation.

The **KMeansAD** (K-Means anomaly detector) is also based on K-Means clustering, but includes a sliding window, and the K-Means algorithm is applied on each window for clustering. The anomaly score for each time point is calculated as the average Euclidean distance between the windows containing that point and their corresponding cluster centers.

This approach captures temporal patterns by using sliding windows and can detect contextual outliers by considering the surrounding data points. The steps are similar to the steps used in the previous recipe, when using PyODAdapter.

Start by importing the class from the aeon library and instantiating KMeansAD:

```
from aeon.anomaly_detection.series.distance_based import KMeansAD
# window_size=7 means that the K-Means algorithm is applied to 7 consecutive
# data points
aeon_kmeans = KMeansAD(n_clusters=8, window_size=7)
```

Use the `fit_predict` method and store the resulting **scores** in a pandas' Series:

```
aeon_pred = pd.Series(aeon_kmeans.fit_predict(tx, axis=0),
                      index=tx.index)
```

The anomaly score for each time point is calculated as the average Euclidean distance between the windows containing that point and their corresponding cluster centers. A larger average Euclidean distance suggests that the data point's surrounding window deviates significantly from the typical patterns within its assigned cluster, indicating a potential outlier.

Establish a threshold using the quantile method. For consistency with the previous steps (contamination=0.03), you will use `quantile(0.97)` for the top 3% of outliers.

```
threshold = aeon_pred.quantile(0.97)
outliers_mask = aeon_pred > threshold
outliers = tx[outliers_mask]
print(outliers)
>>
```

	value
timestamp	
2015-01-25	14463.791667
2015-01-26	7818.979167
2015-01-27	4834.541667
2015-01-28	12947.562500
2015-01-29	14686.145833
2015-01-30	16676.625000
2015-01-31	18702.479167

Notice how `KMeansAD` detects a continuous outlier period (Jan 25-31) surrounding the known blizzard date (Jan 27). This demonstrates its ability to identify sustained outliers rather than isolated spikes. Unlike `CBLOF`, which flagged specific holiday dates, `KMeansAD` captures the entire “event window” around the blizzard when taxi patterns were disrupted over multiple days.

In the first recipe, you explored how using `PyOD` as is (without temporal adaptation) will detect **point outliers** since the library is not specific for time-series data without transforming the data if you want contextual outliers. You can explore `aeon's PyODAdapter` with `CBLOF` to compare the results from `KMeansAD` using the same `window_size=7` parameter—this would help evaluate whether the window-based approach or the specific clustering mechanism drives the improved contextual detection.

Finally, use the `plot_outliers` functions to visualize the outliers and compare the difference between point outliers in *Figure 13.8* and contextual outliers in *Figure 13.9*:

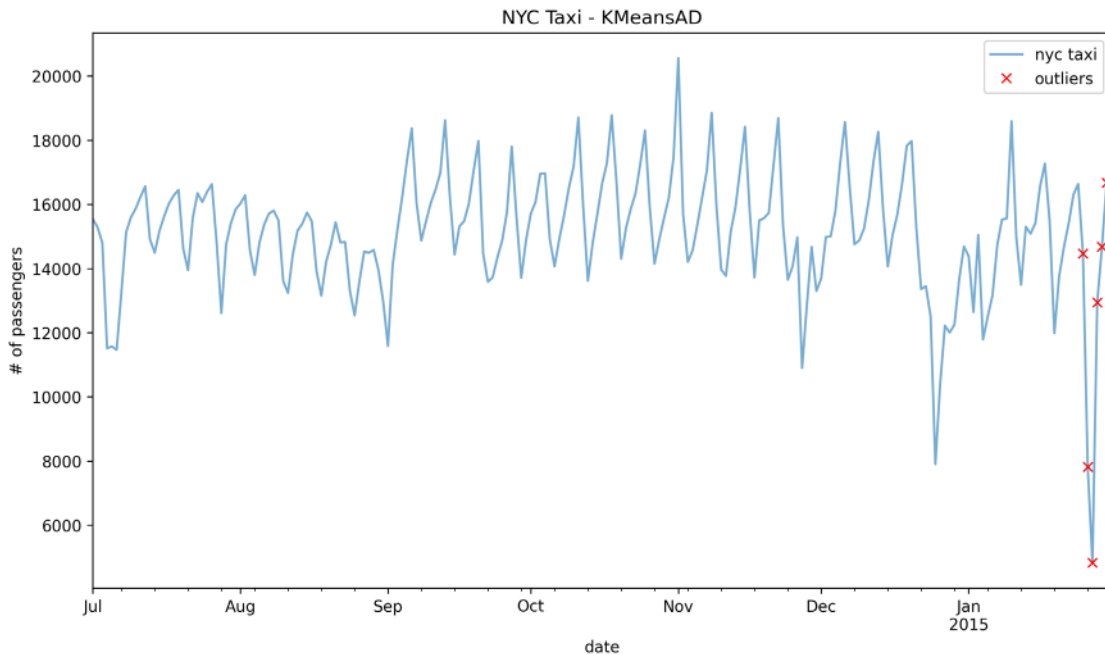


Figure 13.9: Markers showing the identified potential outliers using the KMeansAD algorithm



Keep in mind that K-means-based methods, including CBLOF and KMeansAD, may produce varying results due to random initialization. To ensure reproducibility and consistent results across runs, set the `random_state` parameter:

```
cblof = CBLOF(random_state=42)
aeon_kmeans = KMeansAD(random_state=42)
```

See also

- To learn more about **CBLOF** in **PyOD**, visit the official documentation: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.cblof>
- To learn more about **KMeansAD** in **aeon**, visit the official documentation: https://www.aeon-toolkit.org/en/stable/api_reference/auto_generated/aeon.anomaly_detection.series.distance_based.KMeansAD.html

Detecting outliers using probabilistic and statistical algorithms

In this recipe, we focus on probabilistic and statistical outlier detection algorithms, which identify anomalies by modeling the underlying probability distributions of data. Specifically, you will explore the **Copular-Based Outlier Detection (COPOD)** algorithm, a parameter-free method based on a paper published in September 2020, which you can read here: <https://arxiv.org/abs/2009.09463>. While distance-based methods such as KNN and LOF examine spatial relationships between points, and clustering-based methods such as CBLOF organize data into groups (clusters), COPOD takes a statistical approach by modeling the joint distribution of features using copulas. This makes COPOD computationally efficient and highly interpretable.

COPOD is based on **copula** theory, where a copula is a mathematical function that joins (or “couples”) multivariate distributions to their one-dimensional marginal distributions. This approach allows COPOD to model complex dependencies between variables without making strong assumptions about the underlying data distribution. The algorithm works by doing the following:

1. Computing empirical distributions for each feature.
2. Transforming data into copula space.
3. Calculating tail probabilities (how extreme values are).
4. Converting these probabilities into outlier scores.

Copulas, as functions, can model non-Gaussian and non-linear relationships, and can capture extreme values (tails of distributions), which often correspond to outliers.

What makes COPOD particularly attractive for practical applications is that it’s almost parameter-free—the only configuration needed is the contamination factor (defaulting to 0.1), which dictates the proportion of data expected to be outliers. This makes COPOD much easier to configure compared to algorithms such as KNN or LOF, which require careful tuning of the *k* parameter, or CBLOF, which needs an appropriate number of clusters.

Extending our coffee shop analogy from previous recipes, let’s understand how COPOD approaches outlier detection differently. Recall that in our analogy, we considered customers waiting in line at a coffee shop and a person sitting alone in their car in the parking lot. While KNN, LOF, and CBLOF focused on spatial relationships (distances, densities, or clusters), COPOD instead asks: “How statistically unusual is each customer’s behavior, considering all attributes together?”

COPOD would analyze the coffee shop scenario by first examining the statistical distributions of various attributes. For each customer, it might consider variables such as waiting time, distance from the counter, and purchase amount. Rather than directly measuring distances between customers (KNN) or analyzing local densities (LOF) or cluster memberships (CBLOF), COPOD transforms each of these attributes into probability distributions.

For instance, COPOD would determine how common or rare a customer’s waiting time is by calculating its probability based on the observed distribution of all waiting times. It would do the same for each attribute independently, then combine these probabilities using copula functions to create a joint probability model that accounts for dependencies between attributes.

The approach allows COPOD to model complex dependencies between variables without making strong assumptions about the underlying data distribution.

The person waiting in their car would be flagged as an outlier because their combination of attributes (very long waiting time, large distance from counter, zero current purchase amount) would have a very low joint probability according to the model. However, COPOD might also identify subtle outliers that other methods miss—such as a customer who has a normal waiting time and distance, but whose purchase pattern is statistically unusual when considered in conjunction with other variables.

COPOD excels at identifying statistical outliers, detecting points whose attribute combinations are improbable according to the learned joint distribution, regardless of their spatial positioning or cluster membership.

How to do it...

In this recipe, you will continue to work with the `tx` DataFrame to detect outliers using the COPOD class from the PyOD library:

1. Start by importing the COPOD class and instantiating it with a contamination value, then fit the model on the resampled `tx` DataFrame:

```
from pyod.models.copod import COPOD
copod = COPOD(contamination=0.03)
copod.fit(tx)
```

The only parameter you need to provide is `contamination`. Generally, think of this parameter (used in all the PyOD outlier detection implementations) as a threshold to control the model's sensitivity and minimize the false positives. Since it is a parameter you control, ideally, you want to run several models to experiment with the ideal threshold rate that works for your use cases.

Use the `predict` method to identify outliers. The method will output either 1 or 0 for each data point where 1 indicates an outlier. Store the results in a pandas Series.

```
copod_pred = pd.Series(copod.predict(tx), index=tx.index)
print('Number of COPOD outliers = ', copod_pred.sum())
>>
Number of COPOD outliers = 7
```

2. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from copod_pred
copod_outliers = copod_pred[copod_pred == 1]
copod_outliers = tx.loc[copod_outliers.index]
print(copod_outliers)
>>
           value
timestamp
2014-07-04  11511.770833
```

```

2014-07-06 11464.270833
2014-11-27 10899.666667
2014-12-25 7902.125000
2014-12-26 10397.958333
2015-01-26 7818.979167
2015-01-27 4834.541667

```

Compared with other algorithms you have explored so far, you will notice some interesting outliers captured with COPOD that were not identified before. For example, COPOD identified July 4, a national holiday in the US (Independence Day). It happens to fall on a weekend (Friday being off). The COPOD model captured anomalies throughout the weekend for July 4 and July 6. It happens that July 6 was an interesting day due to a baseball game in New York.

3. Use the `plot_outliers` function created in the *Technical requirements* section to visualize the output to gain better insights:

```
plot_outliers(copod_outliers, tx, 'COPOD')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the X markers are based on the outliers identified using the COPOD algorithm:

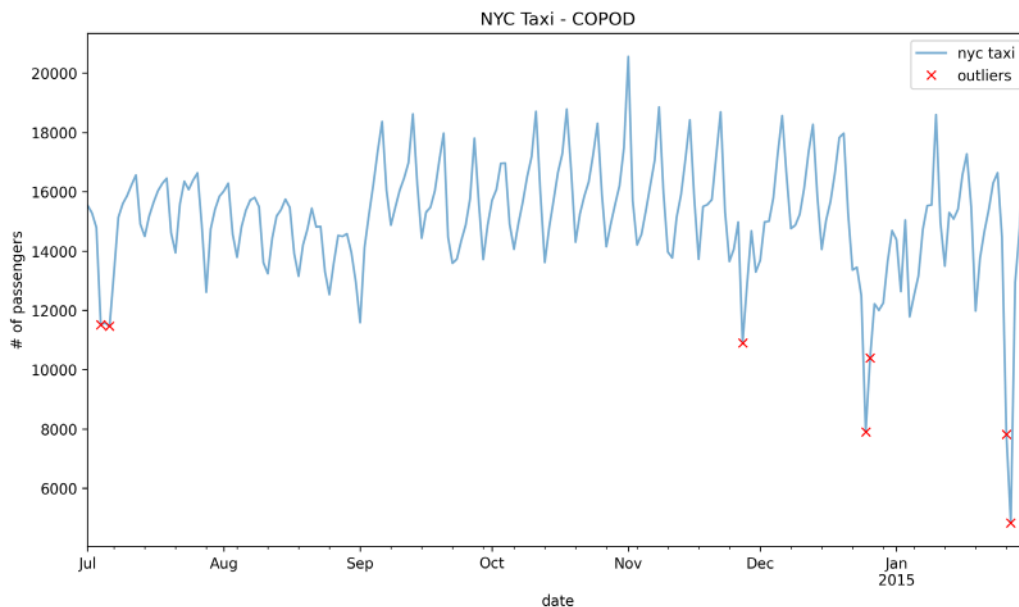


Figure 13.10: Markers showing the identified potential outliers using the COPOD algorithm

While KNN, LOF, and CBLOF all identified the extreme anomalies of November 1 (Marathon) and January 27 (Blizzard), COPOD uniquely detected July 4 (Independence Day) which the others missed. However, COPOD did not flag November 1, which was one of our known anomalies. This demonstrates how different algorithmic approaches can complement each other by capturing different types of outliers.

How it works...

COPOD is an advanced yet intuitive algorithm that leverages probabilistic modeling to identify data points that deviate significantly from expected statistical patterns. Its effectiveness stems from its ability to assess outliers using empirical cumulative distributions, without making strong assumptions about the underlying data distribution. While COPOD excels at modeling complex dependencies in multivariate

datasets, it also performs well on univariate time series by evaluating how each

point's probability deviates from the expected distribution. Several tests using COPOD have demonstrated its superb performance against benchmark datasets. Unlike many other outlier detection methods that require extensive hyperparameter tuning, COPOD's parameter-free nature (aside from the contamination factor) makes it remarkably easy to use, saving valuable time and effort.

There's more...



For deeper inspection of results, you can access the `decision_scores_`, `threshold_`, and `predict_proba` attributes, which are available across all PyOD models. For a refresher on interpreting these attributes and methods to incorporate temporal context (such as sliding windows, feature engineering, or using PyODAdapter), refer to the first recipe in this chapter, *Detecting outliers using distance-based algorithms*.

Another powerful probabilistic algorithm that's conceptually similar to COPOD is ECOD (Empirical Cumulative Distribution Function), introduced in a 2022 paper: <https://arxiv.org/abs/2201.00382>. Like COPOD, ECOD is almost parameter-free (except for the contamination parameter) and highly interpretable. While COPOD models the joint probability distribution using copulas, ECOD simplifies the approach by directly using **empirical cumulative distribution functions (ECDFs)** to estimate the tail probabilities of data points. This makes ECOD even more computationally efficient while maintaining strong detection performance.

You can implement ECOD using the following code:

```
from pyod.models.ecod import ECOD

# Initialize and fit the ECOD model
ecod = ECOD(contamination=0.03)
ecod.fit(tx)

# Predict outliers
ecod_pred = pd.Series(ecod.predict(tx), index=tx.index)
print('Number of ECOD outliers = ', ecod_pred.sum())

# extract the outlier values and dates from ecod_pred
ecod_outliers = ecod_pred[ecod_pred == 1]
```

```

ecod_outliers = tx.loc[ecod_outliers.index]
print(ecod_outliers)
>>
Number of ECOD outliers = 6

          value
timestamp
2014-10-18  18778.958333
2014-11-01  20553.500000
2014-11-08  18857.333333
2014-12-25   7902.125000
2015-01-26   7818.979167
2015-01-27   4834.541667

```

Note how ECOD produces different results from COPOD. While both identify the Christmas Day (2014-12-25) and Blizzard days (2015-01-26, 2015-01-27) as outliers, ECOD also flags November 1 (New York Marathon), which COPOD missed. However, ECOD doesn't detect Thanksgiving (2014-11-27) or the days after Christmas (2014-12-26), which COPOD identified.

Interestingly, ECOD also highlights October 18 and November 8, 2014, which weren't in our known anomaly dates.

Finally, use the `plot_outliers` functions to visualize the outliers and compare the difference between COPOD, in *Figure 13.10*, and ECOD, in *Figure 13.11*:

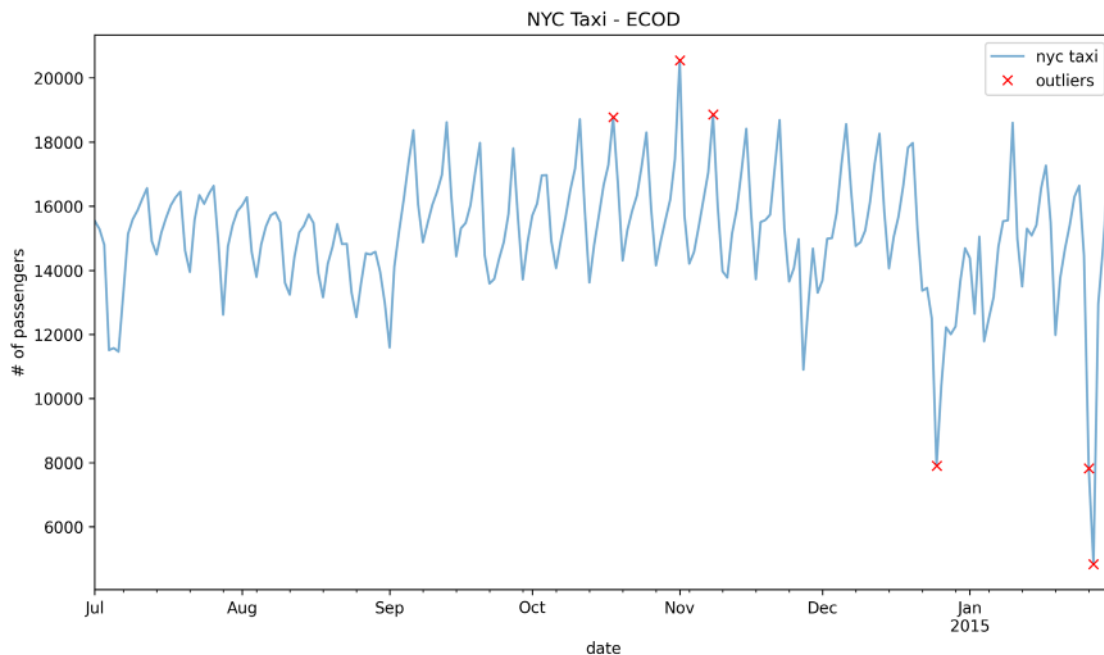


Figure 13.11: Markers showing the identified potential outliers using the ECOD algorithm

See also

- To learn more about **COPOD** in PyOD, visit the official documentation: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.copod>
- To learn more about **ECOD** in PyOD, visit the official documentation: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.ecod>
- Other probabilistic and statistical algorithms you should explore in PyOD include **ABOD** (Angle-Based Outlier Detection), **MAD** (Median Absolute Deviation), **KDE** (Kernel Density Estimation), and **GMM** (Gaussian Mixture Model).

Detecting outliers using kernel-based algorithms (PyOD)

In this recipe, we explore kernel-based outlier detection algorithms, an approach that identifies anomalies by learning complex, non-linear boundaries between normal and outlier data. Specifically, you will be introduced to **One-Class Support Vector Machine (OCSVM)**. Traditional **Support Vector Machine (SVM)** is a popular supervised machine learning algorithm used for classification and regression tasks. You were introduced to the SVM in *Chapter 11* as a regressor through the SVR class from the `scikit-learn` library. OCSVM adapts SVM for unsupervised outlier detection. A key difference is that traditional SVM for classification problems can be used for multi-class classifications, while OCSVM is used for a single class of “normal” data points.

The popularity of SVM in general comes from its use of **kernel functions**, such as linear, polynomial, **Radius-Based Function (RBF)**, and sigmoid. These kernels give SVM the ability to adapt and handle complex data relationships (from linear to non-linear). Similarly, the power of OCSVM comes from its use of these kernel functions, allowing it to transform data into higher-dimensional feature spaces where it becomes easier to separate normal data from outliers. The ability of OCSVM to leverage different kernels makes it a versatile tool for creating complex decision boundaries that better capture what constitutes “normal” behavior in your data. Any observation outside this boundary will be considered an outlier.

Similar to how we adapted KNN from a supervised classification technique to an unsupervised outlier detection method in the *Detecting outliers using distance-based algorithms* recipe, OCSVM operates in an unsupervised manner despite SVM’s traditional supervised applications. However, unlike distance-based methods such as KNN and LOF that rely on proximity calculations, OCSVM creates a boundary or frontier around the normal data, treating points that fall outside this boundary as potential outliers. Kernel-based methods, such as OCSVM, excel in outlier detection compared to proximity-based approaches (such as KNN and LOF) when dealing with complex, non-linear relationships in the time series data that simple distance metrics might miss. It’s effective, especially when “normal” does not form a spherical cluster but instead has a complex or irregular shape, which is accomplished through the kernel function (also known as the kernel trick). However, these advantages come with trade-offs: kernel methods typically require more computational resources and careful parameter tuning (e.g., `kernel`, `gamma`, and `nu` parameters).

Building on our coffee shop analogy from previous recipes, where we considered customers waiting in line and a person sitting alone in their car in the parking lot, OCSVM would construct a boundary that best captures the region where most customers are concentrated—perhaps the area around the counter and the waiting line. The person sitting in their car, far from the counter, would be classified as an outlier because they fall outside this boundary.

However, OCSVM's strength lies in its ability to use kernel functions to create highly flexible boundaries. For instance, it could define a curved boundary that accurately represents the complex flow of customers within the coffee shop, capturing patterns that a simple linear boundary would miss. This flexibility enables OCSVM to identify subtle outliers that might be overlooked by other methods.

How to do it...

In this recipe, you will continue to work with the `tx` DataFrame to detect outliers using the OCSVM class from the PyOD library:

1. Start by loading the OCSVM class:

```
from pyod.models.ocsvm import OCSVM
```

2. There are a few parameters that you should be familiar with that are specific to OCSVM to control the algorithm's behavior, such as `kernel`, `gamma`, and `nu`, in addition to the `contamination` parameter in all PyOD implementations. For now, you will keep the default values for all these parameters, except for `contamination`, which you will change to 0.03 to be consistent with the other recipes.

Instantiate OCSVM, then fit it to the `tx` DataFrame:

```
ocsvm = OCSVM(contamination=0.03,
               kernel='rbf',
               gamma='auto',
               nu=0.5)

ocsvm.fit(tx)

>>
OCSVM(cache_size=200, coef0=0.0, contamination=0.03, degree=3,
       gamma='auto', kernel='rbf', max_iter=-1, nu=0.5,
       shrinking=True, tol=0.001, verbose=False)
```

3. Use the `predict` method to identify outliers. The method will output either 1 or 0 for each data point where 1 indicates an outlier. Store the results in a pandas Series.

```
ocsvm_pred = pd.Series(ocsvm.predict(tx), index=tx.index)
print('Number of OCSVM outliers = ', ocsvm_pred.sum())

>>
Number of OCSVM outliers = 5
```

4. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from ocsvm_pred
ocsvm_outliers = ocsvm_pred[ocsvm_pred == 1]
ocsvm_outliers = tx.loc[ocsvm_outliers.index]
print(ocsvm_outliers)
>>
```

	value
timestamp	
2014-08-09	15499.708333
2014-11-18	15499.437500
2014-11-27	10899.666667
2014-12-24	12502.000000
2015-01-05	12502.750000

Interestingly, it captured one out of the five known dates (2014-11-27, which was Thanksgiving Day). It did capture the day before Christmas (2014-12-24) but missed Christmas Day.

One-Class SVM (OCSVM) works by learning a decision boundary that encloses the “normal” data points in your time series. It maps the data into a higher-dimensional space using the kernel function (in this case, the radial basis function or 'rbf') and attempts to separate the data from the origin with maximum margin. The `nu` parameter controls the upper bound on the fraction of training errors and the lower bound on the fraction of support vectors. For time-series data, using OCSVM as is treats each observation as an independent data point, making it primarily suited for point outlier detection without temporal context.



For different options on how to include temporal context, you can review the first recipe, *Detecting outliers using distance-based algorithms*.

5. Use the `plot_outliers` function created in the *Technical requirements* section to visualize the output to gain better insight:

```
plot_outliers(ocsvm_outliers, tx, 'OCSVM')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the X markers are based on the outliers identified using the OCSVM algorithm:

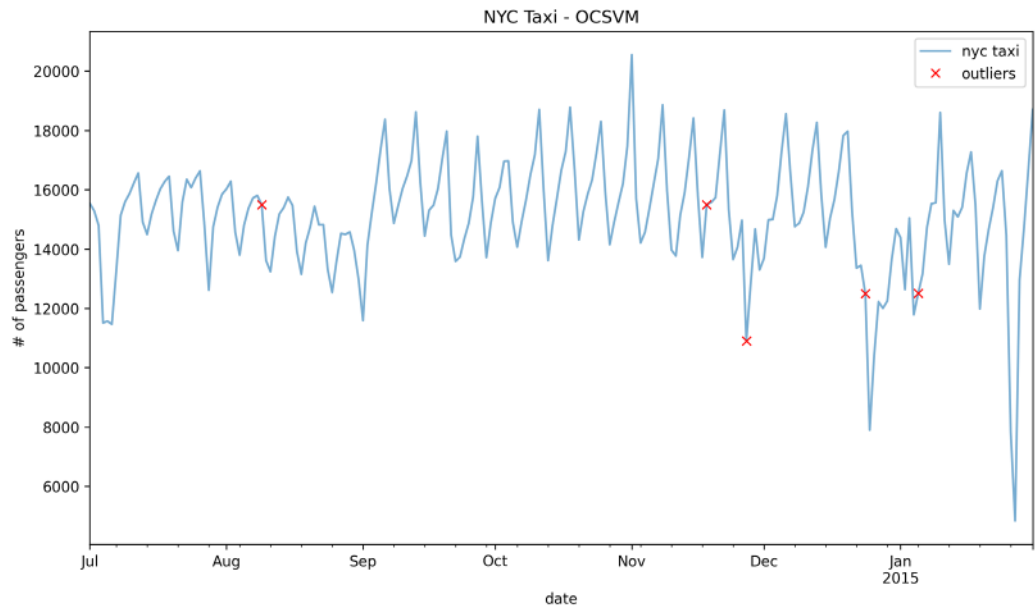


Figure 13.12: Markers showing the identified potential outliers using the OCSVM algorithm

When examining the plot in *Figure 13.12*, it is not clear why OCSVM picked up on those dates as being outliers. The RBF kernel can capture non-linear relationships, so you would expect it to be a robust kernel.

One possible reason for this inaccuracy is that SVM is sensitive to data scaling. OCSVM is particularly sensitive to the scale of the input features because it computes distances between points in the feature space. Without scaling, features with larger magnitudes can dominate the distance calculation and bias the results. The PyOD library provides a utility function called `standardizer` that transforms the data to have a mean of zero and a standard deviation of 1 (basically, it's z-score normalization).

6. Let's fix this issue and standardize the data, and then rerun the algorithm:

```
from pyod.utils.utility import standardizer
ocsvm = OCSVM(contamination=0.03,
               kernel='rbf',
               gamma='auto',
               nu=0.5)

scaled = standardizer(tx)
ocsvm.fit(scaled)
ocsvm_pred_sc = pd.Series(ocsvm.predict(scaled),
```



```

index=tx.index)

# extract the outlier values and dates from ocsvm_pred_sc
ocsvm_outliers_sc = ocsvm_pred_sc[ocsvm_pred_sc == 1]
ocsvm_outliers_sc = tx.loc[ocsvm_outliers_sc.index]
print(ocsvm_outliers_sc)
>>

```

timestamp	value
2014-07-06	11464.270833
2014-11-01	20553.500000
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

It's interesting to see how the output is completely different now after scaling the dataset. The model identified four out of the five known outlier dates and only missed New Year's Day.

7. Use the `plot_outliers` function on the new result set:

```
plot_outliers(ocsvm_outliers_sc, tx, 'OCSVM after scaling')
```

The preceding code should produce a more reasonable plot, as shown in the following figure:

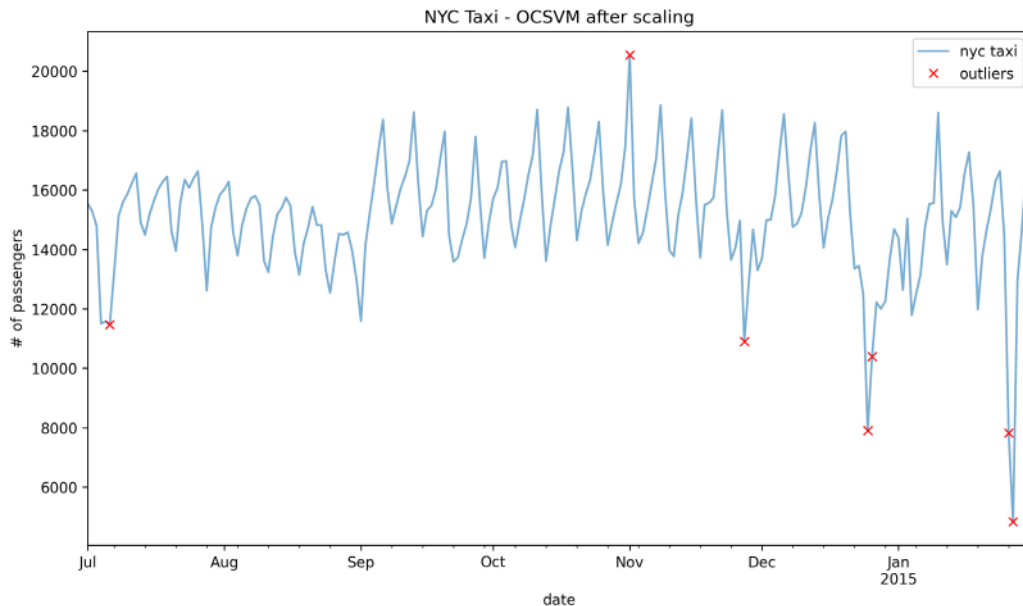


Figure 13.13: OCSVM after scaling the data using the `standardizer` function from PyOD

Compare the results from *Figure 13.12* and *Figure 13.13* to see how scaling made an impact on how the OCSVM algorithm identified outliers.

How it works...

One-Class Support Vector Machine (OCSVM) is a kernel-based algorithm that learns a decision boundary around normal data points. It works by mapping the data into a higher-dimensional feature space using a kernel function (in this case, the Radial Basis Function or RBF kernel), then finding the maximum margin hyperplane that separates the data from the origin.

The PyOD implementation for OCSVM is a wrapper to scikit-learn's `OneClassSVM` implementation, but with additional functionalities.

The key parameters that control OCSVM's behavior include the following:

- **kernel:** Determines the type of kernel used for the algorithm. The default `'rbf'` (Radial Basis Function) kernel is effective for capturing non-linear relationships in the data.
- **gamma:** This is the kernel coefficient for `'rbf'`, `'poly'`, and `'sigmoid'` kernels, with a default value of `'auto'`. Gamma controls how far the influence of each data point reaches, which directly shapes the decision boundary around normal data points. Higher gamma values create tighter, more complex boundaries that might lead to overfitting, while lower gamma values produce smoother, more generalized boundaries. With `'auto'`, it's set to $1/n\_features$ where `n_features` is the number of input features (in our recipe, that's $1/1=1$). When tuning OCSVM, experimenting with different gamma values is often necessary, as the optimal value depends on your specific dataset's characteristics.
- **nu:** The `nu` parameter (between 0 and 1) serves dual purposes: it sets an upper bound on the fraction of training errors (outliers) and a lower bound on the fraction of support vectors. In practice, it helps control how tightly the decision boundary fits around your normal data points (a trade-off between capturing all normal data points and minimizing the boundary volume). Lower `nu` values create tighter boundaries that might exclude some normal instances but result in smaller boundary volumes, while higher `nu` values create looser boundaries that capture more normal instances at the cost of larger boundary volumes.
- **contamination:** The expected proportion of outliers in the dataset. While both `contamination` and `nu` affect the decision boundary, `contamination` is more directly interpretable as it represents your prior knowledge about the expected proportion of outliers in your dataset.

As demonstrated in this recipe, OCSVM is highly sensitive to the scale of input features. Without proper scaling, the algorithm may produce unreliable results because distance-based calculations in the kernel space can be dominated by features with larger magnitudes. The standardization step transforms the data to have zero mean and unit variance, which allows OCSVM to treat all features equally.

After scaling, the algorithm successfully identified most of the known anomalies in the NYC taxi dataset, demonstrating the importance of proper data preprocessing when working with kernel-based methods.

There's more...



For deeper inspection of results, you can access the `decision_scores_`, `threshold_`, and `predict_proba` attributes, which are available across all PyOD models. For a refresher on interpreting these attributes and methods to incorporate temporal context (such as sliding windows, feature engineering, or using PyODAdapter), refer to the first recipe in this chapter, *Detecting outliers using distance-based algorithms*.

Let's explore how the different kernels perform on the same dataset. In the following code, you will test four kernels: 'linear', 'poly', 'rbf', and 'sigmoid'.

Recall that when working with SVM, you will need to scale your data, so you will use the scaled dataset created earlier:

```
for kernel in ['linear', 'poly', 'rbf', 'sigmoid']:
    ocsvm = OCSVM(contamination=0.03, kernel=kernel)
    ocsvm.fit(scaled)
    ocsvm_pred_sc = pd.Series(ocsvm.predict(scaled),
                             index=tx.index,
                             name=kernel)

    ocsvm_outliers_sc = ocsvm_pred_sc[ocsvm_pred_sc == 1]
    ocsvm_outliers_sc = tx.loc[ocsvm_outliers_sc.index]
    print(f"Outliers using {kernel} kernel: \n{ocsvm_outliers_sc}")
    #plot_outliers(ocsvm_outliers_sc, tx, f"OCSVM using {kernel} kernel")
```

Interestingly, each kernel method captured slightly different outliers:

- The linear kernel mainly identified high-value outliers at the upper range
- The polynomial, RBF, and sigmoid kernels identified more of the known anomaly dates
- Both RBF and sigmoid kernels showed nearly identical results, suggesting similar decision boundaries in this dataset

For visual comparison, you can uncomment the `plot_outliers` line and rerun the code.

See also

- To learn more about the OCSVM implementation, visit the official documentation here: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.ocsvm>.

Detecting outliers using ensemble method algorithms

In this recipe, we explore ensemble methods for outlier detection, which identify anomalies by combining multiple models to improve detection accuracy and robustness. Specifically, we will explore the Isolation Forest (IForest) algorithm, an effective tree-based method that excels at identifying outliers by isolating them rather than modeling their behavior.

While distance-based algorithms such as KNN and LOF examine proximity between points, clustering-based methods such as CBLOF organize data into groups, probabilistic approaches such as COPOD model statistical distributions, and kernel-based methods such as OCSVM learn complex decision boundaries, ensemble methods take a different approach. They build multiple weak detectors and combine their outputs to achieve superior performance, particularly in high-dimensional spaces where other methods may struggle. In PyOD, you will be using the IForest class, which is a wrapper to scikit-learn's IsolationForest class.

The name “Isolation Forest” reflects its core mechanism: it creates an ensemble of isolation trees to detect anomalies. While it shares the “forest” concept with Random Forest (building multiple trees), Isolation Forest is a distinct algorithm specifically designed for unsupervised outlier detection, not an adaptation of Random Forest. In scikit-learn, IsolationForest is implemented using ExtraTreeRegressor (Extremely Randomized Trees) as its base estimator, which employs random splitting rather than optimized splits like Random Forest. Unlike supervised ensemble methods that learn to predict labels or values, Isolation Forest constructs trees by randomly partitioning the feature space until each data point is isolated, with the key insight that anomalies require fewer splits to isolate than normal points.



For a refresher on ensemble learning and algorithms for supervised learning, including several tree-based algorithms such as Random Forest, Gradient Boosting, and XGBoost, review *Chapter 11*.

The premise behind IForest is that outliers are easier to isolate than normal points. The algorithm works by randomly partitioning the dataset into multiple subspaces, creating isolation trees. This is performed recursively until all data points are isolated. The key insight is that outliers require fewer partitions (splits) to become isolated compared to normal points. Since anomalies are typically few and different, they tend to be isolated in fewer steps, resulting in shorter paths in the isolation trees. The average path length across multiple trees becomes the basis for the outlier score—shorter paths indicate a higher likelihood of being an outlier. We can summarize the intuition behind the algorithm as follows:

- Normal points are densely packed and require many splits to isolate
- Outliers are few and distant, so they're isolated in far fewer splits

Continuing our coffee shop scenario, let's explore how IForest approaches outlier detection. Picture our familiar scene: customers waiting in line inside the shop, and that lone person sitting in their car in the parking lot. IForest would analyze this scenario by repeatedly drawing imaginary lines that divide the space, creating smaller and smaller regions until each person is isolated. In this process, the person waiting alone in their car would typically require fewer dividing lines to isolate compared to the densely packed customers in line.

For instance, a single dividing line might already separate the car person from everyone else in the coffee shop (e.g., inside versus outside), while it would take many more dividing lines to separate individuals within the crowded line. IForest would repeat this random partitioning process multiple times (creating multiple trees), and then calculate how many partitions, on average, it takes to isolate each person. The person in the car consistently requiring fewer partitions across multiple trees would receive a higher outlier score.



What makes IForest particularly powerful is that it doesn't just make this assessment once—it repeats the process hundreds of times (e.g., `n_estimators=100`) with different random splits, then averages the results. This ensemble approach makes IForest robust against noise and capable of detecting various types of outliers that might be missed by single-model approaches.

How to do it...

In this recipe, you will continue to work with the `tx` DataFrame to detect outliers using the IForest class from the PyOD library:

1. Start by loading the IForest class:

```
from pyod.models.iforest import IForest
```

There are a few key parameters that you should be familiar with that are specific to iForest to control the algorithm's behavior, such as `n_estimators`, `max_features`, and `bootstrap`, in addition to the `contamination` parameter in all PyOD implementations. For now, you will keep the default values for all these parameters, except for `contamination`, which you will change to 0.03 to be consistent with the other recipes.

Since the iForest algorithm randomly generates partitions for the data, setting the `random_state` parameter ensures that your work is reproducible. This way, you'll get consistent results back when you rerun the code. For example, you can use `random_state=45`.

Instantiate IForest with updates to the `contamination` and `random_state` parameters, then fit it to the `tx` DataFrame:

```
iforest = IForest(contamination=0.03,
                  n_estimators=100,
                  bootstrap=False,
                  random_state=45)

iforest.fit(tx)

>>
IForest(behaviour='old', bootstrap=False, contamination=0.03, max_
features=1.0,
        max_samples='auto', n_estimators=100, n_jobs=1, random_state=45,
        verbose=0)
```

2. Use the `predict` method to identify outliers. The method will output either 1 or 0 for each data point, where 1 indicates an outlier. Store the results in a pandas Series:

```
iforest_pred = pd.Series(iforest.predict(tx), index=tx.index)
print('Number of IForest outliers = ', iforest_pred.sum())
>>
Number of IForest outliers = 7
```

3. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from iforest_pred
iforest_outliers = iforest_pred[iforest_pred == 1]
iforest_outliers = tx.loc[iforest_outliers.index]
print(iforest_outliers)
>>
```

	value
timestamp	
2014-11-01	20553.500000
2014-11-08	18857.333333
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

IForest captured four of the five known outlier dates (missing New Year's Day), while detecting one additional date (November 8) worth investigating.

4. Use the `plot_outliers` function created in the *Technical requirements* section to visualize the output to gain better insight:

```
plot_outliers(iforest_outliers, tx, 'IForest')
```

The preceding code should produce a plot similar to that in *Figure 13.1*, except the X markers are based on the outliers identified using the iForest algorithm:

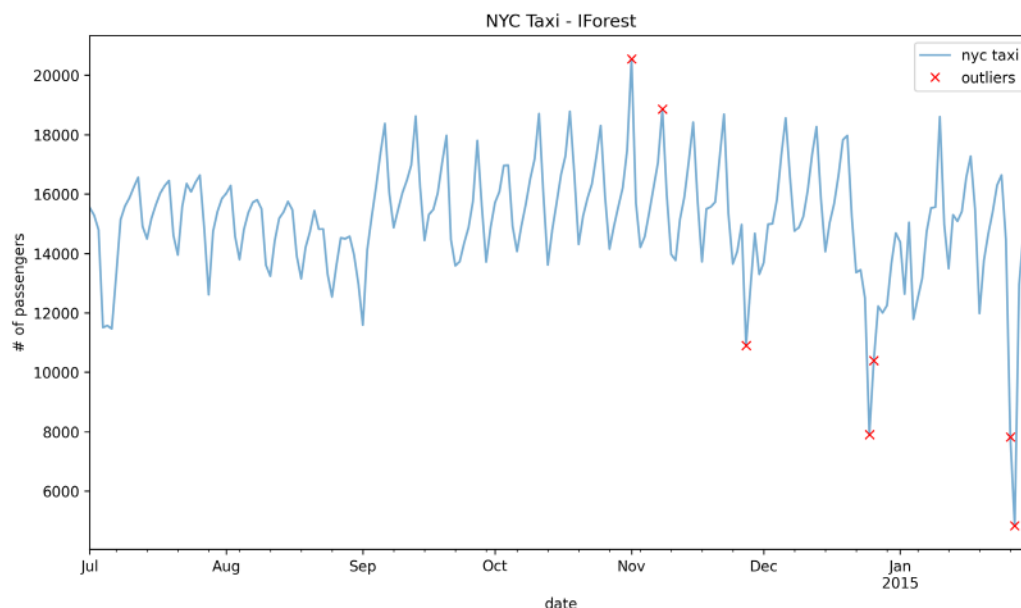


Figure 13.14: Markers showing the identified potential outliers using the IForest algorithm

How it works...

As an ensemble method, IForest will be creating multiple models (isolation trees). The default number of base estimators or trees is 100 (`n_estimator=100`). Increasing `n_estimators` may improve model performance up to a certain level before the computational performance takes a hit. So, for example, think of the number of estimators as the number of isolation trees created. For instance, for 100 estimators, you are essentially creating 100 isolation trees that collectively determine which points are outliers.

There is one more parameter worth mentioning, which is the `bootstrap` parameter. It is a Boolean set to `False` by default. Since IForest randomly samples the data, you have two options: random sampling with replacement (known as *bootstrapping*) or random sampling without replacement. The default behavior is sampling without replacement (`bootstrap=False`). Setting `bootstrap=True` can increase the diversity of trees in the forest, potentially improving robustness to noise. However, it may also reduce the algorithm's ability to isolate true anomalies if they are underrepresented in some bootstrap samples.

The core of IForest's effectiveness lies in how it calculates anomaly scores from path lengths. Each isolation tree recursively splits the data by randomly selecting a feature and splitting the value until either of the following happens:

- A single point remains (isolated)
- The maximum tree depth has been reached

The `max_samples` parameter indirectly controls the maximum depth of each tree. In scikit-learn, the maximum depth of each tree is set to `ceil(log2(max_samples))`, so smaller `max_samples` values result in shallower trees. For each point, IForest calculates the average path length across all trees (the number of splits required to isolate it). Shorter paths lead to higher outlier scores. Scores are normalized to (0, 1), where values closer to 1 indicate stronger outliers and scores closer to 0.5 represent normal points.

Several other parameters can further fine-tune IForest's behavior:

- `max_samples`: Determines the subset size used for building each tree (default="auto", which uses `min(256, n_samples)`). While this parameter affects tree depth, as mentioned earlier, it also controls the balance between computational speed and model precision. Values closer to the dataset size may better capture global patterns but increase the processing time, while smaller values provide faster training with potentially more diverse trees.
- `max_features`: Controls how many features are considered when making each split (default=1.0 for 100% of features). Reducing this value in high-dimensional datasets can increase tree diversity and potentially improve performance.
- `random_state`: Ensures reproducibility of results across runs, which is essential when comparing different models or configurations.
- `n_jobs`: Allows parallelization across multiple CPU cores (default=1), with -1 using all available cores for faster processing of large datasets.

The `contamination` parameter we set to 0.03 defines the expected proportion of outliers in the dataset, which influences the threshold used to distinguish between normal points and anomalies.

There's more...



For deeper inspection of results, you can access the `decision_scores_`, `threshold_`, and `predict_proba` attributes, which are available across all PyOD models. For a refresher on interpreting these attributes and methods to incorporate temporal context (such as sliding windows, feature engineering, or using `PyODAdapter`), refer to the first recipe in this chapter, *Detecting outliers using distance-based algorithms*.

Isolation Forest was introduced in 2008 (<https://dl.acm.org/doi/10.1145/2133360.2133363>) and emerged as a highly efficient ensemble method for outlier detection, especially when handling large-scale datasets due to its linear time complexity. While effective in many scenarios, standard IForest relies on random splits and may fail to capture non-linear anomaly patterns, limiting its performance.

Deep Isolation Forest (DIF) was proposed as an advancement over the standard Isolation Forest in a paper published in 2023 (<https://arxiv.org/pdf/2206.06602>). DIF integrates Isolation Forest with deep learning by using randomly initialized deep learning networks (DNNs) to transform the original data into multiple representation spaces. This transformation enables non-linear partitioning of the data, making underlying outlier patterns more apparent, so it becomes easier for the Isolation Forest step to isolate.

Luckily for us, PyOD provides an implementation of DIF that you can use with a familiar API like other PyOD algorithms. In this example, we use mostly default parameters to demonstrate DIF's effectiveness even without extensive tuning. However, when using DIF, you might find it beneficial to work directly with **decision scores** rather than the `predict` method, as this approach gives you more direct control over the threshold selection process, especially in cases where the built-in threshold may not align perfectly with your specific dataset characteristics:



The **Deep Isolation Forest (DIF)** implementation in PyOD requires PyTorch as a dependency. If you encounter an error about Torch being missing, you can install it using the following:

```
pip install torch
```

For platform-specific installation instructions (including GPU support), refer to PyTorch's official installation guide:

<https://pytorch.org/get-started/locally/>

```
import pandas as pd
from pyod.models.dif import DIF
import warnings
warnings.filterwarnings('ignore')

# Initialize and fit the DIF model
dif = DIF(contamination=0.03, n_estimators=100, random_state=42)
dif.fit(tx)
```



For the **Deep Isolation Forest (DIF)** algorithm, we use `decision_function()` instead of `decision_scores_` to obtain anomaly scores. This is because of a known implementation issue where `decision_scores_` may be affected by double normalization during training. Using `decision_function()` ensures more consistent and accurate anomaly scores for both training and new data.

```
# Work with decision scores directly
dif_scores = dif.decision_scores_
# Set threshold based on expected contamination rate (e.g., 3%)
dif_threshold = np.quantile(dif_scores, 0.97) # Adjust percentile as needed
dif_outliers = tx[dif_scores > dif_threshold]
print(dif_outliers)

>>

          value
timestamp
2014-09-13  18617.729167
```

```
2014-10-11 18706.416667
2014-10-18 18778.958333
2014-11-01 20553.500000
2014-11-08 18857.333333
2014-11-22 18688.062500
2015-01-31 18702.479167
```

Unlike IForest, which successfully identified 4 out of our 5 known anomalies, DIF only catches November 1st. The other 6 flagged dates appear to be normal high-volume days rather than the holiday anomalies we're looking for. This difference in results, despite their architectural similarities, suggests the following:

- For this particular univariate time series dataset, DIF's non-linear partitioning capabilities don't provide advantages over standard IForest. In fact, IForest performs better.
- The anomalies in our NYC taxi data are distinct enough that simpler methods like IForest capture them more effectively.
- For more complex datasets with higher dimensionality or subtler anomaly patterns, DIF's advanced architecture might show more pronounced benefits.

See also

The PyOD iForest implementation is actually a wrapper to the IsolationForest class from scikit-learn:

- To learn more about **IForest** from PyOD and the different parameters available, visit their official documentation here: <https://pyod.readthedocs.io/en/latest/pyod.models.html?highlight=knn#module-pyod.models.iforest>
- To learn more about the IsolationForest class from scikit-learn, you can visit their official documentation page here: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html#sklearn-ensemble-isolationforest>
- To learn more about Deep Isolation Forest (**DIF**) from PyOD and the different parameters available, visit their official documentation here: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.dif>

Detecting outliers using deep learning-based algorithms (AutoEncoder)

In this recipe, we explore deep learning-based outlier detection algorithms, which identify anomalies by leveraging neural networks to learn complex data representations. Specifically, we will focus on **AutoEncoders**, a type of neural network architecture designed for unsupervised learning that excels at capturing intricate, non-linear patterns in data.

While distance-based methods, such as KNN and LOF, examine proximity between points, clustering-based methods, such as CBLOF, organize data into groups, probabilistic approaches, such as CO-POD, model statistical distributions, kernel-based methods, such as OCSVM, learn complex decision boundaries, and ensemble methods, such as IForest, isolate anomalies through random partitioning, deep learning-based methods represent another approach. AutoEncoders learn to compress normal data patterns into a lower-dimensional space (**encoding**) and then reconstruct the original data (**decoding**), with the premise that normal data reconstructs well, while outliers will produce larger reconstruction errors, which become outlier scores.

In *Chapter 12*, we focused on supervised learning techniques for creating forecasting models. Deep learning, like machine learning techniques, can be **supervised** or **unsupervised**. AutoEncoders fall under the unsupervised learning category, as they don't require labeled data to learn the underlying patterns. They operate on a simple yet powerful principle: compress the data into a condensed representation (**latent space**) and then decompress it to recreate the original input. The network is trained to minimize the reconstruction error between the input and output.

The architecture of an AutoEncoder consists of two main components:

- **Encoder:** Compresses input data into a lower-dimensional **latent space** (the **bottleneck**)
- **Decoder:** Reconstructs the original data from this compressed representation

A characteristic of AutoEncoders is their bottleneck architecture, where the number of neurons decreases in the encoder (e.g., from input size to 64 to 32) and then increases in the decoder (e.g., from 32 back to 64, and finally to the original input size). This type of architecture, by having the data go through a narrow layer, forces the model to learn and preserve the most essential patterns and features while discarding noise or redundant information. This creates a compressed representation of the data that we call latent space, effectively performing non-linear dimensionality reduction and capturing essential data characteristics in the compressed latent space representation.

For outlier detection, the trained AutoEncoder computes reconstruction errors by measuring the distance between the original input and its reconstruction. The underlying assumption is that the model will learn to better reconstruct common patterns (normal data) while struggling with rare patterns (outliers), resulting in higher reconstruction errors for anomalies. The contamination parameter determines the threshold for classifying points as outliers based on their reconstruction errors.

What makes AutoEncoders particularly powerful is their ability to capture complex non-linear relationships and hierarchical patterns in the data that simpler models might miss. AutoEncoders excel at identifying **representation-based outliers**, detecting points that cannot be efficiently compressed and reconstructed by the learned model. This approach is especially effective for high-dimensional time-series data where traditional methods might suffer from the “curse of dimensionality.”

The PyOD implementation offers flexibility in model architecture through parameters such as `hidden_neuron_list`, which controls the bottleneck structure, and `dropout_rate`, which helps prevent overfitting. The model is trained directly on the dataset containing both normal data and potential outliers, using mean squared error (MSE) as the loss function to minimize the reconstruction error. Training is conducted over multiple **epochs**, where each epoch represents one complete pass through the entire dataset. The number of epochs (`epoch_num` parameter) significantly impacts the model's performance.

However, these advantages come with trade-offs: AutoEncoders typically require larger amounts of training data and more computational resources compared to simpler outlier detection methods. The performance is also sensitive to hyperparameters such as the number of hidden layers, neurons per layer, learning rate, and epoch count.



AutoEncoders do not inherently account for temporal dependencies in time-series data. The PyOD library implementation is not designed to capture temporal patterns in the data or accommodate for seasonality or trends. Using PyOD as is will help us capture **point outliers**, where each time point is treated independently. For **contextual outliers**, where a value's status as an anomaly depends on its temporal context, you should refer to the first recipe, *Detecting outliers using distance-based algorithms*, in which we discussed different techniques to incorporate temporal context into your feature engineering process.

Getting ready

The AutoEncoder implementation in PyOD requires PyTorch as a dependency. If you encounter an error about Torch being missing, you can install it using the following:

```
pip install torch
```

For platform-specific installation instructions (including GPU support), refer to PyTorch's official installation guide:

<https://pytorch.org/get-started/locally/>

How to do it...

In this recipe, you will continue to work with the tx DataFrame to detect outliers using the AutoEncoder class from the PyOD library:

1. Start by loading the AutoEncoder class:

```
from pyod.models.auto_encoder import AutoEncoder
```

There are a few key parameters that you should be familiar with that are specific to AutoEncoder to improve the model's performance, such as `lr` (learning rate), `epoch_num` (number of training iterations), `batch_size` (samples per batch), in addition to the `contamination` parameter in all PyOD implementations. For now, you will use the default values for all these parameters and only change `contamination` to 0.03 to be consistent with the other recipes.

Instantiate AutoEncoder with `contamination=0.03`, then fit it to the tx DataFrame. The code below shows the default values for the key training parameters explicitly:

```
# lr: Learning rate for optimization
# epoch_num: Number of training iterations
# batch_size: Number of samples per training batch
auto_encoder = AutoEncoder(contamination=0.03,
```

```
lr=0.001,
epoch_num=10,
batch_size=32)

auto_encoder.fit(tx)
```

2. Use the predict method to identify outliers. The method will output either 1 or 0 for each data point where 1 indicates an outlier. Store the results in a pandas Series:

```
ae_predicted = pd.Series(auto_encoder.predict(tx), index=tx.index)
print('Number of AutoEncoder outliers = ', ae_predicted.sum())
>>
Number of AutoEncoder outliers = 7
```

3. Filter the outputs to only show the outlier values and associated dates:

```
# extract the outlier values and dates from ae_predicted
ae_outliers = ae_predicted[ae_predicted == 1]
ae_outliers = tx.loc[ae_outliers.index]
```

In the previous recipe, we discussed that you might find it beneficial to work directly with **decision scores** rather than the predict method if you prefer more control over the threshold and work directly with the scores. The following will produce the same results:

```
ae_scores = pd.Series(auto_encoder.decision_scores_,
                      index=tx.index)

threshold = auto_encoder.threshold_
# or you can do it using quantile
threshold = ae_scores.quantile(0.97)
ae_outliers = tx[ae_scores > ae_scores.quantile(0.97)]
print(ae_outliers)
>>

           value
timestamp
2014-09-23  13729.354167
2014-09-29  13715.166667
2014-11-17  13722.791667
2014-11-24  13652.625000
2014-12-01  13683.625000
2014-12-30  13659.687500
2015-01-20  13759.416667
```

The results differ significantly from those obtained with previous algorithms. Before analyzing these outputs further, it's important to note that deep learning models generally require more careful parameter tuning than traditional methods. The default setting of 10 epochs is quite low for AutoEncoders and likely insufficient for the model to converge and learn the complex patterns in our time-series data.

4. Let's rerun the model, but this time with `epoch_num=1000`.

```

auto_encoder = AutoEncoder(contamination=0.03,
                             lr=0.001,
                             epoch_num=1000,
                             batch_size=32)

auto_encoder.fit(tx)
ae_predicted = pd.Series(auto_encoder.predict(tx),
                          index=tx.index)
# extract the outlier values and dates from ae_predicted
ae_outliers = ae_predicted[ae_predicted == 1]
ae_outliers = tx.loc[ae_outliers.index]
print(ae_outliers)
>>

```

	value
timestamp	
2014-11-01	20553.500000
2014-11-08	18857.333333
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

The results are much better now. With more training epochs, the model has more opportunities to learn subtle patterns in the data, leading to more accurate reconstructions of normal data and better differentiation of outliers. The model captured 4 out of the 5 known outliers (missing New Year's Day). In fact, the results are exactly the same as the ones obtained from the previous recipe using ensemble algorithms such as IForest and DIF.

AutoEncoders provide a powerful tool for detecting anomalies in high-dimensional time-series data by learning compressed representations of normal patterns and identifying deviations through reconstruction errors. While they excel at capturing non-linear relationships, their performance depends heavily on proper hyperparameter tuning and sufficient training data.

How it works...



For deeper inspection of results, you can access the `decision_scores_`, `threshold_`, and `predict_proba` attributes, which are available across all PyOD models. For a refresher on interpreting these attributes and methods to incorporate temporal context (such as sliding windows, feature engineering, or using PyODAdapter), refer to the first recipe in this chapter, *Detecting outliers using distance-based algorithms*.

The AutoEncoder functions as an outlier detector by learning to reconstruct normal patterns and identifying data points that don't reconstruct well. The model is trained to minimize the **Mean Squared Error (MSE)** between input data and its reconstruction. Points with high reconstruction errors—those that the model struggles to accurately reproduce—are flagged as potential outliers. The contamination parameter determines the threshold above which reconstruction errors are classified as outliers.

Several parameters in PyOD's AutoEncoder will be familiar to readers who explored deep learning models in *Chapter 12*, though their specific impact on outlier detection deserves attention:

Here are some key parameters in PyOD's AutoEncoder:

- **preprocessing** (default=True): Controls whether input data is standardized (scaled to zero mean and unit variance). When enabled, the model calculates statistics from training data and applies consistent scaling during both training and prediction. This improves convergence and ensures features with different scales contribute equally. *Disable only if you've already standardized your data or need to preserve original scales.*
- **contamination** (default=0.1): This sets the proportion of outliers in the data, used to determine the threshold for classifying points as outliers based on reconstruction errors. This is a common parameter across all PyOD algorithms.
- **lr** (default=0.001): This is the learning rate, which controls how quickly the model learns (adjusts its weights). In deep learning, the learning rate is like the step size the model takes to adjust its weights during training. Higher values (e.g., 0.01) can speed up training but risk instability and overshoot the optimal solution, while lower values (e.g., 0.0001) may slow convergence but yield stable training due to finer adjustments.
- **epoch_num** (default=10): The number of complete training passes through the dataset. Too few epochs can yield to underfitting (failing to learn patterns), while too many may improve accuracy but can lead to overfitting (memorizing noise). In our example, epoch_num=1000 yielded better results.
- **batch_size** (default=32): The number of data points per training step. It affects training speed and memory usage, with larger sizes offering stability but requiring more memory.
- **hidden_neuron_list** (default=[64, 32]): Defines the encoder and decoder structure (bottleneck architecture). You specify the number of neurons in the encoder, since the decoder mirrors the structure but in reverse. This affects the model's capacity to learn complex patterns. For example, [64, 32] creates encoder layers (input → 64 → 32) and symmetric decoder layers (32 → 64 → output). Smaller values create a more restrictive bottleneck, forcing the model to learn more efficient representations but potentially losing important information, while larger values can risk overfitting. For high-dimensional time series, you might need larger values to capture complex patterns.
- **hidden_activation_name** (default= 'relu'): Specifies the activation function used in the hidden layers, controlling the non-linear transformations. Common choices include 'relu', 'tanh', and 'sigmoid'.
- **dropout_rate** (default=0.2): A regularization technique to prevent overfitting by randomly dropping neurons (0.2 indicates a 20% dropout).

- `batch_norm` (default=True): Batch normalization stabilizes and accelerates training by normalizing layer inputs, which is particularly beneficial for deep networks. Keeping this enabled is generally recommended for time series data.

When working with AutoEncoders for outlier detection in time series, the most impactful parameters to tune are typically `epoch_num`, `hidden_neuron_list`, and `lr`. As we saw in our experiment, increasing the number of epochs from 10 to 1,000 dramatically improved the model's ability to detect known outliers. For complex time series with many features, you might also consider adjusting `hidden_neuron_list` to create a deeper or wider network architecture.

Unlike simpler outlier detection methods, AutoEncoders require more careful tuning but can capture complex non-linear relationships that other methods might miss.

The AutoEncoder matches ensemble methods (IForest, DIF) by detecting four of the five known outliers, missing only New Year's Day. Distance-based methods (KNN, LOF) and kernel-based approaches (OCSVM) show similar performance, while probabilistic approaches (COPOD, ECOD) vary more in their detection patterns. Notably, no method captured all five known outliers, highlighting the challenge of outlier detection even with significant events.

Each algorithm has its advantages in how it detects outliers. This is where the **SUOD (Scalable Unsupervised Outlier Detection)** framework comes into play. SUOD is a meta-framework that allows you to combine multiple outlier detection algorithms to leverage their collective strength while accelerating computation. It is designed to expedite the training of a large number of heterogeneous unsupervised outlier detection models and combines the results through aggregation methods such as the average or maximum (taking the max value across all base detectors). Here is an example using different PyOD outlier detection classes:



You may need to install SUOD which you can do using:

```
pip install suod
```

```
from pyod.models.suod import SUOD

# assuming these classes are already imported from prior recipes
knn = KNN(method='mean', n_neighbors=5)
lof = LOF(n_neighbors=20)

cblof = CBLOF(n_clusters=8, alpha=0.9, beta=5)
copod = COPOD()
ocsvm = OCSVM(kernel='rbf', gamma='auto', nu=0.5)
iforest = IForest(n_estimators=100, random_state=45)

# initialized a group of outlier detectors for acceleration
detector_list = [knn, lof, cblof, copod, ocsvm, iforest]
```



```

clf = SUOD(contamination=0.03,
           base_estimators=detector_list, # List of models to combine
           n_jobs=2, # Number of parallel jobs
           combination='average', # How to combine model predictions
           verbose=True)

clf.fit(tx)
# Predict outliers
clf_pred = pd.Series(clf.predict(tx),
                    index=tx.index)
print('Number of SUOD outliers = ', clf_pred.sum())

# extract the outlier values and dates from clf_pred
clf_outliers = clf_pred[clf_pred == 1]
clf_outliers = tx.loc[clf_outliers.index]
print(clf_outliers)
>>
Number of SUOD outliers = 7

```

	value
timestamp	
2014-07-06	11464.270833
2014-11-01	20553.500000
2014-11-27	10899.666667
2014-12-25	7902.125000
2014-12-26	10397.958333
2015-01-26	7818.979167
2015-01-27	4834.541667

Interestingly, SUOD detects 7 outliers, including 4 of our 5 known anomalies (2014-11-01, 2014-11-27, 2014-12-25, 2015-01-27), similar to several individual models. However, it also flags 2014-07-06 as an outlier, which only COPOD and OCSVM detected individually. This demonstrates how ensemble approaches like SUOD can incorporate diverse detection patterns from multiple algorithms.

Beyond improved detection capabilities, SUOD offers significant computational advantages, particularly for high-dimensional data or when running multiple detection algorithms. The framework implements several **acceleration** techniques, including random projection for dimensionality reduction and balanced parallel scheduling for distributed computing environments. You can learn more about the SUOD in the paper *SUOD: Accelerating Large-Scale Unsupervised Heterogeneous Outlier Detection* (<https://arxiv.org/abs/2003.05731>).

There's more...

In addition to the standard AutoEncoder, PyOD offers other deep learning models for outlier detection. **Variational AutoEncoders (VAEs)** provide another approach to outlier detection. Like the AutoEncoder, VAEs learn to reconstruct input data, but they do so by modeling a probability distribution over the latent space. This probabilistic approach allows VAEs not only to reconstruct input data but also to generate new samples from the learned distribution, making them a key component in generative modeling techniques often used in **Generative AI (GenAI)**. Additionally, the probabilistic nature of VAEs can sometimes capture more nuanced patterns in the data.

To compare results, you can replace the AutoEncoder with a VAE in your code:

```
from pyod.models.vae import VAE

# Lr: Learning rate for optimization
# epoch_num: Number of training iterations
# batch_size: Number of samples per training batch
vae = VAE(contamination=0.03,
          lr=0.001,
          epoch_num=1000,
          batch_size=32)

vae.fit(tx)

vae_predicted = pd.Series(vae.predict(tx),
                          index=tx.index)

# extract the outlier values and dates from vae_predicted
vae_outliers = vae_predicted[vae_predicted == 1]
vae_outliers = tx.loc[vae_outliers.index]
print(vae_outliers)

>>
               value
timestamp
2014-07-04  11511.770833
2014-07-06  11464.270833
2014-11-27  10899.666667
2014-12-25   7902.125000
2014-12-26  10397.958333
2015-01-26   7818.979167
2015-01-27   4834.541667
```

The VAE results show a different detection pattern than standard AutoEncoder, capturing all three holiday-related outliers (Thanksgiving, Christmas, and the January Blizzard), plus identifying July 4 (Independence Day) weekend anomalies, demonstrating how its probabilistic approach can detect different types of outliers while missing the New York Marathon event—reinforcing that combining multiple detection methods often provides the most comprehensive anomaly detection strategy.

See also

- To learn more about **AutoEncoder** from PyOD and the different parameters available, visit their official documentation here: https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.auto_encoder
- To learn more about **VAE** from PyOD and the different parameters available, visit their official documentation here: <https://pyod.readthedocs.io/en/latest/pyod.models.html#module-pyod.models.vae>

Get This Book's PDF Version and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

UNLOCK NOW



14

Advanced Techniques for Complex Time Series

As time series data becomes more complex, you will encounter multiple, overlapping seasonal patterns. For example, electricity consumption recorded hourly: over 24 months, you will spot annual patterns (such as spikes in the summer for cooling), weekly patterns (higher usage on weekdays compared to weekends), and daily cycles (with peaks in the evenings). These patterns don't exist in isolation; they are blended within the data.

With the rise of connected devices such as IoT sensors, we now collect data at much higher frequencies: hourly, every minute, or every second. This is a shift from classical time series research, where datasets were smaller and recorded less frequently, usually annually or monthly.

Many traditional statistical models, like **Seasonal Autoregressive Integrated Moving Average (SARIMA)**, were built for scenarios with a single dominant seasonal pattern. They are great for straightforward cases, but often will struggle when faced with complex data with multiple overlapping seasonalities. The same goes for classical decomposition methods like **Seasonal-Trend decomposition using LOESS (STL)**, which expects a single seasonal pattern at a time.

In previous chapters, you encountered algorithms that can handle time series with multiple seasonal patterns, such as **Prophet** in *Chapter 11* and **Neural Hierarchical Interpolation for Time Series Forecasting (N-HITS)** in *Chapter 12*. In this chapter, we will focus on methods specifically designed for handling time series data with overlapping seasonalities. You will explore multiple seasonal decompositions using **Multiple Seasonal-Trend Decomposition using LOESS (MSTL)**, and multi-seasonal forecasting with **Trigonometric, Box-Cox, ARMA errors, Trend, Seasonality (TBATS)**, **Unobserved Components Model (UCM)**, and **NeuralProphet**.

In this chapter, you will explore the following recipes:

- Decomposing time series with multiple seasonal patterns using MSTL
- Forecasting with multiple seasonal patterns using UCM
- Forecasting with multiple seasonal patterns using AutoTBATS
- Forecasting with multiple seasonal patterns using NeuralProphet

Technical requirements

In this chapter, you will be using the *Hourly Energy Consumption* data from Kaggle (<https://www.kaggle.com/datasets/robikscube/hourly-energy-consumption>). Throughout the chapter, you will be working with the same energy dataset to observe how the different techniques compare.

The ZIP folder contains 13 CSV files, which are available in the GitHub repository for this book and can be found here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition/tree/main/datasets/Ch14>.

The recipes of this chapter will use one of these files, but feel free to explore the other ones. The files represent hourly energy consumption measured in **megawatts (MW)**.

Start by loading the data as a pandas DataFrame and preparing the data for the recipes. You will also load the shared libraries used throughout the chapter:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
from statsmodels.tools.eval_measures import rmse, meanabs as mae
import warnings
warnings.filterwarnings('ignore')
plt.style.use('grayscale')
```

Read the `AEP_hourly.csv` file:

```
folder = Path('../..../datasets/Ch14/')
file = folder.joinpath('AEP_hourly.csv')
df = pd.read_csv(file, index_col='Datetime', parse_dates=True)
df.shape
>>
(121273, 1)
```

The raw data needs to be cleaned: the index is not in order and has both duplicate timestamps and gaps in the hourly readings. Let's sort the index, then use `resample('h')` to enforce a complete hourly timeline. This step fills any gaps, which is why the record count increases slightly. We'll use `change` it to `.last()` to aggregate any duplicates within an hour and finally use `.ffill()` to populate the newly created gaps.

```
df.sort_index(inplace=True)
df = df.resample('h').last()
df.columns = ['y']
df.ffill(inplace=True)
print(f"Final dataset shape: {df.shape}")
print(f>Data frequency: {df.index.freq}")
print(f"Any missing values: {df.isnull().any().any()}")
```

```
>>  
Final dataset shape: (121296, 1)  
Data frequency: <Hour>  
Any missing values: False
```

You should have a DataFrame with 121296 records of hourly energy consumption from October 2004 to August 2018 (around 14 years).

Plot the data for visual inspection:

```
df.plot()
```

This should produce the time series plot shown in *Figure 14.1* for the hourly energy consumption:

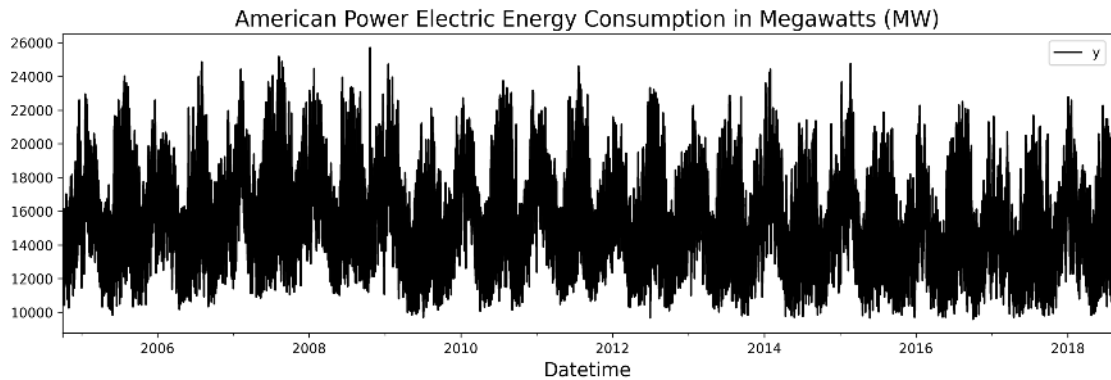


Figure 14.1: Plot of the hourly American Power Electric (AEP) energy consumption from October 2004 to August 2018

Understanding state-space models

In this chapter, you will see references to **state-space models (SSMs)**. In *Chapter 9*, you were introduced to exponential smoothing (*Holt-Winters*) and ARIMA-type models. Before defining what state-space models are, I want to point out that these models can be represented in a state-space formulation.

SSMs have their roots in the field of engineering (more specifically control engineering) and offer a generic approach to modeling dynamic systems and how they evolve over time. In addition, SSMs are widely used in other fields, such as economics, neuroscience, electrical engineering, and other disciplines. They're particularly useful for time series with the following:

- Multiple seasonality
- Missing data
- Non-stationary or non-linear patterns
- Measurement noise

In time series data, the central idea behind SSMs is that of **latent variables** (unobserved), also called **latent state**, that evolve over time (what we observe is not the whole story). For example, in a univariate time series, we have a response variable at time t ; this is the **observed** value termed Y_t which depends on a **latent variable** termed X_t . The X_t variable is the **unobserved** state that we are interested in estimating.

An SSM provides a system of equations to estimate these unobserved states from the observed values and is represented mathematically in a vector-matrix form. In general, there are two equations: one that describes how the hidden state evolves (*state equation*), and one that connects that state to what we actually observe (*observation equation*).

- **State equation:** $X_{t+1} = FX_t + w_t$
- **Observation equation:** $Y_t = HX_t + v_t$

Where F is the state transition matrix, H is the observation matrix, and w_t (process noise) and v_t (measurement noise) are noise terms.

One key aspect of their popularity is their flexibility and ability to work with complex time series data that can be multivariate, non-stationary, non-linear, or contain multiple seasonality, gaps, or irregular effects.

In addition, SSMs can be generalized and come in various forms, several of which make use of **Kalman filters** (Kalman recursions). The benefit of using SSMs in time series data is that they are used in *filtering*, *smoothing*, or *forecasting*, as you will explore in the different recipes of this chapter. Examples of popular SSMs for working with time series data that exhibits multiple, overlapping seasonality are TBATS and UCM, which we will cover in this chapter.

Kalman filters



The Kalman filter is an algorithm for extracting signals from data that is either noisy or contains incomplete measurements. The premise behind Kalman filters is that not every state within a system is directly observable; instead, we can estimate the state indirectly, using observations that may be contaminated, incomplete, or noisy.

For example, sensor devices produce time series data known to be incomplete due to interruptions or unreliable due to noise. Kalman filters are effective when working with time series data with a low signal-to-noise ratio (meaning a high amount of noise), as they work on smoothing and denoising the data to make the underlying signal more reliable.

Decomposing time series with multiple seasonal patterns using MSTL

In *Chapter 8*, the recipe *Decomposing time series data* introduced the fundamental concept of decomposition into trend, seasonal, and residual (remainder) components. In those examples, the time series contained a single seasonal pattern, which allowed it to be represented as a simple additive model:

$$Y_t = S_t + T_t + R_t$$

Where S_t , T_t , R_t represent the seasonal, trend, and remainder components at time t respectively. Using tools such as `seasonal_decompose` and `STL` from `statsmodels`, you explored how to interpret such additive decompositions.

But what about data with higher frequency – for example, IoT devices that can record data every second, minute, or hour? Such data may exhibit multiple seasonal patterns. Given the limitations of traditional seasonal decomposition techniques, a new approach for decomposing time series with multiple seasonality was introduced in a paper published by *Bandara, Hyndman, and Bergmeir*, titled *MSTL: A Seasonal-Trend Decomposition Algorithm for Time Series with Multiple Seasonal Patterns*. The paper was published in July 2021, which you can read here: <http://arxiv.org/abs/2107.13462>.

Multiple STL Decomposition (MSTL) is an extension of the STL algorithm and similarly is an additive decomposition, but it extends the equation to include multiple seasonal components and not just one:

$$Y_t = S_t^1 + S_t^2 + \dots + S_t^n + T_t + R_t$$

Here, n represents the number of seasonal cycles. The MSTL algorithm works by iteratively applying STL decomposition to each specified seasonal period in the data to isolate each seasonal component. After each STL application, the estimated seasonal component is subtracted from the series, allowing the next longer seasonal component to be isolated and extracted. This sequence avoids confounding effects between cycles.

Once the iterative process for each seasonal cycle is completed (and all seasonal components have been extracted $S_t^1 + S_t^2 + \dots + S_t^n$), then MST estimates the trend component. If the time series does not have any seasonal components, MSTL will only estimate the trend.

For example, with hourly electricity data, we might specify daily (24) and weekly (24x7=168) seasonal periods. MSTL would first extract the daily pattern, subtract it from the original series, and then extract the weekly pattern from the remainder. This stepwise extraction ensures each seasonal component is cleanly separated and interpreted.

Getting ready

The authors initially implemented the paper referenced earlier in the R language as the `mstl` function in the `forecast` package.

In Python, the algorithm is implemented in `statsmodel` as the `MSTL` class and was first introduced in version 0.14.0 (released May 2023).

You can install `statsmodels` using `pip`:

```
pip install statsmodels
```

You can also install it using `conda`:

```
conda install -c conda-forge statsmodels
```


How to do it...

Generally, high-frequency data can exhibit multiple seasonal patterns. In this recipe, you will decompose the daily energy consumption dataset into multiple seasonal components (daily, weekly, and annually).

1. Import the MSTL class from statsmodels:

```
from statsmodels.tsa.seasonal import MSTL
plt.rcParams["figure.figsize"] = [14, 10]
```

2. Create three variables to hold values for day, week, and year calculations; this way, you can reference these variables. For example, since the data is hourly, a day is represented as 24 hours, a week as 24 x 7, and a year as 24 x 365:

```
day = 24
week = day*7
year = day*365
print(f'''
    day = {day} hours
    week = {week} hours
    year = {year} hours
''')
>>
day = 24 hours
week = 168 hours
year = 8760 hours
```

3. Start by providing the different seasonal cycles you identified and suspect to exist in the *hourly energy consumption dataset*. In our example, this includes day, week, and year patterns that we identified in the previous step.

To improve computational performance, we will leverage the `stl_kwargs` parameter to pass jump values to the underlying LOESS smoother. We will set `seasonal_jump`, `trend_jump`, and `low_pass_jump` to 6. This instructs the algorithm to calculate the smoothed value for every 6 data points and uses linear interpolation for the points in between, which would significantly improve the processing speed.

```
jump = 6
mstl = MSTL(df,
             periods=(day, week, year),
             iterate=2,
             stl_kwargs={"seasonal_jump": jump,
                         "trend_jump": jump,
                         "low_pass_jump": jump})
```

4. Use the `fit()` method to estimate the three specified seasonal components ($S_{24}^1, S_{24 \times 7}^2, S_{24 \times 365}^3$), the overall trend, and finally, the remainder.

```
results = mstl.fit()
```

5. The results object is an instance of the `DecomposeResult` class, which gives access to the `seasonal`, `trend`, and `resid` attributes. The `seasonal` attribute will display a `DataFrame` with three columns labeled `seasonal_24`, `seasonal_168`, and `seasonal_8760`. You can plot each attribute individually or use the `plot` method to plot all the components:

```
results.plot()
plt.show()
```

The preceding code will produce six subplots: one each for observed, trend, seasonal_24 (daily), seasonal_168 (weekly), seasonal_8760 (annual), and resid (the remainder), respectively:

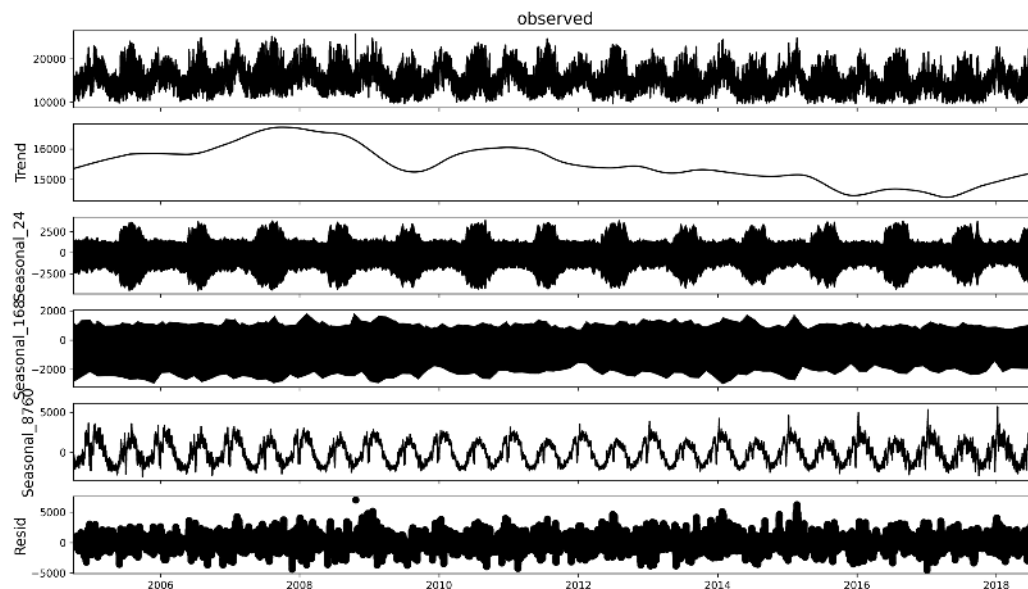


Figure 14.2: The MSTL decomposition plot for the energy consumption data

You can plot individual components – for example, to plot all three seasonal patterns only, you can use the following:

```
results.seasonal.plot(subplots=True)
plt.show()
```

Given the amount of data, it is hard to observe the extracted patterns. Let's slice the data for a zoom-in effect for a better visual perspective.

- With domain knowledge, one can make a general assumption to validate against. In energy consumption, you would expect a daily pattern where energy consumption peaks during the day and declines at nighttime, around bedtime. Additionally, you would anticipate a weekly pattern where consumption is higher on weekdays compared to weekends when people travel or do outdoor activities more. An annual pattern is also expected, with higher energy consumption peaking in the summer than in other cooler months. These different seasonal patterns also overlap; the daily pattern could potentially be different in summer seasons compared to winter seasons.

Let's examine this further by zooming in on a 24-hour period in **January** (*cooler weather*) and **August** (*warmer weather*). Slice the data and plot the daily seasonal components individually:

```
fig, ax = plt.subplots(2, 1, figsize=(16, 5), sharey=True)
(results.seasonal['seasonal_24']
 .loc['2016-01-04']
 .plot(ax=ax[0],
       title='Daily (24 hour) Seasonality in January'))
(results.seasonal['seasonal_24']
 .loc['2016-08-01']
 .plot(ax=ax[1],
       title='Daily (24 hour) Seasonality in August'))
fig.tight_layout()
plt.show()
```

The preceding code will produce two subplots: 24 hours on January 4 (Monday) and 24 hours on August 1 (Monday):

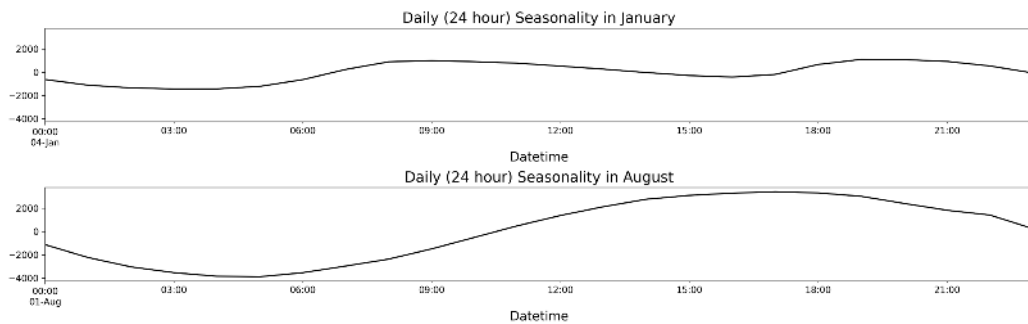


Figure 14.3: The daily (24-hour) seasonal pattern changes with the annual season

In Figure 14.3, we notice a 24-hour pattern: peaks during the day that later decline as they get closer to the evening year-round, but its specific shape is not static as it changes based on the annual season. For example, the magnitude of the daily fluctuation is much greater in August, and the high consumption persists well into the late evening, probably attributed to extended use of air conditioning.

7. Let's perform another slice for the weekly seasonal pattern:

```
fig, ax = plt.subplots(2, 1, figsize=(16, 5), sharey=True)
(results.seasonal['seasonal_168']
 .loc['2016-01-04':'2016-01-10']
 .plot(ax=ax[0],
       title='Weekly Seasonality in January'))
(results.seasonal['seasonal_168']
 .loc['2016-08-01':'2016-08-7']
 .plot(ax=ax[1],
       title='Weekly Seasonality in August'))
fig.tight_layout()
plt.show()
```

The preceding code will show two subplots: 7 days in January (Monday-Sunday) and 7 days in August (Monday-Sunday):

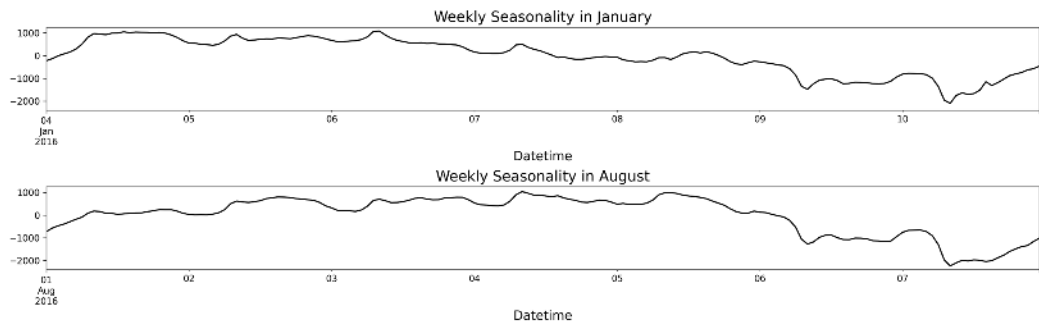


Figure 14.4: The weekly (168-hour) pattern shows a clear drop in energy use over the week-ends

In Figure 14.4, there is a noticeable pattern of highs during weekdays that slowly decays during the weekend. The pattern is similar in January and August.

How it works...

MSTL is an intuitive and powerful algorithm for decomposing time series data with multiple seasonal patterns. It is an extension of the classic STL designed to extract multiple seasonal patterns from a time series dataset. At its core, MSTL estimates the trend and seasonal components in an iterative process:

- It starts with the shortest period you provided (in our example, the 24-hour daily cycle). It runs STL to estimate this pattern, and then subtracts this daily component from the original series.
- Next, it takes the leftover series and runs STL again to estimate the next seasonal period (the 168-hour weekly cycle). It subtracts that out, and so on for all specified periods. This iterative process prevents the different seasonalities from interfering with each other.

- Once all seasonal components are removed, a final STL pass is done on the remaining data to isolate the overall trend. Trend estimation is done via **Locally Weighted Scatterplot Smoothing (LOESS)**, which uses locally estimated regression to capture the trend line.

The `iterate` parameter, which defaults to `iterate=2`, controls how many times the entire multi-stage process is repeated to refine its estimates.

Note that MSTL requires at least one seasonal period; if none are specified, it focuses solely on trend estimation. However, in the `statsmodels` implementation, if no period is provided and the input is a pandas Series or DataFrame with a defined frequency, MSTL will attempt to infer a single seasonal period from the frequency of the time series index. If it cannot infer a period, it will raise an error and not proceed with only trend estimation, and you will get the following error:

```
ValueError: Unable to determine period from endog
```

There's more...

MSTL is mainly used for time series decomposition as it gives you back individual components such as seasonal components (e.g., daily, weekly, yearly), trend, and remainder (residuals). You can use these components to build a forecasting model – for example, you can train a model to forecast the trend using ARIMA, ETS, or Theta. If the seasonal components have a regular pattern, then you can repeat the last observed full cycle (roll forward) for each seasonal component and combine them with the trend. If the seasonal components are evolving (changing), you can model each seasonal component separately and then combine the forecasts.

$$\text{Forecast}(t) = \text{Trend}_{\text{forecast}(t)} + \text{Seasonal}_{\text{daily}(t)} + \text{Seasonal}_{\text{weekly}(t)} + \text{Seasonal}_{\text{yearly}(t)}$$

Luckily for us, Nixtla's `StatsForecast` provides an MSTL model that can be used for forecasting. You were introduced to `StatsForecast` in *Chapter 10*. `StatsForecast`'s MSTL is used for decomposition in a similar manner as the `statsmodels`' MSTL, but it adds to it a `trend_forecaster` that can forecast the trend using `AutoARIMA`, `AutoTheta`, or any other suitable model.

Start by importing the necessary libraries:

```
from statsforecast import StatsForecast
from statsforecast.models import MSTL, AutoARIMA
```

Many of Nixtla's libraries, such as `StatsForecast` and `NeuralForecast`, require your time series to have at least three columns:

- `ds`: Column for the date
- `y`: Column for the variable we want to predict
- `unique_id`: To identify the time series since we can train multiple time series sets, and this helps separate each with a unique identifier

You will use the same DataFrame prepared from the prior steps, but ensure it is compatible with `StatsForecast`:

```
df_sf = df.copy()
df_sf.reset_index(inplace=True)
df_sf.rename(columns={'Datetime': 'ds'}, inplace=True)
df_sf['unique_id'] = 'AEP_hourly' # Adding a unique ID for StatsForecast
```

Let's split our data into train and test splits. We will hold out one week, since we want to test if we can forecast one week into the future ($24 \times 7 =$ a 168-step forecast horizon).

```
train_sf = df_sf.iloc[:-week]
test_sf = df_sf.iloc[-week:]

print(f"Train length: {len(train_sf)}")
print(f"Test length: {len(test_sf)}")
>>
Train length: 121128
Test length: 168
```

Now, let's construct our MSTL model:

```
mstl = MSTL(
    season_length=[day, week, year], # seasonalities of the time series
    trend_forecaster=AutoARIMA() # model used to forecast trend
)
sf = StatsForecast(
    models=[mstl], # model used to fit each time series
    freq='h', # frequency of the data
)
```

Here, we used the day, week, and year seasonal periods defined in the previous statsmodels' MSTL.

Now you can fit the model:

```
sf = sf.fit(df=train_sf)
```



Since we are not using the jump technique to speed up the training like in statsmodels' MSTL, this process will take a much longer time. Expect to wait 10-15 minutes. Generally, MSTL's computational time scales with the number of seasonal components and data size.

Once training is complete. You can forecast and compare the results against the test set:

```
prediction_sf = sf.predict(h=week)
prediction_sf.shape
>>
(168, 3)
```

Let's compare our results. We will use **Root Mean Square Error (RMSE)** and **Mean Absolute Error (MAE)**:

```
mae_sf = mae(test_sf['y'], prediction_sf['MSTL'])
rmse_sf = rmse(test_sf['y'], prediction_sf['MSTL'])
print(f'StatsForecast MSTL MAE: {mae_sf}')
print(f'StatsForecast MSTL RMSE: {rmse_sf}')
>>
StatsForecast MSTL MAE: 595.5971703053872
StatsForecast MSTL RMSE: 765.4769738233874
```

Use the following code to plot the data to compare the forecast against the actual:

```
test_sf.set_index('ds')['y'].plot(label='Actual', figsize=(16, 5), style='--')
prediction_sf.set_index('ds')['MSTL'].plot(label='Forecast')
plt.legend()
plt.title('StatsForecast MSTL Forecast')
plt.show()
```

Figure 14.5 shows the forecast (solid line) against the actual (dashed line).

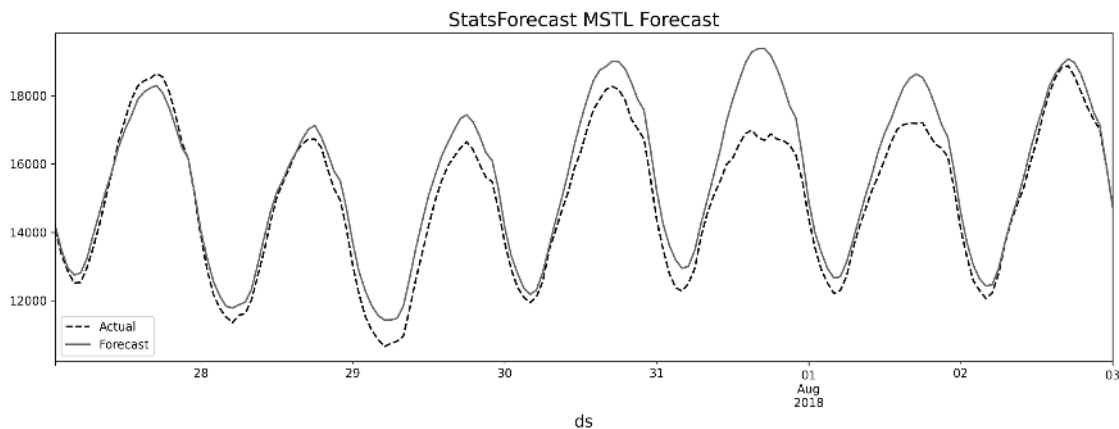


Figure 14.5: One week (168 forecast steps) using StatsForecast MSTL & AutoARIMA

You can explore other forecasting algorithms to forecast the trend in MSTL, such as AutoTheta or AutoETS.

See also

- To learn more about statsmodels' MSTL implementation, visit the official documentation here: <https://www.statsmodels.org/devel/generated/statsmodels.tsa.seasonal.MSTL.html>
- To learn more about StatsForecast's MSTL implementation, visit the official documentation here: <https://nixtlaverse.nixtla.io/statsforecast/src/core/models.html#mstl>

Forecasting with multiple seasonal patterns using the Unobserved Components Model (UCM)

In the previous recipe, you were introduced to MSTL, a technique for decomposing time series with multiple seasonal patterns. Similarly, the **Unobserved Components Model (UCM)** also decomposes a time series, including those with multiple seasonalities, but with an important distinction: the UCM is not only a decomposition method but also a forecasting model.

UCMs, also known as **structural time series models**, were first proposed by *Andrew C. Harvey* in his book *Forecasting, Structural Time Series Models and the Kalman Filter* (1989) as an alternative to the ARIMA model. Unlike an ARIMA model, which requires the time series to be stationary and is often achieved through differencing, UCMs do not make such strict assumptions about stationarity or the data's distribution. While ARIMA models rely on the differencing order (**d**) to enforce stationarity, there are many real-world cases where differencing cannot achieve stationarity or handle complexities such as evolving seasonal patterns or **irregularities** such as changing trend dynamics or smoothly changing patterns (not abrupt structural breaks).

This is where the UCM stands out; it offers a flexible and interpretable approach to modeling complex time series data. UCM can also incorporate external (exogenous) variables, allowing you to model their effect on your forecast.

At its core, the UCM assumes the observed time series is composed of several unobserved components (latent states), such as **level**, **trend**, and **seasonality**, and **cyclical** and **irregular** effects. Because these components evolve over time and can be probabilistically estimated from the observed data, UCMs are expressed as SSMs. They are typically estimated using techniques such as the **Kalman filter**, a powerful recursive algorithm that optimally estimates these unobserved states from the noisy observations (see the *Understanding state-space models* section for more details).

In `statsmodels`, the `UnobservedComponents` class decomposes the time series into components including the following:

- **Trend (μ_t)**: The underlying direction of your data
- **Seasonal patterns (Y_t)**: Supporting multiple seasonalities
- **Cyclical components (c_t)**: Longer-term fluctuations modeled as stochastic cycles
- **An irregular or error term (ϵ_t)**: The unpredictable noise

A general form of the model is:

$$y_t = \mu_t + Y_t + c_t + \epsilon_t$$

Where:

- y_t is the observed variable at time t ,
- μ_t represents the trend component (in some literature, it is sometimes called level)
- Y_t the seasonal component(s)
- c_t cyclical component
- ϵ_t the irregular disturbance (leftover noise)

Each component (μ_t, γ_t, c_t) in the state-space form follows its own evolution equation that defines how it changes over time, making UCM incredibly flexible compared to fixed decomposition methods.

The core idea behind UCM is that each component is treated as an unobserved state that is evolving according to a stochastic process (`statsmodels` supports both stochastic and deterministic specifications for each component). This probabilistic formulation approach allows the model to not only fit the observed data but also provide insight into the underlying dynamics. The forecasts are generated based on the predicted evolution of these states.

Further, UCMs are highly extensible; for example, you can add **autoregressive (AR)** components or include regression effects from external predictors (`exog`), enhancing the ability to capture various data patterns. When it comes to seasonality, `statsmodels` offers flexibility by supporting both standard **time-domain** seasonality (`seasonal`) and **frequency-domain** seasonalities (`freq_seasonal`), which are great for handling multiple and complex seasonal patterns.

By combining decomposition and forecasting in a unified modeling framework, UCMs offer a powerful alternative to classical time series models, especially when dealing with non-stationary series, multiple seasonalities, and complex real-world behaviors.

How to do it...

In this recipe, you will use the UCM implementation in `statsmodels`. You will use the same energy dataset loaded in the *Technical requirements* section. In the *Decomposing time series with multiple seasonal patterns using MSTL* recipe, you were able to decompose the seasonal pattern into daily, weekly, and yearly components. You will use this knowledge in this recipe:

1. Start by importing the `UnobservedComponents` class:

```
from statsmodels.tsa.statespace.structural import UnobservedComponents
```

2. You will use the UCM model to forecast one week ahead (since our data is hourly, this is equivalent to a 168-step forecast horizon). You will hold out one week to validate the model, and you will split the data into train and test sets accordingly. We will use the week variable defined in the previous recipe (as `week = 24x7`):

```
day = 24
week = day*7
year = day*365

train = df.iloc[:-week]
test = df.iloc[-week:]
```

You can plot the data to see the train and test splits:

```
ax = train.iloc[-year:].plot(style='k-', alpha=0.45)
test.plot(ax=ax, style='k')
plt.legend(['train', 'test'])
plt.show()
```

3. `UnobservedComponents` takes several parameters. We will leverage a few of these parameters to showcase UCM's capability.

For `level`, you will set it to `'local linear trend'` or `'l1trend'`. This indicates that both the **level** and **trend** (slope) can change over time and follow a stochastic process. This is equivalent to setting `stochastic_level=True` and `stochastic_trend=True`. This allows for a more flexible trend model where both the level and trend are allowed to evolve over time (stochastic).

For **seasonality**, we have the seasonal (time-domain) and `freq_seasonal` (frequency-domain) parameters. You will set up `freq_seasonal`, which takes a dictionary for each seasonal component in the frequency-domain – for example, *daily*, *weekly*, and *annually*. You will pass a list of key-value pairs for each period. Optionally, within each dictionary, you can also pass a key-value pair for harmonics. If no harmonic values are passed, then the default would be `np.floor(period/2)`.

The `irregular` parameter specifies whether the irregular component should be included. You will set this to `True`.

The `stochastic_freq_seasonal` parameter allows each seasonal component to drift or evolve over time. Setting each component to `True` allows each seasonal pattern to be stochastic, while `False` will assume they are fixed seasonal patterns.

```
params = {
    'level': 'local linear trend', # stochastic level & trend
    'irregular': True,
    'freq_seasonal': [{ 'period': day, 'harmonics': 4},
                      { 'period': week, 'harmonics': 6},
                      { 'period': year, 'harmonics': 8}],
    'stochastic_freq_seasonal': [True, True, False] # annual kept deterministic
}
model = UnobservedComponents(train, **params)
```

The model is configured where daily and weekly patterns might evolve (e.g., changes in work habits, weekend activities), so they are set to be **stochastic** (`True`). While the annual cycle is assumed to be more stable (e.g., hot summers, cold winters) so it is set to **deterministic** (`False`).

Note, even with our optimizations, fitting the model on this large dataset can take several minutes (e.g., up to 10 minutes to complete).

4. You can now fit the model on the training data using the `fit` method:

```
results = model.fit()
```

By default, the `fit` method uses the `lbfgs` solver. Other optimization methods, such as `'bfgs'`, can be specified via the `method` argument.

5. Once training is complete, you can view the model's summary using the `summary` method:

```
results.summary()
```

- The results object is of type `UnobservedComponentsResultsWrapper`, which gives access to several properties and methods, such as `plot_components` and `plot_diagnostics`, to name a couple.

Use `plot_diagnostics` to diagnose the model's overall performance against the residuals:

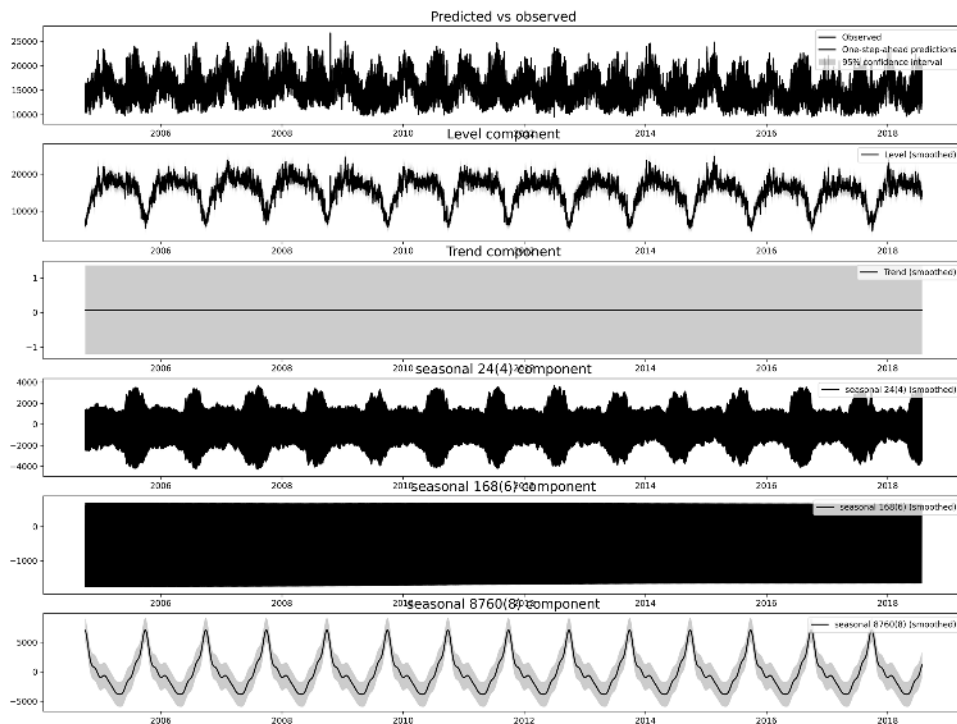
```
fig = results.plot_diagnostics(figsize=(14, 6))
fig.tight_layout()
```

The preceding code should produce 2 x 2 subplots.

- You can visualize the main components of the model using the `plot_components` method, which will produce six subplots using the smoothed results: predicted versus observed, the level component, the trend component, seasonal (24), seasonal (168), and seasonal (8760).

```
results.plot_components(figsize=(20, 15),
                        freq_seasonal=True,
                        which='smoothed')
fig.tight_layout()
```

By default, the value for the `which` parameter is `smoothed`. This should produce the following six subplots:



Note: The first 38 observations are not shown, due to approximate diffuse initialization.

Figure 14.6: Plot the UCM components, including level, trend, and frequency seasonal components (24, 168, 8760)

You can access the data behind each component from the results object. For example, you can access `freq_seasonal`, which should contain a list of three dictionaries, one for each seasonal period: daily, weekly, and yearly, based on what you provided in the key-value pair when you initialized the model (see *Step 3*). Each dictionary contains six outputs or keys, as shown in the following:

```
len(results.freq_seasonal)
>> 3

results.freq_seasonal[1].keys()
>>
dict_keys(['filtered', 'filtered_cov', 'smoothed', 'smoothed_cov',
'offset', 'pretty_name'])
```

You will plot using the smoothed results for each period, starting with daily.

8. Plot the daily seasonal component. For easier interpretation, slice the data to show 24 hours (1 day):

```
daily = pd.DataFrame(
    results.freq_seasonal[0]['smoothed'],
    index=train.index, columns=['y']
).iloc[:day]
daily['hour_of_day'] = daily.index.strftime(date_format = '%I:%M:%S %p')
daily.plot(y='y', x='hour_of_day', title='Daily Seasonality')
```

The preceding code should show a plot representing 24 hours (zoomed in) to focus on the daily pattern. The x-axis ticker labels are formatted to display the time (and AM/PM):

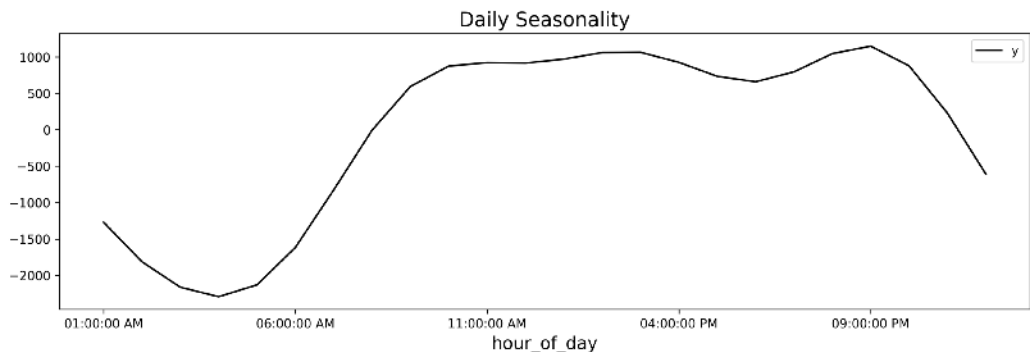


Figure 14.7: Plotting the daily seasonal component

Note that the spike in energy consumption starts around 11:00 AM, peaking at around 9:00 PM, and then declines during sleeping time until the following day. We had a similar observation in the previous recipe using MSTL.

9. Plot the weekly component. Similarly, slice the data (zoom in) to show only one week – ideally, Sunday to Sunday – as shown in the following:

```
weekly = pd.DataFrame(
    results.freq_seasonal[1]['smoothed'],
    index=train.index, columns=['y']
).loc['2004-10-03': '2004-10-10']
weekly['day_name'] = weekly.index.strftime(date_format = '%A')
weekly.plot(y='y', x='day_name', title='Weekly Seasonality')
```

The preceding code should produce a plot showing a whole week to observe the weekly seasonal component. The x-axis ticker labels will display the day's name to make it easier to compare weekends and weekdays:

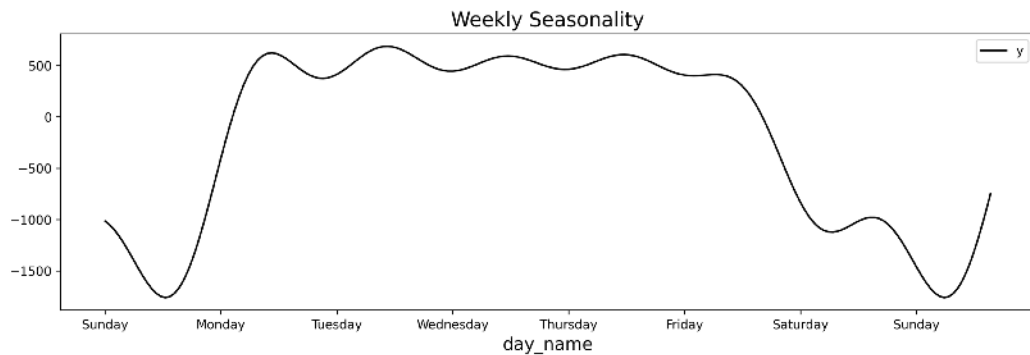


Figure 14.8: Plotting the weekly seasonal component

Notice the spikes during the weekdays (Monday to Friday), which later dip during the weekend. We had a similar observation in the previous recipe using MSTL.

10. Plot the **annual** seasonal component. Similarly, slice the data to show a full year (January to December):

```
annual = pd.DataFrame(
    results.freq_seasonal[2]['smoothed'],
    index=train.index, columns=['y']
).loc['2005']
annual['month'] = annual.index.strftime(date_format = '%B')
ax = annual.plot(y='y', x='month', title='Annual Seasonality')
```

The preceding code should produce a plot of 1 full year from the yearly seasonal component. The x-axis ticker labels will display the month name:

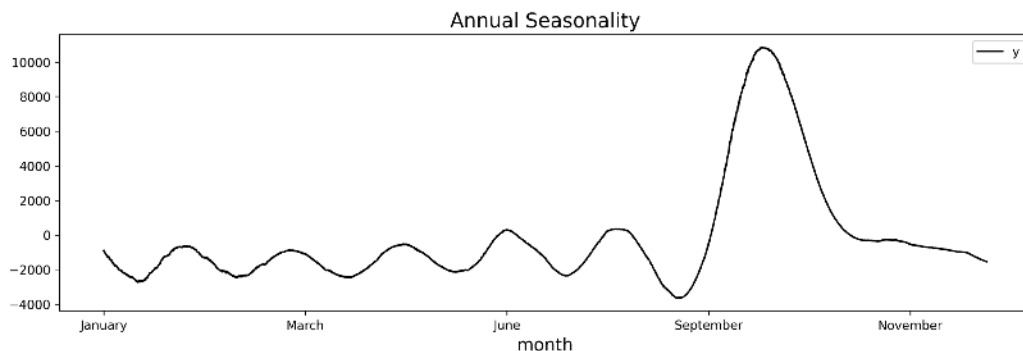


Figure 14.9: Plotting the yearly seasonal component

Notice the spike in January and another spike from July to August.

11. Finally, use the model to make a prediction and compare it against the out-of-sample set (test data):

```
prediction = results.predict(start=test.index.min(),
                             end=test.index.max())
test.plot(style='k--', alpha=0.5)
prediction.plot(style='k')
plt.legend(['Actual', 'Predicted'])
plt.title('Out-of-Sample Forecast: Actual vs Predicted')
```

The preceding code should produce a plot that compares the test (actual) values against the predicted:

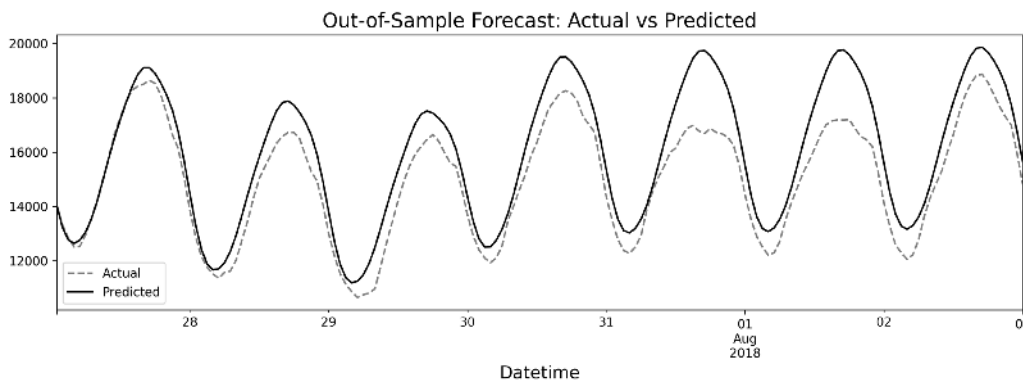


Figure 14.10: Actual versus predicted using the UCM

12. Calculate the RMSE and MAE using the functions imported from statsmodels earlier, in the *Technical requirements* section. You will use the scores to compare with other methods:

```
mae_ucm = mae(test['y'], prediction)
rmse_ucm = rmse(test['y'], prediction)
print(f'UCM MAE: {mae_ucm}')
print(f'UCM RMSE: {rmse_ucm}')
>>
UCM MAE: 969.1994284493461
UCM RMSE: 1146.9518890074817
```

How it works...

The UCM is built as an additive combination of trend, seasonal, cyclical, and irregular components.

$$y_t = \mu_t + Y_t + c_t + \epsilon_t$$

The UCM is a very flexible algorithm that allows for the inclusion of additional models. For example, in the recipe, you used 'level': 'local linear trend', which specifies the use of a **stochastic level and trend** model. The trend is then defined dynamically as:

$$\mu_t = \mu_{t-1} + \beta_{t-1} + \eta_t$$

$$\beta_t = \beta_{t-1} + \zeta_t$$

Where:

- μ_t is the evolving value of the series (trend/level)
- β_t is the slope (trend rate)
- η_t, ζ_t are the random disturbances that allow the level and trend to evolve over time

By specifying stochastic components, we allow our model to adapt to changing energy consumption habits, holiday effects, and other slow shifts in energy use.

You can specify other models to be used either by passing the full name, 'level': 'deterministic trend', or the abbreviated name, 'dtrend'. The statsmodels library provides an extensive list of models that you can specify – for example, 'local linear deterministic trend' (or 'l1dltrend'), 'smooth trend' (or 'strend').

When you provide key-value pairs to the 'freq_seasonal' parameter, each seasonal component is modeled separately. Setting a seasonal component as stochastic allows its amplitude and phase to drift smoothly over time, capturing evolving seasonal patterns in your data. In our implementation, we set both daily and weekly as **stochastic**, but yearly as **deterministic**.

We added **harmonics** to overwrite the default calculation as a way to improve model training duration (otherwise, it would take too much time). Harmonics are a set of sine and cosine waves that repeat at multiple fundamental frequencies (e.g., 24 hours for a daily pattern is the fundamental frequency). Here are some examples:

- **1st harmonic:** This is a simple sine wave that completes one full cycle every 24 hours. This will capture the main up-and-down rhythm of the day.
- **2nd harmonic:** A sine wave that completes two full cycles every 24 hours (i.e., repeats every 12 hours). This can capture morning and evening peaks.
- **3rd Harmonic:** A sine wave that completes three full cycles every 24 hours (i.e., repeats every 8 hours).

By adding these simple waves together, you can approximate complex, real-world patterns.

For a complete list of all the available models and the associated equations, you can visit the documentation here: <https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.structural.UnobservedComponents.html#r0058a7c6fc36-1>.

Keep in mind that when you provide key-value pairs to the `freq_seasonal` parameter, each seasonal component is modeled separately.

There's more...

One of the key advantages of UCMs is their flexibility. The model can easily be extended to capture even more dynamics. For example, we can add **AR** components and incorporate **exogenous variables**.

In the main recipe, we used `plot_diagnostics()` to inspect our model's residuals. Ideally, we would expect the residual to be random noise with no visible pattern (no autocorrelation). The existence of autocorrelation in the residual suggests there are short-term dynamics that our trend and seasonal components haven't fully captured. We can model this leftover structure directly by adding an AR component to the model.

This is done with the autoregressive parameter. You simply specify the order of the AR process you want to include. For example, here's how to add an AR(1) term:

```
# Add an AR(1) component to our previous model
params['autoregressive'] = 1
```

We can also incorporate exogenous variables directly in the model using the `exog` parameter. For example, if we determined that energy consumption is different on weekends versus weekdays, we can include this information as an exogenous variable. We can create a simple variable where 1 indicates a weekend and 0 indicates a weekday:

```
df_exog = df.copy()
df_exog['is_weekend'] = df_exog.index.weekday.isin([5, 6]).astype(int)

# Re-split the data to include the new column
train_exog = df_exog.iloc[:-week]
test_exog = df_exog.iloc[-week:]
```


Let's train a new model with these updates (AR components and an exogenous variable):

```
model_exog_ar = UnobservedComponents(train_exog['y'],
                                     exog=train_exog[['is_weekend']],
                                     **params)

results_exog_ar = model_exog_ar.fit()
```

Note that the model training will take a bit more time with these additions (estimate between 15 and 20 minutes). Once the training is complete, you can use the `plot_components()` method as you did earlier. You will notice the addition of the AR component in the plot.

Use the model to create the forecast. You will need to pass the exogenous variable when making the prediction.

```
prediction_exog_ar = results_exog_ar.predict(start=test_exog.index.min(),
                                             end=test_exog.index.max(),
                                             exog=test_exog[['is_weekend']])
```

Calculate RMSE and MAE to see if the AR component and weekend effect improve the model:

```
mae_ucm = mae(test_exog['y'], prediction_exog_ar)
rmse_ucm = rmse(test_exog['y'], prediction_exog_ar)
print(f'UCM w/AR and Exog MAE: {mae_ucm}')
print(f'UCM w/AR and Exog RMSE: {rmse_ucm}')
>>
UCM w/AR and Exog MAE: 672.3820040398705
UCM w/AR and Exog RMSE: 856.6806116754617
```

Compared to the first model, you can see there is a significant improvement in performance.

See also

- To learn more about the UCM implementation in statsmodels, you can visit the official documentation here: <https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.structural.UnobservedComponents.html>

Forecasting time series with multiple seasonal patterns using AutoTBATS

Like the UCM, TBATS is an SSM that decomposes a time series into multiple components. Both models represent seasonality using a series of trigonometric functions, known as **harmonics** or a **Fourier series** (sine and cosine terms), making them highly effective for handling multiple or complex seasonal patterns. Note that UCMs can support seasonality representation in both the time domain and frequency domain.

TBATS extends the framework with several additional features, including the **Box-Cox transformation** for variance stabilization, and **ARMA errors** for residual autocorrelation modeling (while UCM can include AR components, ARMA modeling is a built-in part of the TBATS specification).

The name **TBATS** stands for the following:

- Trigonometric seasonal components
- Box-Cox transformation for stabilizing variance
- ARMA (**AutoRegressive Moving Average**) errors
- Trend components (including damped trends)
- Seasonal components (including multiple and complex seasonalities)

In this recipe, you will use the **AutoTBATS** implementation from `StatsForecast`, which you were first introduced to in *Chapter 10*.

Unlike other SSMs, such as UCM, TBATS does not support exogenous variables (covariates). This means TBATS is best suited for univariate forecasting.

Getting ready

You can install `StatsForecast` using `pip`:

```
pip install statsforecast
```

You can also install `StatsForecast` using `conda`:

```
conda install -c conda-forge statsforecast
```

How to do it...

You will use the same energy dataset (`df`) that was loaded in the *Technical requirements* section.

1. Start by importing the necessary libraries for this recipe.

```
from statsforecast import StatsForecast
from statsforecast.models import AutoTBATS
```

2. Prepare the `DataFrame` to be compatible with `StatsForecast`. The library expects specific column names:

- `ds`: The timestamp column
- `y`: The target variable to be forecast
- `unique_id`: An identifier for each time series. Since we only have one, we'll create a single ID.

```
df_sf = df.copy()
df_sf.reset_index(inplace=True)
df_sf.rename(columns={'Datetime': 'ds'}, inplace=True)
df_sf['unique_id'] = 'AEP_hourly' # Adding a unique ID for StatsForecast
```

3. Split the data into train and test sets. Use the same definitions as in the previous recipes for daily and weekly components:

```
day = 24
week = day*7

train_sf = df_sf.iloc[:-week]
test_sf = df_sf.iloc[-week:]
```

4. TBATS is powerful for handling complex seasonalities. In our hourly energy dataset, we have *daily*, *weekly*, and *yearly* patterns. A naive approach would include all three seasonal periods: `season_length=[day, week, year]`. However, the computational cost of automatically finding the best trigonometric representation for a very long season like 8760 can be high.

A more practical approach is to focus on the seasonalities that are relevant to the short-term forecast horizon (one week). Thus, by modeling only the daily (24 hours) and weekly (168 hours) patterns, the model can be trained in a reasonable time (~11 minutes) while still producing strong results. The yearly pattern is partly absorbed by the trend component over this short horizon (short-term forecasts). If we are forecasting for longer-term horizons, then explicit yearly seasonality may be necessary.

```
# model multiple seasons: daily, weekly
tbats = AutoTBATS(
    season_length=[day, week],
    use_trend=None, # None means to try both (with or without)
    use_boxcox=None, # None means to try both (with or without)
    use_arma_errors=None, # None means to try both (with or without)
    use_damped_trend=None, # None means to try both (with or without)
)
```

5. Instantiate `StatsForecast` and specify hourly frequency:

```
# hourly frequency
sf = StatsForecast(
    models=[tbats],
    freq='H',
    n_jobs=-1
)
```

6. Fit the model on the training set. `AutoTBATS` will automatically search for the best parameters.

```
# train our model
sf.fit(train_sf)
```

7. Use the `predict` method to create the forecast and calculate both MAE and RMSE scores against the test set:

```
prediction_sf = sf.predict(h=week)
mae_sf = mae(test_sf['y'], prediction_sf['AutoTBATS'])
rmse_sf = rmse(test_sf['y'], prediction_sf['AutoTBATS'])
print(f'AutoTBATS MAE: {mae_sf}')
print(f'AutoTBATS RMSE: {rmse_sf}')
>>
AutoTBATS MAE: 868.4687534921502
AutoTBATS RMSE: 1129.4514312286358
```

AutoTBATS is a powerful and automatic tool designed to handle complex seasonality within a family of statistical models like MSTL and UCM. Remember, there is no universally best model for all forecasting problems, as demonstrated by the varying scores and performance of the models explored so far.

How it works...

TBATS was developed by *De Livera, Hyndman, & Snyder* in the paper *Forecasting time series with complex seasonal patterns using exponential smoothing*.

AutoTBATS is not a single model but an algorithm that finds the optimal TBATS model by making several key decisions automatically:

- Whether Box-Cox transformation improved the model
- Whether the trend component is needed, and whether it should be damped or not
- For each seasonal component (e.g., day and week), AutoTBATS automatically chooses the optimal number of Fourier terms (pairs of sine and cosine waves) that best capture the shape of the seasonality
- For the residuals, it uses ARMA errors to model the remainders if there are any leftover auto-correlations to capture any remaining information, such as short-term dynamics

The selection for the best combinations of parameters is done using the AIC score by default. In our model, we specified `use_trend`, `use_boxcox`, `use_damped_trend`, and `use_arma_errors` as `None` to enable automatic selection. The algorithm will try both with and without to determine the optimal solution. If we set the parameter to `True`, it will always use it.

There's more...

Using the `StatsForecast` class has several advantages, including its ability to use multiple CPU cores for fast processing by setting `n_jobs=-1` (which uses all available cores), fit multiple models in parallel, and handle multiple time series seamlessly.

If you are interested in learning about the model's configuration, such as whether it used a Box-Cox transformation, or what ARMA order was chosen, then you can do so through the **fitted** `StatsForecast` object:

```
fitted_model = sf.fitted_[0][0]
```

You can access the model configuration:

```
fitted_model.model_['description']  
  
>>  
{'use_boxcox': True,  
  'use_trend': True,  
  'use_damped_trend': True,  
  'use_arma_errors': False}
```

You can also access the residuals and fitted values:

```
fitted = fitted_model.model_['fitted'][0]  
residuals = fitted_model.model_['errors'][0]
```

This can be useful if you want to plot the residuals:

```
import matplotlib.pyplot as plt  
plt.figure(figsize=(12,4))  
plt.plot(residuals)  
plt.title('AutoTBATS Residuals')  
plt.xlabel('Time')  
plt.ylabel('Residual')  
plt.show()
```

This should produce the following plot:

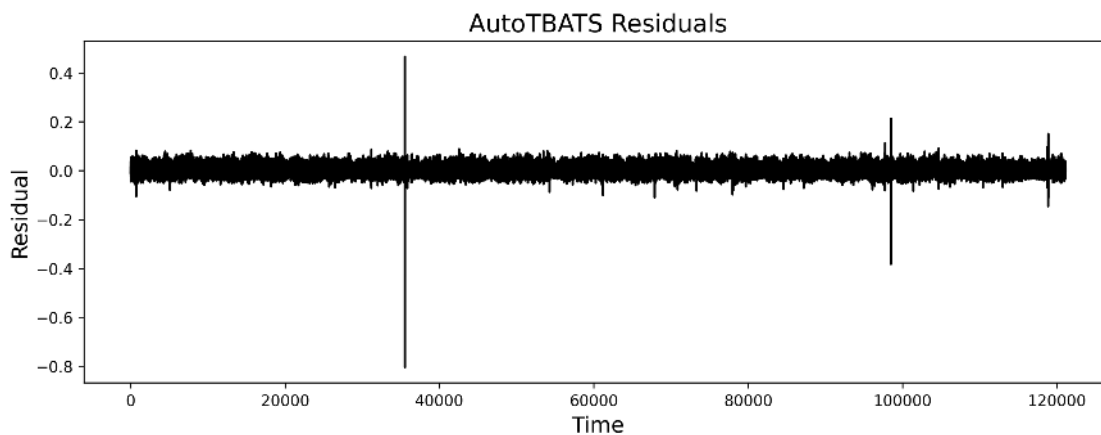


Figure 14.11: Plot of AutoTBATS residuals

You can also obtain the *p* and *q* orders from the ARMA model:

```
p = fitted_model.model_['p']
q = fitted_model.model_['q']
print(f"ARMA's p={p} and q={q}")
>>
ARMA's p=0 and q=0
```

Notice that the orders *p* and *q* are both zero. This directly corresponds to the model's configuration we saw earlier (`'use_arma_errors': False`), which tells us that AutoTBATS determined that modeling the residuals with an ARMA was not beneficial.

You can access the *lambda* used in the Box-Cox transformation:

```
fitted_model.model_['BoxCox_lambda']
>>
0.00046153629516299167
```

A *lambda* value that is close to zero indicates that the model applied a log transformation to the data. See *Chapter 8*, in the *Applying power transformations* recipe, to learn more about Box-Cox and *lambda*.

See also

- To learn more about AutoTBATS, you can visit the official documentation here: <https://nixtlaverse.nixtla.io/statsforecast/src/core/models.html#autotbats>

Forecasting time series with multiple seasonal patterns using NeuralProphet

NeuralProphet was inspired by the Prophet library and Autoregressive Neural Network (AR-Net) to bring a new implementation, leveraging deep neural networks to provide a more scalable solution.

Prophet was built on top of CmdStan, a Bayesian inference library based on Stan, and is one of the main dependencies when installing Prophet. Conversely, NeuralProphet is based on PyTorch and is used as the deep learning framework internally. NeuralProphet brings neural networks – specifically, autoregressive architecture – via AR-Net. This allows NeuralProphet to scale to larger datasets and can often lead to improved accuracy compared to Prophet, especially on larger datasets. Like Prophet's approach, NeuralProphet performs hyperparameter tuning and fully automates many aspects of time series forecasting.

Both Prophet and NeuralProphet can handle complex time series data with multiple and overlapping seasonal patterns. You learned about Prophet in *Chapter 10*, so in this recipe, we will introduce a different library to help explore more options.

Getting ready

Creating a new Python environment when installing new libraries such as NeuralProphet is always a good idea. If you need a quick refresher on creating a virtual Python environment, check out the *Development environment setup* recipe from *Chapter 0* on GitHub. The chapter covers two methods: using conda and venv.

The following instructions will show how to create a virtual environment using conda:

```
>> conda create -n neuralprophet python=3.12 -y
>> conda activate neuralprophet
>> pip install neuralprophet plotly-resampler ipywidgets statsmodels
```

To make the new neuralprophet environment visible within Jupyter, you can run the following code:

```
python -m ipykernel install --user --name neuralprophet --display-name
"NeuralProphet"
```

How to do it...

NeuralProphet makes it easy to transition from Prophet, as they follow a very similar framework for training the model. We assume you ran the code to load and prepare the dataset in the *Technical requirements* section and have the `df` DataFrame available:

1. Start by importing the library:

```
from neuralprophet import NeuralProphet, set_random_seed
set_random_seed(seed=42) # for reproducibility
```

2. Prepare the dataset to be used with NeuralProphet, since it expects the DataFrame to have two columns: `ds` (datestamp, as a datetime object) and `y` (the target variable to forecast).

```
df_np = df.copy()
df_np.reset_index(inplace=True)
df_np.rename(columns={'Datetime': 'ds'}, inplace=True)
```

3. Split the data into train, validation, and test sets. The validation set will be used during the training process to evaluate at every epoch:

```
week = 24*7
train_np = df_np.iloc[:-week*2]
val_np = df_np.iloc[-week*2:-week]
test_np = df_np.iloc[-week:]
print(f"""
    train_np : {len(train_np)}
    val_np : {len(val_np)}
    test_np: {len(test_np)}
""")
```

```
>>
train_np : 120960
val_np : 168
test_np: 168
```

4. Instantiate the `NeuralProphet` class with default parameters. `NeuralProphet` does automatic selection of hyperparameters, including the detection of seasonal components. Then fit the model on the training set and provide the validation set to use for evaluating performance at each epoch.

```
m = NeuralProphet()
metrics = m.fit(train_np, validation_df=val_np)
```

`NeuralProphet` will automatically set the learning rate, number of epochs, and batch size. You can still set them manually if needed. Similarly, for `yearly_seasonality`, `weekly_seasonality`, and `daily_seasonality`, they are all set to 'auto', but you can manually turn them on or off (True or False). Additionally, you can provide an integer for the number of **Fourier** terms to generate.

When training starts, `NeuralForecast` will print out some of the automated configurations based on the dataset provided, including batch size and number of epochs:

```
INFO - (NP.df_utils._infer_frequency) - Dataframe freq automatically
defined as h
INFO - (NP.config.set_auto_batch_epoch) - Auto-set batch_size to 256
INFO - (NP.config.set_auto_batch_epoch) - Auto-set epochs to 30
```

`NeuralProphet` auto-selects frequency, batch size, and epochs for convenience. However, you can specify parameters manually if required:

```
m = NeuralProphet(batch_size=128, epochs=50, learning_rate=1e-3)
```

5. Use the `predict` method to generate your forecast.
6. Plot the identified components, which may include *trend*, *yearly*, *weekly*, and *daily* seasonalities, using the `plot_components` method:

```
m.plot_components(forecast)
```

This should produce the following four interactive subplots, one for each component.

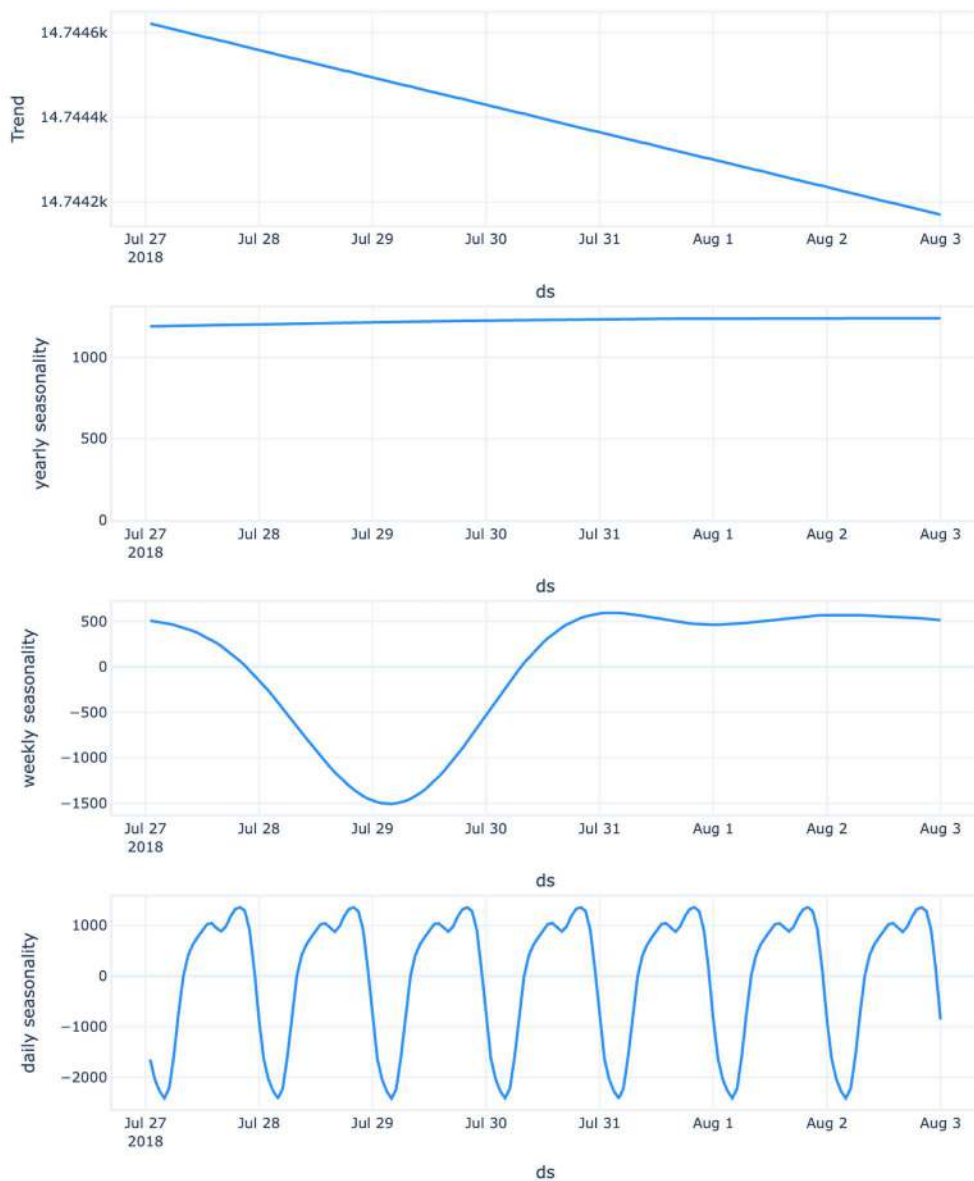


Figure 14.12: Decomposition of the forecast into trend and seasonal patterns

By default, NeuralProphet will plot interactive visualizations using a Plotly backend. If you want static visualization, you can change the setting using `m.set_plotting_backend('plotly-static')`, which will change all plots globally. You can just update for a single plot by using the `m.plot_components(forecast, plotting_backend='plotly-static')` parameter:

```
m.plot_components(forecast, plotting_backend='plotly-static')
```

You can also plot only a subset of the components through the `components` parameter:

```
m.plot_components(forecast, components=['seasonality'])
```

- NeuralProphet provides the `plot_parameters` method, which gives more insight into the individual parameters of the model.

```
m.plot_parameters()
```

This should produce five subplots, including *Trend*, *Trend Rate Change*, *Yearly Seasonality*, *Weekly Seasonality*, and *Daily Seasonality*:

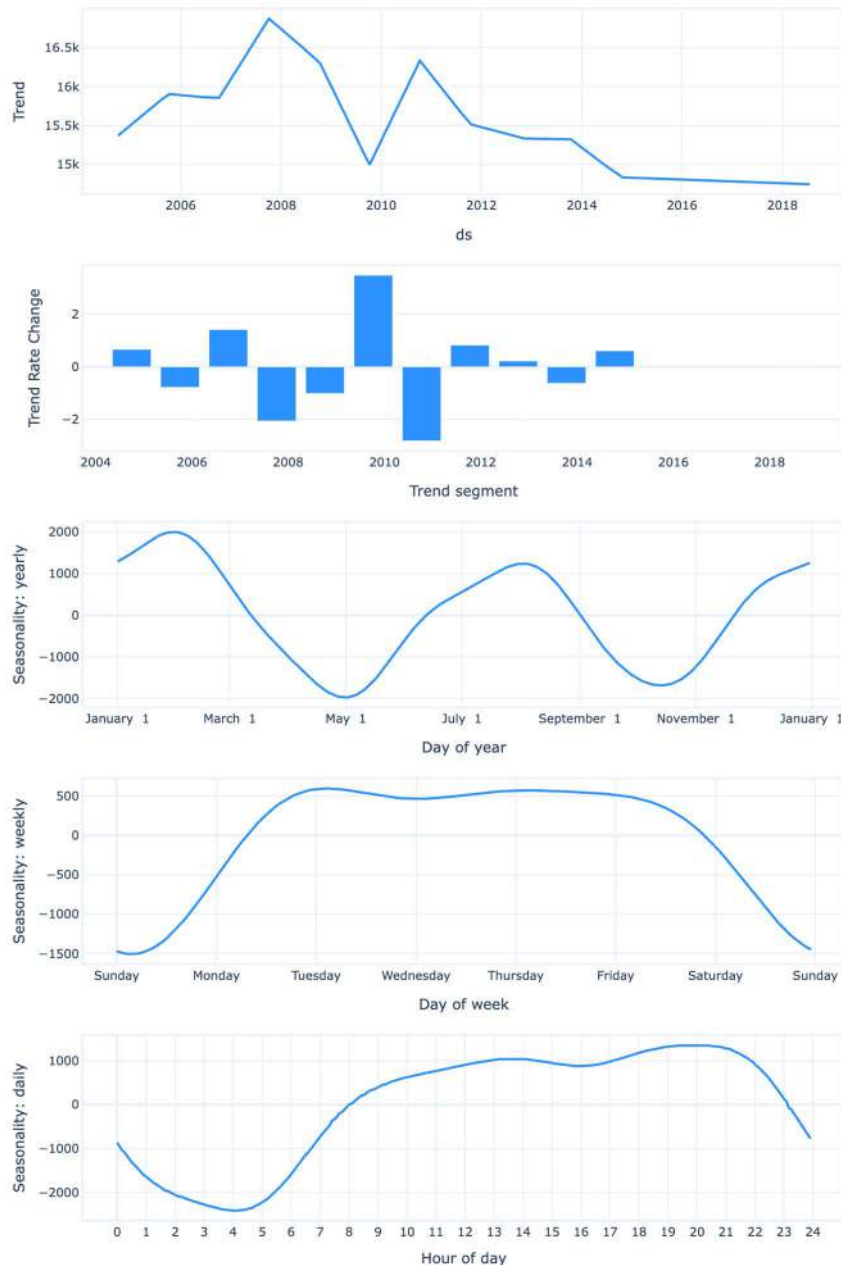


Figure 14.13 – Visualizing the model's learned seasonal and trend parameters

Unlike `plot_components`, which shows the component values for the forecast period, `plot_parameters` visualizes the general patterns the model has learned from the entire training dataset.

You can plot only a subset of the parameters by using the `components` parameter:

```
m.plot_parameters(components=['Trend', 'trend_rate_change'])
```

8. The `metrics` object is a `DataFrame` that captures evaluation scores for each epoch:

```
metrics.columns
>>
Index(['MAE_val', 'RMSE_val', 'Loss_val', 'RegLoss_val', 'epoch', 'train_
loss',
      'reg_loss', 'MAE', 'RMSE', 'Loss', 'RegLoss'],
      dtype='object')
```

You can view the values from the last epoch:

```
print(metrics.tail(1))
>>
   MAE_val  RMSE_val  Loss_val  RegLoss_val  epoch  train_loss \
29  1168.61731  1367.575317  0.028064         0.0      29    0.028404

   reg_loss      MAE      RMSE      Loss  RegLoss
29      0.0  1225.663818  1596.395142  0.028406      0.0
```

9. Compare the out-of-sample prediction values with the test dataset. The forecast `DataFrame` contains several columns, with the prediction stored in `yhat1`. The 1 signifies a one-step ahead forecast, which is the default.

```
forecast.columns
>>
Index(['ds', 'y', 'yhat1', 'trend', 'season_yearly', 'season_weekly',
      'season_daily'],
      dtype='object')
```

Let's grab the `yhat1` column and plot it against the actual values from the test set:

```
prediction_np = forecast.set_index('ds')['yhat1']
test_np.set_index('ds').plot(style='k--', alpha=0.5)
prediction_np.plot(style='k')
plt.legend(['Actual', 'Predicted'])
plt.title('Out-of-Sample Forecast: Actual vs Predicted')
plt.show()
```

This should produce a plot similar to *Figure 14.14*:

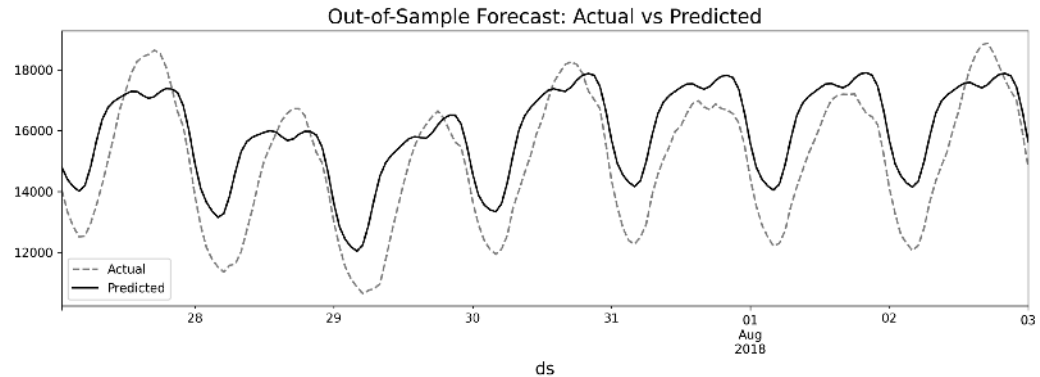


Figure 14.14: Actual versus predicted with NeuralProphet

10. NeuralProphet provides a convenient `test()` method to evaluate the model on holdout data using the following:

```
test_results = m.test(df=test_np)
print(test_results)
>>
```

Test metric		DataLoader 0	
Loss_test		0.030542725697159767	
MAE_val		1236.0711669921875	
RMSE_val		1427.038818359375	
RegLoss_test		0.0	

	MAE_val	RMSE_val	Loss_test	RegLoss_test
0	1236.071167	1427.038818	0.030543	0.0

You may notice the metrics are labeled `_val` (for validation) in the output, but in reality, the scores are calculated based on the `test_np` data provided. This is a quirk in NeuralProphet's output formatting.

```
from statsmodels.tools.eval_measures import rmse, meanabs as mae
mae_np = mae(test_np['y'], prediction_np)
rmse_np = rmse(test_np['y'], prediction_np)
print(f'NeuralProphet MAE: {mae_np}')
print(f'NeuralProphet RMSE: {rmse_np}')
```

```
>>
NeuralProphet MAE: 1236.0711553664435
NeuralProphet RMSE: 1427.0386964275483
```

How it works...

Similar to Prophet, NeuralProphet decomposes a time series into components. At its core, the model can be presented as:

$$y(t) = T(t) + S(t) + E(t) + F(t) + A(t) + L(t) + \epsilon(t)$$

Where $T(t)$ is the trend, $S(t)$ is seasonality, $E(t)$ is events/holidays, $F(t)$ are future regressors, $A(t)$ is AR components, $L(t)$ are the lagged external regressors, and $\epsilon(t)$ is the error term.

Similar to Prophet, NeuralProphet decomposes a time series into its components. NeuralProphet can include different terms such as *trend*, *seasonality*, *autoregression* (the `n_lags` parameter), special events (the `add_events` method), holidays (the `add_country_holidays` method), future regressors (the `add_future_regressor` method), and lagged regressors (the `add_lagged_regressor` method).

The autoregression piece is based on AR-Net, an autoregressive neural network. NeuralProphet allows you to add recurring special events – for example, a Super Bowl game or a birthday. The time series modeling process is similar to that in Prophet, which was made by design. Keep in mind that NeuralProphet was built from the ground up, leveraging PyTorch, and is not a wrapper to Prophet.

There's more...

NeuralProphet offers flexibility and extensibility (and is very modular for anyone who wants to extend its capabilities). The library also provides additional features tailored to working with deep neural networks, such as logging performance, cross-validation, splitting data, benchmarking, and experimentation.

Let's explore some of these features: adding future regressors (future covariates) such as `is_weekend` (similar to our UCM recipe), include U.S. holidays, and plot the train and validation loss:

```
df_np['is_weekend'] = df_np['ds'].dt.weekday.isin([5, 6]).astype(int)
train_np = df_np.iloc[:-week*2]
val_np = df_np.iloc[-week*2:-week]
test_np = df_np.iloc[-week:]

m = NeuralProphet()
m.add_future_regressor("is_weekend")
m.add_country_holidays("US")

metrics = m.fit(train_np, validation_df=val_np)
```

Once training is done, you can inspect the effect of **Additive Future Regressor Weight** and **Additive Event Weights**:

```
m.plot_parameters(components=['events', 'future_regressors'],
                  plotting_backend='plotly-static')
```

This will produce two subplots: the additive future regressor weight and the additive event weight, illustrating the learned importance that the model assigns to each regressor or event.

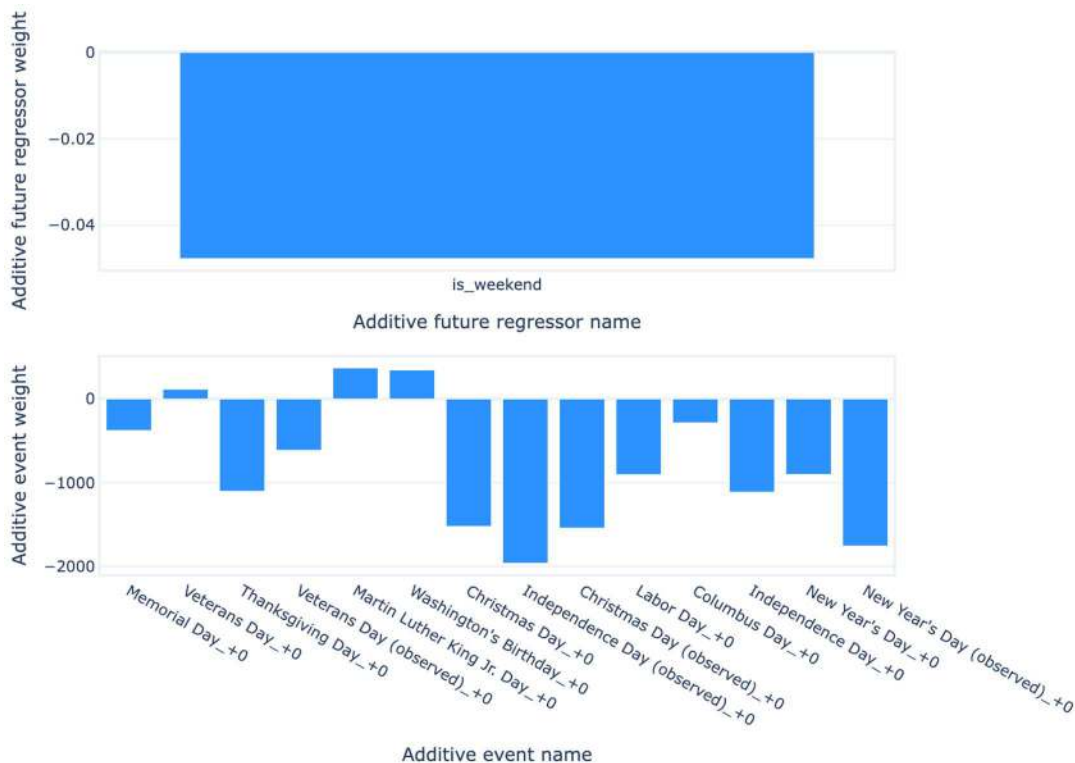


Figure 14.15: Plotting additive future regressor and additive event weights

The weight for `is_weekend` is negative, around -0.045, indicating that when `is_weekend` is active (i.e., it's a weekend), the model predicts a decrease in electricity demand. In the bottom plot, each holiday/event has a bar indicating its impact on the forecast. Most holidays show negative weights, with very substantially negative impacts from Christmas Day, New Year's Day, Independence Day, Thanksgiving Day, and Labor Day. These holidays are associated with significantly reduced electricity demand, reflecting the closure of most businesses, factories, and offices during these holidays. A few federal holidays show small positive weights (+500): Martin Luther King Jr. Birthday, Washington's Birthday, and Veterans Day.

See also

- To learn more about NeuralProphet, you can reference the official documentation here: <https://neuralprophet.com/contents.html>



There's an exciting GitHub-only bonus—*Chapter 15: Probabilistic Time Series Forecasting* and *Chapter 16: Analyzing Time Series in the Frequency Domain*—available here: <https://github.com/PacktPublishing/Time-Series-Analysis-with-Python-Cookbook-Second-Edition>

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:

-  Next-gen Packt Reader
-  DRM-free PDF/ePub downloads

Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.



Note: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to [packtpub.com/unlock](https://www.packtpub.com/unlock).



On the page that opens (similar to *Figure U.1* on desktop), search for this book by name and select the correct edition.

packt Search... Subscription Shopping Cart User Profile

Explore Products Best Sellers New Releases Books Videos Audiobooks Learning Hub Newsletter Hub Free Learning

Discover and unlock your book's exclusive benefits

Bought a Packt book? Your purchase may come with free bonus benefits designed to maximise your learning. Discover and unlock them here

Discover Benefits Sign Up/In Upload Invoice

[Need Help?](#)

1. Discover your book's exclusive benefits

Search by title or ISBN

CONTINUE TO STEP 2

2. Login or sign up for free

3. Upload your invoice and unlock

Figure U.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.



Note: If you are still facing issues, reach out to customer@packt.com.



www.packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Modern Time Series Forecasting with Python

Manu Joseph, Jeffrey Tackes

ISBN: 9781835883181

- Build machine learning models for regression-based time series forecasting
- Apply powerful feature engineering techniques to enhance prediction accuracy
- Tackle common challenges like non-stationarity and seasonality
- Combine multiple forecasts using ensembling and stacking for superior results
- Explore cutting-edge advancements in probabilistic forecasting and handle intermittent or sparse time series
- Evaluate and validate your forecasts using best practices and statistical metrics



Time Series Analysis with Spark

Yoni Ramaswami

ISBN: 9781803232256

- Understand the core concepts and architectures of Apache Spark
- Clean and organize time series data
- Choose the most suitable modeling approach for your use case
- Gain expertise in building and training a variety of time series models
- Explore ways to leverage Apache Spark and Databricks to scale your models
- Deploy time series models in production
- Integrate your time series solutions with big data tools for enhanced analytics
- Leverage GenAI to enhance predictions and uncover patterns

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packt.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Time Series Analysis with Python Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

- acorr_ljungbox function** 326
 - reference link 326
- activation functions** 569, 570
- additive decompositions** 295
- Additive Event Weights** 765
- Additive Future Regressor Weight** 765
- additive regression model** 403
- aeon library** 666
- aggregation pipeline processes** 72
- AIC** 399
- AICc** 399
- Air Passengers dataset** 284, 307
- Akaike Information Criterion** 305
- Amazon Redshift**
 - database, extending to 47, 48
 - writing to, with AWS SDK for pandas 116, 117
 - writing to, with redshift_connector 115, 116
 - writing to, with SQLAlchemy 111-115
- Amazon S3**
 - reference link 101
 - writing to, with boto3 library 99
 - writing to, with pandas 97
- Amazon S3 bucket** 1
- Anderson-Darling test** 316-318
- Apache Arrow** 31
- Apache Foundation** 25
- arch library** 401
- ARIMA models** 365
 - principal components 365
- ARMA errors** 753
- AR order**
 - versus MA order 337
- artificial neural networks (ANNs)** 567, 569
- Augmented Dickey-Fuller (ADF) test** 298-301
- auto_arima**
 - model estimation 399
 - model search process 398
 - optimal parameters 391
 - stationary testing 398
 - univariate time series data,
 - forecasting with 391-399
- AutoARIMA** 473-476, 482, 740
- autocorrelation** 252, 312, 322, 452
 - testing 323-325
- autocorrelation function (ACF)** 658
 - plotting 335-345
- Autocorrelation Function (ACF) test** 430
- AutoEncoder** 662, 721, 722
 - architecture 722
- AutoETS** 362, 473, 474
 - using 362-364
- automated forecasting** 401
- automated univariate forecasting**
 - with StatsForecast library 472-484

autoregressive (AR) 460
Autoregressive Conditional Heteroskedasticity (ARCH) model 460
autoregressive model (AR(p)) 373
Autoregressive Neural Network (AR-Net) 757
AutoTBATS
 for forecasting, time series with multiple seasonal patterns 752-757
 implementation 753
AutoTheta 473, 474, 740
AWS access key ID 19
AWS SDK for pandas
 used, for writing to Amazon Redshift 116, 117
AWS SDK for Python (Boto3) 15
aws wrangler library 116
 reference link 117
Azure Blob Storage 75
 reference link 101
 writing to, with azure-storage-blob library 100
 writing to, with pandas 98, 99

B

backtesting tools 653, 657
backtest() method 655, 656
backward fill (bfill) 205
bagging ensemble 556
BaggingRegressor 220
ball_tree 688
batch size 570
Bayesian inference 403
Bayesian Information Criterion 305
BayesianRidge 220
BIC 399
binary classification problems 488
BlockRNNModel from Dart
 using 607-614

boto3 library
 used, for writing to Amazon S3 99
Boto3 library 15
boxcox function
 reference link 322
Box-Cox transformation 307, 317, 753
 applying 312-317
boxen plot 248
box plots 244, 250
Breusch-Pagan Lagrange Multiplier test 312
Breusch-Pagan Lagrange test 308
Breusch-Pagan test 319
brute 688
BSON format 117
bucket pattern 124
business day rule 240

C

Canova-Hansen (CH) test 391
categorical features encoding 552
causal relationships 446
cell input 573
censoring 188
change point detection (CPD) 234
change points 403, 413
 detecting 415-417
Chebyshev distance 688
chunksize parameter 48, 49
Client API 20, 99
Closing Price Stock Data dataset 284
cloud storage
 data, storing 95, 96
Cluster-Based Local Outlier Factor (CBLOF)
 algorithm 695-699
clustering-based algorithms
 used, for outlier detection 695-701

CmdStan 757
cmdstanpy 403
CO2 dataset 284, 287
CO2 emissions dataset 186
coefficients 336
collective outliers 665
comma-separated values (CSV) 1
computer vision (CV) 565
concept drift 276
Conda 15, 427
Conditional Volatility plot 466
confidence intervals 431, 343, 443
config file 18
constant seasonal variation 288
consumer price index (CPI) 481
contamination parameter 689
contamination percentage 662
context manager
 using 55
contextual outlier detection 662
contextual outliers 665, 666
convolutional operation 637
Coordinated Universal Time (UTC) 152
Copular-Based Outlier Detection (COPOD)
 algorithm 662, 703
copulas 703
copula theory 703
correlation matrix 442
correlation statistics 337
correlograms 337, 372
Cosine distance 687
covariates 606
cross-correlation 452
Cross-Correlation Function (CCF) 438
cross-validated grid search 534, 535

cross-validation 478-480
 performing 417, 418
CSV file
 data, reading from 3-5
 writing to 83-88
cumulative distribution function (CDF) plot 247
cumulative impulse responses 455
custom business days
 working with 177-182
cyclical encoding 495
cyclic encoding 559

D

D'Agostino-Pearson test 308, 316-318
Darts' BlockRNNModel class
 reference link 605
Darts BlockRNNModel for direct multi-output
 using 591-594
Darts library 360, 585
 AutoETS 362-364
 exponential smoothing 361, 362
 installing 360, 361
 StatsForecastAutoARIMA 399, 400
Darts model capabilities
 reference link 606
Darts RNNModel class
 reference link 605
 using 585-589
Darts RNNModel for recursive multi-output
 using 590
DartsXGBModel 557, 559
data
 reading, from CSV and other delimited files 3-8
 reading, from Excel file 9-14
 reading, from GitHub 16, 17
 reading, from HTML 20-25
 reading, from MongoDB 61
 reading, from Parquet files 26, 27

- reading, from public S3 bucket 17, 18
- reading, from relational database 36
- reading, from Snowflake 50
- reading, from URLs 14, 15
- reading, Parquet files 25
- writing, to Excel file format 88-94
- database API (DBAPI) 45**
- databases**
 - advantages 103
- DataFrame 1**
 - using 207
- DataFrame index 191**
- DataFrame.interpolate() method 223**
- DataFrame.to_sql() function**
 - reference link 117
- data quality checks**
 - performing 189-199
- data, reading from MongoDB 61-66**
 - connection, establishing 66
 - databases and collections, navigating 67
 - data filtration, with MQL 69-71
 - documents, querying 68
 - time series data, working with 71-74
- data, reading from private S3 bucket 18**
 - Boto3, using 19, 20
 - pandas, using with storage_options 18, 19
- data, reading from relational database 36-38**
 - high-level ORM approach 36
 - low-level connection approach 36
 - pandas, using 42-44
 - psycopg, using 38-44
 - SQLAlchemy, using 42-46
- data, reading from Snowflake**
 - Snowflake Python Connector 58
 - Snowpark API 59, 60
 - Snowpark API, using 56, 57
 - SQLAlchemy API 58, 59
 - SQLAlchemy, using 54, 55
- datasets**
 - loading 486, 487
- date-based features 494**
- date offsets**
 - custom holiday calendar, defining 175-177
 - existing holiday offset, using 174
 - working with 170-173
- date_range function 146-148**
- DateTime**
 - converting, with time zone information 165-170
 - format argument, providing to 148-152
- datetime attribute encoding 559**
- DatetimeIndex 1, 3**
 - working with 138-148
- datetime objects 3**
- DBeaver 37**
- DBeaver Community Edition**
 - download link 37
- Deep Isolation Forest (DIF) 719, 720**
- deep learning 565**
- deep learning-based algorithms**
 - used, for outlier detection 721-728
- deep learning models**
 - training 603-605
- deep LSTM 570**
- deep neural network (DNN) 568**
- DeepOD 667**
- delimited files**
 - writing to 83-88
- delta 416**
- Dense layers 571**
- density-based algorithm 669**
- dependent variables 488, 489**
- deseasonalized data 428**
- de-trending 302**
- dialect 45**

differencing 446

direct-recursive (DirRec) 602

direct strategy 517

exploring 532, 533

implementing 530

DirRec strategy 517

discrete value 488

distance-based algorithms

used, for outlier detection 668

distance metric 669

domain knowledge 445

downsampling 237-243

Dropout layers 571

E

ElasticNet 217, 220, 502

ElasticNet regression model 507, 508

empirical cumulative distribution functions (ECDFs) 706

endogenous variables 434, 481

ensemble learning

forecasting with, in sktime library 544-554

ensemble method 499

ensemble method algorithms

used, for outlier detection 714-721

epoch 569

error accumulation 512, 515

error metrics 349

ETS (Error-Trend-Seasonal) models 364

Euclidean distance 278, 669, 687

evaluation metrics

Mean Absolute Error (MAE) 418

Mean Absolute Percent Error (MAPE) 418

Mean Squared Error (MSE) 418

Median Absolute Percent Error (MdAPE) 418

Root Mean Squared Error (RMSE) 418

Symmetric Mean Absolute Percent Error (SMAPE) 418

Excel 1

data, reading from file 9-14

data, writing to 88-94

exogenous variables 434, 481, 548-551, 606

forecasting with, in sktime library 544-554

handling 606

exploitation phase 635

exploration phase 635

Exploratory Data Analysis (EDA) 283

exponential smoothing 347

exploring 347, 348

univariate time series data,
forecasting with 347-357

versus ETS 362

Exponential Smoothing 635

Extra Tree regression 502-505

ExtraTreesRegressor 220

Extremely Randomized Trees (Extra Tree)
regression 499

extreme value theory (EVT) 276

F

fastparquet 26-29

feature engineering 493, 545

feature importance

in ensemble methods 509

methods 511

feature selection 507

Federal Reserve Economic Data (FRED) 435

feed-forward networks (FFNs) 572

filesystem specification (fsspec) 99

financial time series data

volatility, forecasting with GARCH 460-472

fine-tuning 534

first-order differencing 302, 303

forecast error variance decomposition (FEVD) 446

forecasting 285

forecasts

in-sample forecast 333

out-of-sample forecast 333

format argument

providing, to DateTime 148-152

formatting directive 151

forward fill (ffill) 205

Fourier series 752

functional API 576

future covariates 606

G

GARCH models 401

gated recurrent unit (GRU) 565

Gaussian error function (erf) 690

Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model 460

generative AI 572

GitHub

data, reading from 16, 17

global outliers 668

GMM (Gaussian Mixture Model) 666

Goldfeld-Quandt test 319

Google Cloud Storage

reference link 101

writing to, with google-cloud-storage library 100

writing to, with pandas 97

google-cloud-storage library

used, for writing to Google Cloud Storage 100

Google Drive 75

Gradient Boosting 499, 502, 520

Granger causality test 434, 445, 449, 450

grangercausalitytests function 451

granularity parameter 124

grid search 336, 631

H

Hamiltonian Monte Carlo (HMC) algorithm 413

Hampel filter 270

used, for detecting outliers 270-273

working 273-276

harmonics 752

HDoutliers algorithm 276

heteroskedasticity 306, 310, 460

HierarchicalForecast library 472

histograms 244, 250, 372

historical_forecasts() method 654

Hodrick-Prescott filter 286, 296

cyclical component 296

trend component 296

Holt's exponential smoothing 347, 358, 359

Holt-Winters' exponential smoothing 347, 359, 360

Holt-Winters formulas 349

homoskedasticity 306-310, 322, 460

testing 311, 312

HQIC (Hannan-Quinn Information Criterion) 399

HTML

data, reading from 20-24

html5lib 15

HTML parsing 15

Huber regression 520, 521

hyperbolic tangent 571

hyperparameter optimization

reference link 634

hyperparameters 336

hyperparameter tuning 391, 502, 534, 566
 for deep learning forecasters 621, 622
 with sktime 534-543
 working 631, 632

I

impulse response analysis 445
Impulse Response Function (IRF) 449
impulse responses (IRs) 446
independent variables 488, 489
index 1
information criteria metrics 349
Informer 572
innovations 471
input modulation 573
interdependencies 450
interpolation 223
 linear interpolation 224
 missing data, handling with 223-229
 polynomial interpolation 224
interquartile range (IQR) 237
Isolation Forest (IForest)
 algorithm 714, 715, 662
IterativeImputer class 216

J

Jarque-Bera test 318, 467

K

Kalman filters 734
kd_tree 688
Keras 565, 574
 using 576
Keras methods
 functional API 576
 model subclassing 576
 sequential API 576

kernel 250
kernel-based algorithms
 used, for outlier detection 708-714
kernel density estimation (KDE)
 plot 250, 371, 372
kernel function 498, 708
key-value pair 69
KMeansAD 700
K-Nearest Neighbors (KNN) 488, 662, 668
 for global outlier detection 668, 669
 perspective 669
KNeighborsRegressor 220
KNNImputer class 218, 219
KNN regressor 540
Kolmogorov-Smirnov test 263, 308, 316-318
Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test 298-301, 391

L

L1 regularization 507
L2 regularization 507
lagged features 666
lag plot 251
large language models (LLMs) 572
 forecasting with 572-602
Lasso 502
Lasso regression 507, 508
Last Observation Carried Forward (LOCF) 200
latent state 734
latent variables 734
lazy evaluation 60
learning rate 570
letter-value plot 248
Linear 569
linear combination 435
linear interpolation 224

linear regression 217, 220, 430, 498
 linear regression model 508
 Ljung-Box test 322, 468
 LLikelihood ratio test 451
 Locally Estimated Scatterplot Smoothing (LOESS) 295
 Local Outlier Factor (LOF) 662, 666-668
 for local outlier detection 669
 perspective 670
 local outliers 273, 668
 logistic regression 488
 log likelihood 442
 log transformation 292, 304
 long short-term memory (LSTM) 565
 using, from NeuralForecast 615-617
 loss function 570
 lxml 15

M

machine learning, parameters types
 hyperparameters 631
 model parameters 631
 MAE 644
 Mahalanobis distance 687
 Manhattan distance 669, 687
 Markov Chain Monte Carlo (MCMC) 413
 Matplotlib 244
 matrix notation 497
 Matrix Profile 278
 outliers, detecting with 278-281
 Maximum a Posteriori (MAP) 413
 maximum likelihood estimation (MLE) 362
 maximum likelihood method 465
 MeanAbsoluteError class
 using 543

mean_absolute_error function
 using 543
 Mean Absolute Error (MAE) 501, 569
 Mean Absolute Percentage Error (MAPE) 500
 Mean Absolute Scaled Error (MASE) 500, 569
 mean imputation 199
 mean quantile loss (MQL) 646
 mean reversion 304, 460
 Mean Squared Error (MSE) 498, 569
 median absolute deviation (MAD) 264
 memory mapping 29
 metadata 28
 Minkowski distance 669, 687
 missing data 187, 189
 estimating 188
 handling, IterativeImputer used 216-218
 handling, KNNImputer used 218-223
 handling, pandas DataFrame used 207
 handling, Python dictionary used 206, 207
 handling, with interpolation 224-230
 handling, with multivariate imputation 215
 handling, with univariate imputation
 using pandas 199-205
 handling, with univariate imputation using
 scikit-learn 208-214
 rows, dropping 188
 tagging 188
 ML algorithms 496
 MLForecast library 472
 model agnostic 510
 model subclassing 576
 modified z-score 264, 265
 used, for detecting outliers 264-270
 working 268
 MongoDB
 connection, establishing 66
 data, reading from 61-66
 time series data, writing to 117-127

MongoDB Compass

download link 62

MongoDB Query Language (MQL)

data, filtering with 69, 70

**MonteCarloDropout (mc_dropout)
layer 644-646****Monte Carlo (MC) 644****Monthly Energy Consumption dataset 567****moving averages 287****multi-class classification 488****multi-layer perceptron (MLP) 647****multi-output strategy**

implementing 530

testing 530, 531

MultiPatchFormer 572**multiple-output strategy 517****multiple regression problem 496****Multiple Seasonal-Trend Decomposition using
LOESS (MSTL) 731, 735, 740-742**

for decomposing, time series with multiple
seasonal patterns 734-739

working 739

multi-rate sampling 653**multi-step forecasting 512-516, 522**

with scikit-learn 511-522

multivariate forecasting 606**multivariate imputation 199**

used, for handling missing data 215

multivariate time series

handling 606, 618-621

multivariate time series modeling 401**N****naïve forecaster 549****naïve regressor 544****National Oceanic and Atmospheric
Administration (NOAA) 26****natural language processing (NLP) 565****N-BEATS 565, 647****Neural Basis Expansion Analysis Time Series
(N-BEATS) 572****NeuralForecast 594****NeuralForecast for direct multi-output**

using 594-599

NeuralForecast library 472**NeuralForecast model capabilities**

reference link 606

NeuralForecast's LSTM class

reference link 605

**Neural Hierarchical Interpolation for Time
Series (N-HITS) 572****NeuralProphet**

for forecasting, time series with multiple
seasonal patterns 757-765

**Next Observation Carried Backward
(NOCB) 200****N-HiTS 565, 647****N-HiTSModel class**

reference link 658

Nixtla 594**n_neighbors 671****non-linear trends 403****non-parametric methods 233-236, 244****non-seasonal ARIMA**

univariate time series data,
forecasting with 365-379

non-seasonal components 341**non-stationary process 298****normality 306-308, 316, 322**

testing 308-311

null hypothesis 299**number of epochs 570****number of lags 365****number of layers and nodes 570****numerical feature scaling 552****NumPy array 514**

O

objective function 498

Object-Relational Mapper (ORM) 42, 110

Occam's Razor 389

OCSB (Osborn-Chui-Smith-Birchenhall) 392

OCSVM (One-Class SVM Detector) 666

One-Class SVM (OCSVM) 710, 713

contamination 713

gamma 713

kernel 713

nu parameter 713

OneHotEncoding 547

one-step forecast 489

one-step forecasting 511, 574

with scikit-learn 496-506

one-time anomaly 445

openpyxl 89

opposite null hypotheses 299

Optuna 622

reference link 634

using, with Darts 622-628

using, with NeuralForecast 628-631

visualizations 633

Ordinary Least Squares (OLS) 222, 305, 429

outlier detection 237, 665

outlier detection, with distance-based algorithms 668

contextual outliers, detecting with feature engineering 681-683

contextual outliers, detecting with PyODAdapter 684-687

contextual outliers, detecting with sliding window 677-681

KNN, for global outlier detection 668, 669

LOF, for local outlier detection 669

outliers, detecting after decomposition 673-677

point outliers, detecting 670-673

working 687-694

outliers 233

collective outliers 233

contextual outliers 233

detecting, with clustering-based algorithms 695-701

detecting, with deep learning-based algorithms 721-729

detecting, with ensemble method algorithms 714-721

detecting, with Hampel filter 270-276

detecting, with Matrix Profile 278-281

detecting, with modified z-score 264-270

detecting, with probabilistic and statistical algorithms 703-707

detecting, with STRAY 276, 277

detecting, with Tukey method 253-259

detecting, with kernel-based algorithms 708-714

detecting, with visualizations 244-252

detecting, with z-score 259-264

global outlier 233

in time-series data 236, 237

point outlier 233

overfitting 225

P

pandas 1

used, for writing pickle files 76-78

used, for writing to Amazon S3 97

used, for writing to Azure Blob Storage 98, 99

used, for writing to Google Cloud Storage 97

pandas library 104, 190, 238

pandas.read_sql function

reference link 49

pandas.read_sql_query function

reference link 50

pandas.read_sql_table function

reference link 50

- Parameter F test** 451
- parameters**
 - versus hyperparameters 336
- parametric methods** 233, 236, 244
- Parquet** 1
- Parquet files**
 - all partitions, reading 26
 - data, reading from 25
 - specific partitions, reading 26, 27
 - working with 28-31
- parsimonious models** 441
- partial autocorrelation function (PACF)**
 - plotting 335-345
- partitioning** 28
- parts per million (ppm)** 287
- past covariates** 606
- past covariates only** 606
- pattern anomalies** 278
- Pearson correlation** 322
- percentiles** 255
- percent point function (PPF)** 265
- permutation importance** 509, 510
- pickle files**
 - used, for serializing time series data 76-83
 - writing, with pandas 76-78
 - writing, with pickle library 79
- piecewise regression** 416
- pip** 15
- plot_forecast()** function 585
- point anomalies** 278
- point outlier detection** 662, 665
- point outliers** 665, 666
- polynomial degrees** 365
- polynomial interpolation** 224
- position encoding** 559
- POSIX time** 152
- PostgreSQL** 105
 - reference link 37
- power transformations**
 - applying 306-308
- Predictive Mean Matching (PMM)** 222
- predictors** 488
- preprocessing** 6
- pretrained models** 565
- private S3 bucket**
 - data, reading from 18
- proactive schedulers** 575
- probabilistic and statistical algorithms**
 - used, for outlier detection 703-707
- Probability Plot Correlation Coefficient** 320
- Prophet** 360, 401, 403
 - holiday configuration 412
 - holidays and regressors, incorporating 419-426
 - seasonality configuration 411, 412
 - trend configuration 411
 - uncertainty configuration 413-415
 - uncertainty intervals 413
 - univariate time series data,
 - forecasting with 402-410
 - working 410
- proximity-based algorithm** 669
- psycopg**
 - using 38-41
- public S3 bucket**
 - data, reading from 17, 18
- p-value** 299
- p-value result** 304
- PyArrow** 26, 29-31
- PyMongo API**
 - reference link 74
- PyOD** 665, 667
- PyODAdapter class** 666
- PySpark** 60

PyStan 403

Python 103

Python dictionary

using 206

PyTorch 565, 585, 757

PyTorch Lightning 585

Q

QQ plot 269

quantile loss (QL) 646

quantile method 685

quartiles 254, 255

R

Radial Basis Function (RBF) kernel 498

Random Forest 488, 502

Random Forest regressor 538

Ray Tune 622

reference link 632, 634

reactive schedulers 575

read_dataset function 187

rectified linear unit (ReLU) 569

recurrent neural networks (RNNs) 565

recursive multi-step strategy 513

recursive strategy 512, 516

redshift_connector

used, for writing to Amazon Redshift 115, 116

regression

multiple regression 488

multivariate multiple regression 488

simple regression 488

regression algorithms

elastic net regression 488

Lasso regression 488

linear regression 488

regularization 507

relational database

data, reading from 36

time series data, writing to 105-111

relational database (RDB) 632

representation-based outliers 722

resampling 237

time-series data 237-243

residual autocorrelation 344

residuals 310

residual standardization 344

Residual Sum of Squares (RSS) 498

ResNet architectures 637

Resource API 20, 99

ridge 502

ridge regression 505, 507

ridge regression model 508

R MICE package 220

robust z-score 265

rolling-window approach 480

Root Mean Squared Error (RMSE) 189, 500

calculating 188

S

s3fs library 15, 18

S3 URL support 15

SARIMA 635

model parameters 475

saturated forecasts 403

scale-invariant 440

scikit-learn 208, 270, 440

estimators 209

MultiOutputRegressor 529

multi-step forecasting, with 511-522

one-step forecasting, with 496-506

SciPy library 269, 307, 308, 671

Seaborn 244

Search and TRace Anomaly (STRAY)
 algorithm 276
 implementing 276, 277
 outliers, detecting with 276, 277
seasonal and trend decomposition
 with LOESS (STL) 286
 using 291-294
seasonal ARIMA
 univariate time series data,
 forecasting with 380-390
seasonal component 341
seasonal_decompose function
 using 287-290
seasonal decomposition 286
seasonal differencing 302
second-order differencing 302, 303
secret access key 19
semi-supervised learning 565
seq2seq models 565
sequential API 576
 working 576-585
sequential data 565
Series 1
session.create_dataframe() method
 usage, in Snowpark 134
Shapiro-Wilk test 308, 316-318
Sigmoid 569
Sigmoid activation functions 571
significant autocorrelation 343
simple exponential smoothing
 (SES) 347, 357, 358, 364, 427
SimpleImputer 211, 212
SimpleRNN 570
single variable 199

sktime library 270, 540
 ExpandingWindowSplitter 540
 forecasting with 522-529
 hyperparameter tuning with 534-543
 SingleWindowSplitter 540
 SlidingWindowSplitter 540
sliding window technique 270, 493
Small and Medium Sized Enterprises (SME) 607
Snowflake
 data, reading from 50-53
Snowflake Connector for Python
 reference link 61
Snowflake database
 snowflake-connector-python library,
 using 129, 130
 Snowpark API, using 132, 133
 SQLAlchemy, using 130-132
 time series data, writing to 127, 129
Snowflake Python Connector
 using 52, 53, 58
Snowflake SQLAlchemy
 reference link 61
Snowpark
 session.create_dataframe() method, using 134
Snowpark API 59, 60
 reference link 61
 using 56
Snowpark Python 51
Softmax 569
spline 224
SQLAlchemy 45, 105
 connection URL, generating 47
 reference link 49
 used, for writing to Amazon Redshift 111-115
 using 54, 55
SQLAlchemy API 58, 59
 reference link 117

sqlalchemy-redshift 112
 installing 47
SSR-based chi2 test 451
SSR-based F test 451
Stan 403
standardized residuals 310
Standardized Residuals plot 372, 466
StandardScaler 547
state-space models (SSMs) 733, 734
stationarity 298, 304
stationary time series 298
statistical algorithms 496
statistical features 494
statistical tests 450
StatsForecast 594
StatsForecastAutoARIMA
 using 399, 400
StatsForecast library 362, 472
 used, for automated univariate
 forecasting 472-484
statsmodels library 221, 286, 308, 322, 427
 using 269
stepwise algorithm 391
stochastic level and trend model 750
strong stationarity 298
Subject Matter Experts (SMEs) 661
sum of squared error (SSE) 360
**SUOD (Scalable Unsupervised Outlier
 Detection) framework** 727, 728
supervised learning 565
 time-series data, preparing for 489-493
supervised ML 488
 classification category 488
 regression category 488
Support Vector Machine (SVM) 488

Support Vector Regression (RBF) model 509
SVM classification algorithm 498

T

tabular format 496
Tanh activation functions 571
targets 488
TBATS 753
TCNModel
 reference link 647
temporal convolutional network (TCN) 565
 forecasting with 635-638
 implementing, with Darts' TCNModel 638, 639
 key features 637
 reference link 646
 working 640-644
temporal fusion transformer (TFT) 565
TensorFlow 565
Test Statistic ADF value 304
Theta lines 427
Theta model 427, 647
 univariate time series data,
 forecasting with 427-433
ThetaModelResults object 431
threshold 260
time deltas
 working with 158-165
TimeGPT 472
time series
 additive decomposition 285
 cyclical components 285
 decomposition 285, 286
 multiplicative decomposition 285
 residual 285
 seasonality 285
 trend 285
time series collection 61, 118

time series components

- predictable 286
- unpredictable 286

time-series cross-validation 522**time series data 137**

- outliers 236
- preparing, for supervised learning 489-493
- resampling 237-243
- serializing, with pickle files 76-83
- writing, to MongoDB 117-127
- writing, to relational database 105-111
- writing, to Snowflake 127-129

time series decomposition 286

- Hodrick-Prescott filter (hp_filter) 286
- seasonal and trend decomposition,
with LOESS 286
- seasonal decomposition 286

time series outlier detection

- considerations 665-668

time series stationarity 297, 298

- detecting 299-305

time series, with multiple seasonal patterns

- decomposing, with MSTL 734-739
- forecasting, with AutoTBATS 752-757
- forecasting, with NeuralProphet 757-765
- forecasting, with UCM 743-751

time zone information

- DateTime, converting with 165-169

transformer architectures 572**Transformers (N-HiTS)**

- forecasting with 647-650
- reference link 658
- working 651-653

tree-based ensemble method 498**Tree-structured Parzen Estimator Sampler
(TPESampler) 625****trend component 429****Tukey fences 237****Tukey method**

- used, for detecting outliers 253-259

Tukey's fences 255**U****unified transformer 552****unit root tests 299, 304, 391****univariate forecasting 606****univariate imputation 199**

- techniques 200

univariate imputation, with pandas

- used, for handling missing data 199-205

univariate imputation, with scikit-learn

- used, for handling missing data 208-214

univariate time series data

- forecasting, with auto_arima 391-399
- forecasting, with exponential
smoothing 347-357
- forecasting, with non-seasonal ARIMA 365-379
- forecasting, with Prophet 402-410
- forecasting, with seasonal ARIMA 380-390
- forecasting, with Theta 427-433
- forecasting, with VAR 433-449

Unix epoch timestamps

- working with 152-158

Unobserved Components Model (UCM) 743

- for forecasting, time series with multiple
seasonal patterns 743-752

unsupervised learning 565**unsupervised machine learning 488, 662****upsampling 237, 241****URLs**

- data, reading from 14, 15

USFederalHolidayCalendar rules 176**utilsforecast library 478**

V

vanishing gradient problem 573

Variational AutoEncoders (VAEs) 729

VAR models

- evaluating 449-459

- univariate time series data,
forecasting with 433-449

- versus AR models 433

Vector Autoregressive Integrated Moving
Average (VARIMA) 433

Vector Autoregressive Moving Average
(VARMA) 433

vector autoregressive (VAR) class 401

Vector Moving Average (VMA) 433

violin plot 249, 251

visualizations

- for detecting outliers 244-252

volatility 460

volatility forecasting 460

volatility in financial time series data

- forecasting, with GARCH 460-472

voting ensemble 556

W

WaveNet architectures 637

weak stationarity 298

White test 308, 312, 319

window size selection 493

write_pandas method 135

X

XGBoost 555, 556, 635

- forecasting with 555-563

Y

Yeo-Johnson transformation 321

Z

z-score 237

- used, for detecting outliers 259-263

z-score (standardization) 690

