python ™

Interpretable Machine Learning with Python

Build explainable, fair, and robust high-performance models with hands-on, real-world examples

Forewords by:

Aleksander Molak
Author of Casual Inference and
Discovery in Python
Creator of CasualPython.io

Denis Rothman
AI Ethicist and Bestselling Author

Second Edition

Serg Masís

Interpretable Machine Learning with Python

Second Edition

Build explainable, fair, and robust high-performance models with hands-on, real-world examples

Serg Masís

<packt>

BIRMINGHAM—MUMBAI

"Python" and the Python Logo are trademarks of the Python Software Foundation.

Interpretable Machine Learning with Python
Second Edition

Forewords

Artificial Intelligence (**AI**) has recently gone from science fiction to an integral component of our daily lives. Its tremendous rise can be witnessed in applications ranging from AI-driven assistants on our smartphones to advanced robotics in manufacturing, from predictive analytics in healthcare to sophisticated algorithms in finance.

AI's reach has pervaded every sector as computational power and data relentlessly grow. The increasing speed of new human-level AI algorithms has shaken the very fabric of our society.

As we stand on the edge of this new age, it's critical to understand the vast range of AI applications reshaping our workplace. But how can we advance in this era if we cannot understand the output of a model? How can we trust an AI-driven system if we cannot interpret its results? Can we let an AI system we don't understand make decisions for us?

The terms 'interpretation,' 'interpretability,' and 'explainability' have thus emerged as key pillars of AI. They are not buzzwords. They bridge the gap between complex, often obscure, algorithms and human users.

This book dives deep into these fundamental concepts that need to be demystified for beginners and advanced specialists. Serg Masís takes the time to help the reader understand the difference between interpretability and explainability.

The book covers practical use cases in machine learning that will provide the readers with the tools to implement interpretable AI with efficient methods, such as SHAP and LIME. While exploring these models, the book takes the reader into the complexity of ML models and the limitations of interpretable AI.

The key to sustainable AI is transparency. By the end of the book, you will be able to face the challenges of real-life AI implementations that require interpretability for legal reasons and to gain user trust.

This guide will take you to the cutting edge of real-life transparent and efficient AI implementations.

Denis Rothman

AI Ethicist and Bestselling Author

Serg is one of those authors who brings true passion to their work.

Not surprisingly, the first edition of his *Interpretable Machine Learning with Python* became a *de facto* go-to reference for model interpretability and explainability in Python.

Today, the need for transparency in modeling is even greater than when the first edition was published.

Over the last decade, many decision-makers and researchers have increasingly realized the challenges that come with black-box modeling.

This realization stimulated research focused on both – making predictive models more transparent and tying modeling results to real-world processes using causal inference.

To the best of my knowledge, Serg's new book offers the most systematic, clear, and comprehensive coverage of explainability and interpretability methods in Python available on the market.

Even if you're a seasoned practitioner, you'll likely learn something new from this book.

Let the journey begin!

Aleksander Molak

Author of Causal Inference and Discovery in Python Creator of CausalPython.io

São Paulo / Zurich, October 2023

Contributors

About the author

Serg Masís has been at the confluence of the internet, application development, and analytics for the last two decades. Currently, he's a lead data scientist at Syngenta, a multinational agribusiness company with the mission to improve global food security. Previously, he co-founded a search engine startup incubated by Harvard Innovation Labs, which combined the power of cloud computing and machine learning with principles in decision-making science to expose users to new places and events efficiently. Whether it pertains to leisure activities, plant diseases, or customer lifetime value, Serg is passionate about providing the often-missing link between data and decision-making.

With deepest appreciation to my wife for her unwavering love and belief in me. This book is a tribute to you and the beautiful future we anticipate with our baby.

About the reviewer

Dr. Ali El-Sharif is a professor teaching Data Analytics and Cybersecurity at St. Clair College and University of Windsor. Dr. El-Sharif earned his Ph.D. in Information Systems at Nova Southeastern University, focusing on explainable artificial intelligence. He also received a Master's in Information Security from Nova Southeastern University, a Master of Business Administration (MBA), and a Bachelor of Computer Engineering from Wright State University in Dayton, Ohio. Dr. El-Sharif transitioned to academia after 20 years of being an industry practitioner, holding roles including software developer, solution architect, project manager, and management consultant.

Dr. Ali El-Sharif is the co-author of the upcoming book *Building Responsible AI with Python* (with Packt Publishing).

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:
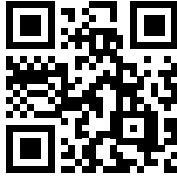
https://packt.link/inml

Table of Contents

Chapter 6: Anchors and Counterfactual Explanations 185

Chapter 10: Feature Selection and Engineering for Interpretability   359

Preface

The title of this book suggests its central themes: **interpretation**, **machine learning**, and **Python**, with the first theme being the most crucial.

So, why is interpretation so important?

Interpretable machine learning, often referred to as **Explainable AI** (**XAI**), encompasses a growing array of techniques that help us glean insights from models, aiming to ensure they are safe, fair, and reliable – a goal I believe we all share for our models.

With the rise of AI superseding traditional software and even human tasks, machine learning models are viewed as a more advanced form of software. While they operate on binary data, they aren't typical software; their logic isn't explicitly coded by developers but emerges from data patterns. This is where interpretation steps in, helping us understand these models, pinpoint their errors, and rectify them before any potential mishaps. Thus, interpretation is essential in fostering trust and ethical considerations in these models. And it's worth noting that in the not-so-distant future, training models might move away from coding to more intuitive drag-and-drop interfaces. In this context, understanding machine learning models becomes an invaluable skill.

Currently, there's still a significant amount of coding involved in data preprocessing, exploration, model training, and deployment. And while this book is rich with Python examples, it's not merely a coding guide removed from practical applications or the bigger picture. The book's essence is to prioritize the *why* before the *how* when it comes to **interpretable machine learning**, as interpretation revolves around the question of *why*.

Therefore, most chapters of this book kickoff by outlining a mission (the *why*) and then delving into the methodology (the *how*). The aim is to achieve the mission using the techniques discussed in the chapter, with an emphasis on understanding the results. The chapters wrap up by pondering on the practical insights gained from the exercises.

The structure of this book is progressive, starting from the basics and moving to more intricate topics. The tools utilized in this book are open source and are products of leading research institutions like Microsoft, Google, and IBM. Even though interpretability is a vast research field with many aspects still in the developmental phase, this book doesn't aim to cover it all. Its primary goal is to delve deeply into a selection of interpretability tools, making it beneficial for those working in the machine learning domain.

The book's initial section introduces interpretability, emphasizing its significance in the business landscape and discussing its core components and challenges. The subsequent section provides a detailed overview of various interpretation techniques and their applications, whether it's for classification, regression, tabular data, time series, images, or text. In the final section, readers will engage in practical exercises on model tuning and data training for interpretability, focusing on simplifying models, addressing biases, setting constraints, and ensuring dependability.

By the book's conclusion, readers will be adept at using interpretability techniques to gain deeper insights into machine learning models.

Who this book is for

This book caters to a diverse audience, including:

- Data professionals who face the growing challenge of explaining the functioning of AI systems they create and manage and seek ways to enhance them.
- Data scientists and machine learning professionals aiming to broaden their expertise by learning model interpretation techniques and strategies to overcome model challenges from fairness to robustness.
- Aspiring data scientists who have a basic grasp of machine learning and proficiency in Python.
- AI ethics officers aiming to deepen their knowledge of the practical aspects of their role to guide their initiatives more effectively.
- AI project supervisors and business leaders eager to integrate interpretable machine learning in their operations, aligning with the values of fairness, responsibility, and transparency.

What this book covers

Chapter 1, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, introduces machine learning interpretation and related concepts, such as interpretability, explainability, black-box models, and transparency, providing definitions for these terms to avoid ambiguity. We then underpin the value of machine learning interpretability for businesses.

Chapter 2, *Key Concepts of Interpretability*, uses a cardiovascular disease prediction example to introduce two fundamental concepts (feature importance and decision regions) and the most important taxonomies used to classify interpretation methods. We also detail what elements hinder machine learning interpretability as a primer for what lies ahead.

Chapter 3, *Interpretation Challenges*, discusses the traditional methods used for machine learning interpretation for regression and classification with a flight delay prediction problem. We will then examine the limitations of these traditional methods and explain what makes "white-box" models intrinsically interpretable and why we cannot always use white-box models. To answer this question, we consider the trade-off between prediction performance and model interpretability. Finally, we will discover some new "glass-box" models that attempt to not compromise in this trade-off.

Chapter 4, *Global Model-Agnostic Interpretation Methods*, explores **Partial Dependence Plots** (**PDP**) and game-theory-inspired **SHapley Additive exPlanations** (**SHAP**) with used car pricing regression models, then visualizes conditional marginal distribution **Accumulated Local Effects** (**ALE**) plots.

Chapter 5, *Local Model-Agnostic Interpretation Methods*, covers local interpretation methods, explaining a single or a group of predictions. To this end, the chapter covers how to leverage SHAP and **Local Interpretable Model-agnostic Explanations** (**LIME**) for local interpretations with a chocolate bar rating example, with both tabular and text data.

Chapter 6, *Anchors and Counterfactual Explanations*, continues with local model interpretations, but only for classification problems. We use a recidivism risk prediction example to understand how we can explain unfair predictions in a human-interpretable way. This chapter covers anchors, counterfactuals, and the **What-If-Tool** (**WIT**).

Chapter 7, *Visualizing Convolutional Neural Networks*, explores interpretation methods that work with **Convolutional Neural Network** (**CNN**) models with a garbage classifier model. Once we have grasped how a CNN learns with activations, we will study several gradient-based attribution methods, such as saliency maps, Grad-CAM, and integrated gradients, to debug class attribution. Lastly, we will extend our attribution debugging know-how with perturbation-based attribution methods such as feature ablation, occlusion sensitivity, Shapley value sampling, and KernelSHAP.

Chapter 8, *Interpreting NLP Transformers*, discusses how to visualize attention mechanisms in a restaurant review sentiment classifier transformer model, followed by interpreting integrated gradient attributions and exploring the **Learning Interpretability Tool** (**LIT**).

Chapter 9, *Interpretation Methods for Multivariate Forecasting and Sensitivity Analysis*, uses a traffic forecasting problem and **Long Short-Term Memory** (**LSTM**) models to show how to employ integrated gradients and SHAP for this use case. Lastly, the chapter looks at how forecasting and uncertainty are intrinsically linked, and sensitivity analysis – a family of methods designed to measure the uncertainty of a model's output in relation to its input. We study two methods: Morris for factor prioritization and Sobol for factor fixing.

Chapter 10, *Feature Selection and Engineering for Interpretability*, uses a challenging non-profit direct mailing optimization problem to review filter-based feature selection methods, such as Spearman's correlation and learn about embedded methods, such as Lasso. Then, you will discover wrapper methods, such as sequential feature selection and hybrid ones, such as recursive feature elimination, as well as more advanced ones, such as genetic algorithms. Lastly, even though feature engineering is typically conducted before selection, there's value in exploring feature engineering for many reasons after the dust has settled.

Chapter 11, *Bias Mitigation and Causal Inference Methods*, takes a credit card default problem to demonstrate leveraging fairness metrics and visualizations to detect undesired bias. Then, the chapter looks at how to reduce it via preprocessing methods such as reweighting and prejudice remover for in-processing and equalized odds for post-processing. Then, we test treatments for lowering credit card default and leverage causal modeling to determine their **Average Treatment Effects** (**ATE**) and **Conditional Average Treatment Effects** (**CATE**). Finally, we test causal assumptions and the robustness of estimates.

Chapter 12, *Monotonic Constraints and Model Tuning for Interpretability*, continues with the recidivism risk prediction problem from *Chapter 7*. We will learn how to place guardrails with feature engineering on the data side and monotonic and interaction constraints on the model to ensure fairness while also learning how to tune a model when there are several objectives.

Chapter 13, *Adversarial Robustness*, uses a face mask detection problem to cover an end-to-end adversarial solution. An adversary can purposely thwart a model in many ways, and we focus on evasion attacks, such as Carlini and Wagner infinity-norm and adversarial patches, and briefly explain other forms of attack. We explain two defense methods: spatial smoothing preprocessing and adversarial training. Lastly, we demonstrate a robustness evaluation method.

Chapter 14, *What's Next for Machine Learning Interpretability?*, summarizes what was learned in the context of the ecosystem of machine learning interpretability methods. And then speculates on what's to come next!

To get the most out of this book

- You will need a Jupyter environment with Python 3.9+. You can do either of the following:

    - Install one on your machine locally via **Anaconda Navigator** or from scratch with `pip`.
    - Use a cloud-based one, such as **Google Colaboratory**, **Kaggle Notebooks**, **Azure Notebooks**, or **Amazon Sagemaker**.

- The instructions on how to get started will vary accordingly, so we strongly suggest that you search online for the latest instructions for setting them up.

- For instructions on installing the many packages employed throughout the book, please go to the GitHub repository, which will have the updated instructions in the `README.MD` file. We expect these to change over time, given how often packages change. We also tested the code with specific versions detailed in the `README.MD`, so should anything fail with later versions, please install the specific version instead.

- Individual chapters have instructions on how to check that the right packages are installed.

- But depending on the way **Jupyter** was set up, installing packages might be best done through the **command line** or using `conda`, so we suggest you adapt these installation instructions to suit your needs.

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

- If you are not a machine learning practitioner or are a beginner, it is best to read the book sequentially since many concepts are only explained in great detail in earlier chapters. Practitioners skilled in machine learning but not acquainted with interpretability can skim the first three chapters to get the ethical context and concept definitions required to make sense of the rest, but read the rest of the chapters in order. As for advanced practitioners with foundations in interpretability, reading the book in any order should be fine.

- As for the code, you can read the book without running the code simultaneously or strictly for the theory. But if you plan to run the code, it is best to do it with the book as a guide to assist with the interpretation of outcomes and strengthen your understanding of the theory.

- While reading the book, think of ways you could use the tools learned, and by the end of it, hopefully, you will be inspired to put this newly gained knowledge into action!

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/` `Interpretable-Machine-Learning-with-Python-2E/`. In case there's an update to the code, it will be updated on the existing GitHub repository. You can also find the hardware and software list of requirements on the repository in the `README.MD` file.

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/9781803235424`.

Conventions used

There are several 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: "Next, let's define a `device` variable because if you have a CUDA-enabled GPU model, inference will perform quicker."

A block of code is set as follows:

```python
def predict(self, dataset):
    self.model.eval()
    device = torch.device("cuda" if torch.cuda.is_available()\
                          else "cpu")
    with torch.no_grad():
        loader = torch.utils.data.DataLoader(dataset, batch_size = 32)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```python
def predict(self, dataset):
    self.model.eval()
    device = torch.device("cuda" if torch.cuda.is_available()\
                          else "cpu")
    with torch.no_grad():
        loader = torch.utils.data.DataLoader(dataset, batch_size = 32)
```

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

```
pip install torch
```

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: "The **Predictions** tab is selected, and this tab has a **Data Table** to the left where you can select and pin individual data points and a pane with **Classification Results** to the left."

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: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

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.packtpub.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@packtpub.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.packtpub.com`.

Share your thoughts

Once you've read *Interpretable Machine Learning with Python 2e*, 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.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803235424

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Interpretation, Interpretability, and Explainability; and Why Does It All Matter?

We live in a world whose rules and procedures are ever-increasingly governed by data and algorithms.

For instance, there are rules about who gets approved for credit or released on bail, and which social media posts might get censored. There are also procedures to determine which marketing tactics are most effective and which chest x-ray features might diagnose a positive case of pneumonia.

We expect this because it is nothing new!

But not so long ago, rules and procedures such as these used to be hardcoded into software, textbooks, and paper forms, and humans were the ultimate decision-makers. Often, it was entirely up to human discretion. Decisions depended on human discretion because rules and procedures were rigid and, therefore, not always applicable. There were *always* exceptions, so a human was needed to make them.

For example, if you apply for a mortgage, your approval depended on an acceptable and reasonably lengthy credit history. This data, in turn, would produce a credit score using a scoring algorithm. Then, the bank had rules that determined what score was good enough for the mortgage you wanted. Your loan officer could follow it or not.

These days, financial institutions train models on thousands of mortgage outcomes, with dozens of variables. These models can be used to determine the likelihood that you would default on a mortgage with a presumed high accuracy. If there is a loan officer to stamp the approval or denial, it's no longer merely a guideline but an algorithmic decision. How could it be wrong? How could it be right? How and why was the decision made?

Hold on to that thought because, throughout this book, we will be learning the answers to these questions and many more!

Machine learning model interpretation enables you to understand the logic behind a decision and trace back the detailed steps of the process behind the logic. This chapter introduces machine learning interpretation and related concepts, such as interpretability, explainability, black-box models, and transparency. This chapter provides definitions for these terms to avoid ambiguity and underpins the value of machine learning interpretability. These are the main topics we are going to cover:

- What is machine learning interpretation?
- Understanding the difference between interpretation and explainability
- A business case for interpretability

Let's get started!

Technical requirements

To follow the example in this chapter, you will need Python 3, either running in a Jupyter environment or in your favorite **integrated development environment** (**IDE**) such as PyCharm, Atom, VSCode, Py-Dev, or Idle. The example also requires the `pandas`, `sklearn`, `matplotlib`, and `scipy` Python libraries.

> The code for this chapter is located here: `https://packt.link/Lzryo`.

What is machine learning interpretation?

To interpret something is to *explain the meaning of it*. In the context of machine learning, that something is an algorithm. More specifically, that algorithm is a mathematical one that takes input data and produces an output, much like with any formula.

Let's examine the most basic of models, simple linear regression, illustrated in the following formula:

$$\hat{y} = \beta_0 + \beta_1 x_1$$

Once fitted to the data, the meaning of this model is that $\hat{y}$ predictions are a weighted sum of the x features with the β coefficients. In this case, there's only one x **feature** or **predictor** variable, and the y variable is typically called the **response** or **target** variable. A simple linear regression formula single-handedly explains the transformation, which is performed on the input data x_1 to produce the output $\hat{y}$. The following example can illustrate this concept in further detail.

Understanding a simple weight prediction model

If you go to this web page maintained by the University of California, `http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_Dinov_020108_HeightsWeights`, you can find a link to download a dataset of 25,000 synthetic records of the weights and heights of 18-year-olds. We won't use the entire dataset but only the sample table on the web page itself with 200 records. We scrape the table from the web page and fit a linear regression model to the data. The model uses the height to predict the weight.

In other words, $x_1=$ *height* and $y=$ *weight*, so the formula for the linear regression model would be as follows:

$$weight \; = \; \beta_0 + \beta_1 height$$

You can find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/01/WeightPrediction.ipynb`.

To run this example, you need to install the following libraries:

- `pandas` to load the table into a DataFrame
- `sklearn` (`scikit-learn`) to fit the linear regression model and calculate its error
- `matplotlib` to visualize the model
- `scipy` to test the correlation

You should load all of them first, as follows:

```python
import math
import requests
from bs4 import BeautifulSoup
import pandas as pd
from sklearn import linear_model
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
from scipy.stats import pearsonr
```

Once the libraries are all loaded, you use `pandas` to fetch the contents of the web page, like this:

```python
url = \
'http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_Dinov_020108_
HeightsWeights'
page = requests.get(url)
height_weight_df = pd.read_html(url)[1][['Height(Inches)','Weight(Pounds)']]
```

`pandas` can turn the raw **HyperText Markup Language (HTML)** contents of the table into a DataFrame, which we subset to only include only two columns. And voilà! We now have a DataFrame with `Heights(Inches)` in one column and `Weights(Pounds)` in another.

Now that we have the data, we must transform it so that it conforms to the model's specifications. `sklearn` needs it as numpy arrays with (200,1) dimensions, so we must first extract the `Height(Inches)` and `Weight(Pounds)` pandas series. Then, we turn them into (200,) numpy arrays, and, finally, reshape them into (200,1) dimensions. The following commands perform all the necessary transformation operations:

```python
x = height_weight_df['Height(Inches)'].values.reshape(num_records, 1)
y = height_weight_df['Weight(Pounds)'].values.reshape(num_records, 1)
```

Then, we initialize the scikit-learn `LinearRegression` model and `fit` it with the training data, as follows:

```
model = linear_model.LinearRegression().fit(x,y)
```

To output the fitted linear regression model formula in scikit-learn, you must extract the intercept and coefficients. This is the **formula** that explains how it makes predictions:

```
print("ŷ =" + str(model.intercept_[0]) + " + " +\
        str(model.coef_.T[0][0]) + " x₁")
```

The following is the output:

```
ŷ = -106.02770644878132 + 3.432676129271629 x1
```

This tells us that, on average, for every additional pound, there are 3.4 inches of height.

However, *explaining how the model works* is only one way to explain this linear regression model, and this is only one side of the story. The model isn't perfect because the actual outcomes and the predicted outcomes are not the same for the training data. The difference between them is the **error** or **residuals**.

There are many ways of understanding an error in a model. You can use an error function such as `mean_absolute_error` to measure the deviation between the predicted values and the actual values, as illustrated in the following code snippet:

```
y_pred = model.predict(x)
print(mean_absolute_error(y, y_pred))
```

A 7.8 mean absolute error means that, on average, the prediction is 7.8 pounds from the actual value, but this might not be intuitive or informative. Visualizing the linear regression model can shed some light on how accurate these predictions truly are.

This can be done by using a `matplotlib` scatterplot and overlaying the linear model (in blue) and the *mean absolute error* (as two parallel bands in gray), as shown in the following code snippet:

```
plt.scatter(x, y, color='black')
plt.plot(x, y_pred, color='blue', linewidth=3)
plt.plot(x, y_pred + mae, color='lightgray')
plt.plot(x, y_pred - mae, color='lightgray')
```

If you run the preceding snippet, the plot shown here in *Figure 1.1* is what you get as the output:
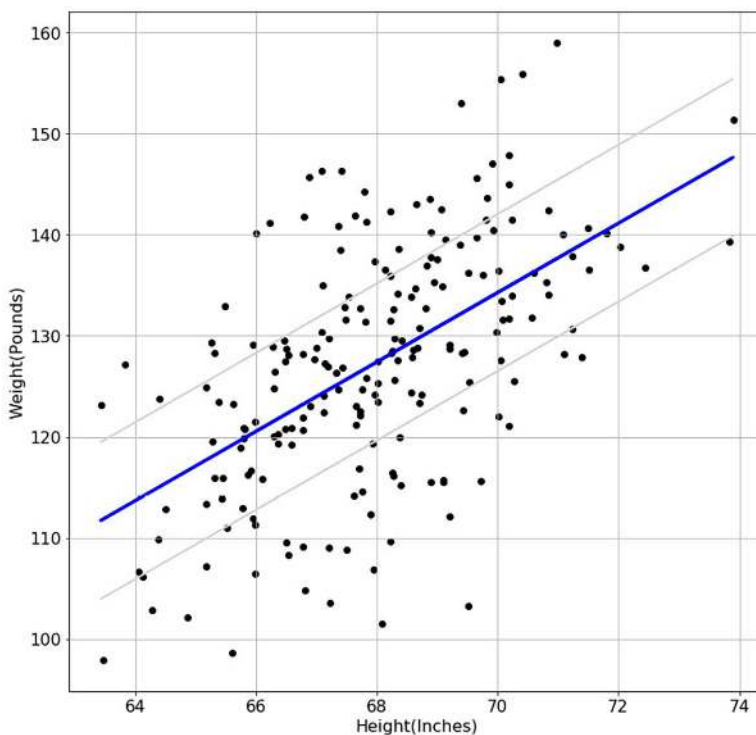
Figure 1.1: Linear regression model to predict weight based on height

As you can appreciate from the plot in *Figure 1.1*, there are many times in which the actuals are 20–25 pounds away from the prediction. Yet the mean absolute error can fool you into thinking that the error is always closer to 8. This is why it is essential to visualize the error of the model to understand its distribution. Judging from this graph, we can tell that there are no red flags that stand out about this distribution, such as residuals being more spread out for one range of heights than for others. Since it is more or less equally spread out, we say it's **homoscedastic**. In the case of linear regression, this is one of many model assumptions you should test for, along with linearity, normality, independence, and lack of multicollinearity (if there's more than one feature). These assumptions ensure that you are using the right model for the job. In other words, the height and weight can be explained with a linear relationship, and it is a good idea to do so, statistically speaking.

With this model, we are trying to establish a linear relationship between *x* height and *y* weight. This association is called a **linear correlation**. One way to measure this relationship's strength is with **Pearson's correlation coefficient**. This statistical method measures the association between two variables using their covariance divided by their standard deviations. It is a number between -1 and 1 whereby the closer the number is to 0, the weaker the association is. If the number is positive, there is a positive association, and if it's negative, there is a negative one. In Python, you can compute Pearson's correlation coefficient with the `pearsonr` function from `scipy`, as illustrated here:

```
corr, pval = pearsonr(x[:,0], y[:,0])
print(corr)
```

The following is the output:

```
0.5568647346122992
```

The number is positive, which is no surprise because as height increases, weight also tends to increase, but it is also closer to 1 than to 0, denoting that it is strongly correlated. The second number produced by the pearsonr function is the *p*-value for testing non-correlation. If we test that it's less than a threshold of 5%, we can say there's sufficient evidence of this correlation, as illustrated here:

```
print(pval < 0.05)
```

It confirms with a True that it is statistically significant.

Understanding how a model performs under different circumstances can help us *explain why it makes certain predictions*, and when it cannot. Let's imagine we are asked to explain why someone who is 71 inches tall was predicted to have a weight of 134 pounds but instead weighed 18 pounds more. Judging from what we know about the model, this margin of error is not unusual even though it's not ideal. However, there are many circumstances in which we cannot expect this model to be reliable. What if we were asked to predict the weight of a person who is 56 inches tall with the help of this model? Could we assure the same level of accuracy? Definitely not, because we fit the model on the data of subjects no shorter than 63 inches. The same is true if we were asked to predict the weight of a 9-year-old, because the training data was for 18-year-olds.

Despite the acceptable results, this weight prediction model was not a realistic example. If you wanted to be more accurate but—more importantly—faithful to what can really impact the weight of an individual, you would need to add more variables. You can add—say—gender at birth, age, diet, and activity levels. This is where it gets interesting because you have to make sure *it is fair to include them, or to exclude them*. For instance, if gender were included and most of our dataset was composed of males, how could you ensure accuracy for females? This is what is called **selection bias**. And what if weight had more to do with lifestyle choices and circumstances such as poverty and pregnancy than gender? If these variables aren't included, this is called **omitted variable bias**. And then, does it make sense to include the sensitive gender variable at the risk of adding bias to the model?

Once you have multiple features that you have vetted for bias, you can find out and *explain which features impact model performance*. We call this **feature importance**. However, as we add more variables, we increase the complexity of the model. Paradoxically, this is a problem for interpretation, and we will explore this in further detail in the following chapters. For now, the key takeaway should be that model interpretation has a lot to do with explaining the following:

- Can we explain how predictions were made, and how the model works?
- Can we ensure that they are reliable and safe?
- Can we explain that predictions were made without bias?

And ultimately, the question we are trying to answer is this:

Can we trust the model?

The three main concepts of interpretable machine learning directly relate to the three preceding questions and have the acronym of **FAT**, which stands for **fairness**, **accountability**, and **transparency**. If you can explain that predictions were made without discernible bias, then there is **fairness.** If you can explain why it makes certain predictions, then there's **accountability**. And if you can explain how predictions were made and how the model works, then there's **transparency**. There are many ethical concerns associated with these concepts, as shown here in *Figure 1.2*. It's portrayed as a triangle because each layer depends on the previous one.
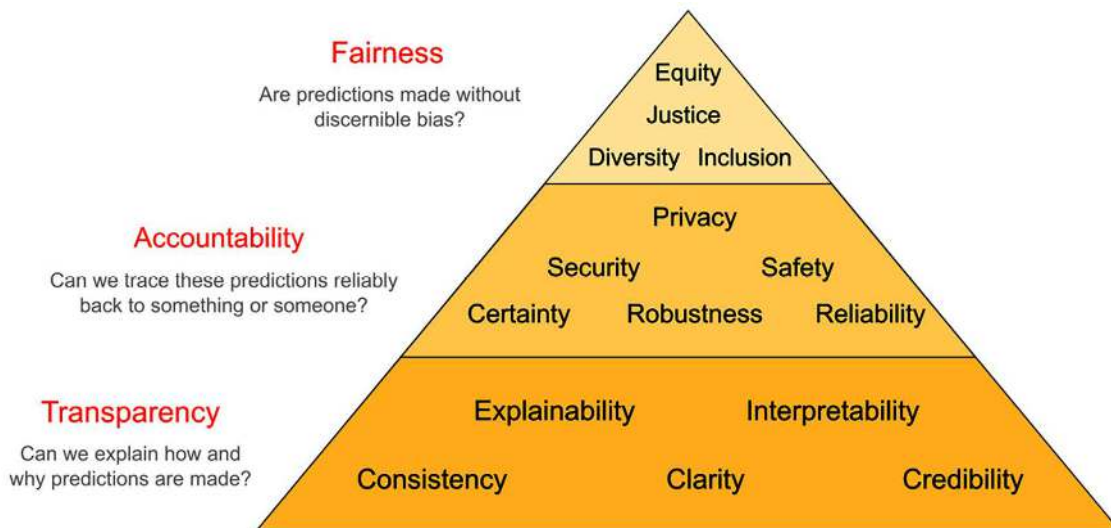
Figure 1.2: Three main concepts of interpretable machine learning

Some researchers and companies have expanded FAT under a larger umbrella of ethical **AI**, thus turning FAT into FATE. However, both concepts very much overlap since interpretable machine learning is how FAT principles and ethical concerns get implemented in machine learning. In this book, we will discuss ethics in this context. For instance, *Chapter 13*, *Adversarial Robustness*, discusses reliability, safety, and security. *Chapter 11*, *Bias Mitigating and Causal Inference Methods*, discusses fairness. That being said, interpretable machine learning can be leveraged with no ethical aim in mind, and also for unethical reasons too.

Understanding the difference between interpretability and explainability

Something you've probably noticed when reading the first few pages of this book is that the verbs *interpret* and *explain*, as well as the nouns *interpretation* and *explanation*, have been used interchangeably. This is not surprising, considering that to interpret is to explain the meaning of something. Despite that, the related terms *interpretability* and *explainability* should not be used interchangeably, even though they are often mistaken for synonyms. Most practitioners don't make any distinction and many academics reverse the definitions provided in this book.

What is interpretability?

Interpretability is the extent to which humans, including non-subject-matter experts, can understand the cause and effect, and input and output, of a machine learning model. To say a model has a high level of interpretability means you can describe in a human-interpretable way its inference. In other words, why does an input to a model produce a specific output? What are the requirements and constraints of the input data? What are the confidence bounds of the predictions? Or, why does one variable have a more substantial impact on the prediction than another? For interpretability, detailing how a model works is only relevant to the extent that it can explain its predictions and justify that it's the right model for the use case.

In this chapter's example, we could explain that there's a linear relationship between human height and weight, so using linear regression rather than a non-linear model makes sense. We can prove this statistically because the variables involved don't violate the assumptions of linear regression. Even when statistics support your explanation, we still should consult with the domain knowledge area involved in the use case. In this scenario, we rest assured, biologically speaking, because our knowledge of human physiology doesn't contradict the relationship between height and weight.

Beware of complexity

Many machine learning models are inherently harder to understand simply because of the math involved in the inner workings of the model or the specific model architecture. In addition to this, many choices are made that can increase complexity and make the models less interpretable, from dataset selection to feature selection and engineering to model training and tuning choices. This complexity makes explaining how machine learning models work a challenge. Machine learning interpretability is a very active area of research, so there's still much debate on its precise definition. The debate includes whether total transparency is needed to qualify a machine learning model as sufficiently interpretable.

This book favors the understanding that the definition of interpretability shouldn't necessarily exclude opaque models, which, for the most part, are complex, as long as the choices made don't compromise their trustworthiness. This compromise is what is generally called **post-hoc interpretability**. After all, much like a complex machine learning model, we can't explain exactly how a human brain makes a choice, yet we often trust its decision because we can ask a human for their reasoning. Post-hoc machine learning interpretation is similar, except it's a human explaining the reasoning on behalf of the model. Using this particular concept of interpretability is advantageous because we can interpret opaque models and not sacrifice the accuracy of our predictions. We will discuss this in further detail in *Chapter 3*, *Interpretation Challenges*.

When does interpretability matter?

Decision-making systems don't always require interpretability. There are two cases that are offered as exceptions, outlined here:

- When incorrect results have no significant consequences. For instance, what if a machine learning model is trained to find and read the postal code in a package, occasionally misreads it, and sends it elsewhere? There's little chance of discriminatory bias, and the cost of misclassification is relatively low. It doesn't occur often enough to magnify the cost beyond acceptable thresholds.

- When there are consequences, but these have been studied sufficiently and validated enough in the real world to make decisions without human involvement. This is the case with a **Traffic-Alert And Collision-Avoidance System** (**TCAS**), which alerts the pilot of another aircraft that poses a threat of a mid-air collision.

On the other hand, interpretability is needed for these systems to have the following attributes:

- **Minable for scientific knowledge:** For example, meteorologists have much to learn from a climate model, but only if it's easy to interpret.
- **Reliable and safe:** The decisions made by a self-driving vehicle must be debuggable so that its developers can understand and correct points of failure.
- **Ethical**: A translation model might use gender-biased word embeddings that result in discriminatory translations, such as a doctor being paired with male pronouns, but you must be able to find these instances easily to correct them. However, the system must be designed in such a way that you can be made aware of a problem before it is released to the public.
- **Conclusive and consistent:** Sometimes, machine learning models may have incomplete and mutually exclusive objectives—for instance, a cholesterol-control system may not consider how likely a patient is to adhere to the diet or drug regimen, or there might be a trade-off between one objective and another, such as safety and non-discrimination.

By explaining the decisions of a model, we can cover gaps in our understanding of the problem—*its incompleteness*. One of the most significant issues is that given the high accuracy of our machine learning solutions, we tend to increase our confidence level to a point where we think we fully understand the problem. Then, we are misled into thinking our solution covers *EVERYTHING*!

At the beginning of this book, we discussed how leveraging data to produce algorithmic rules is nothing new. However, we used to second-guess these rules, and now we don't. Therefore, a human used to be accountable, and now it's the algorithm. In this case, the algorithm is a machine learning model that is accountable for all of the ethical ramifications this entails. This switch has a lot to do with accuracy. The problem is that although a model may surpass human accuracy in aggregate, machine learning models have yet to interpret their results as a human would. Therefore, it doesn't second-guess its decisions, so as a solution, it lacks a desirable level of completeness. That's why we need to interpret models so that we can cover at least some of that gap. So, why is machine learning interpretation not already a standard part of the data science pipeline? In addition to our bias toward focusing on accuracy alone, one of the biggest impediments is the daunting concept of black-box models.

What are black-box models?

This is just another term for opaque models. A black box refers to a system in which only the input and outputs are observable, and you cannot understand what is transforming the inputs into the outputs. In the case of machine learning, a black-box model can be opened, but its mechanisms are not easily understood.

What are white-box models?

These are the opposite of black-box models (see *Figure 1.3*). They are also known as transparent because they achieve total or near-total interpretation transparency. We call them **intrinsically interpretable** in this book, and we cover them in more detail in *Chapter 3*, *Interpretation Challenges*.

Have a look at a comparison between the models here:
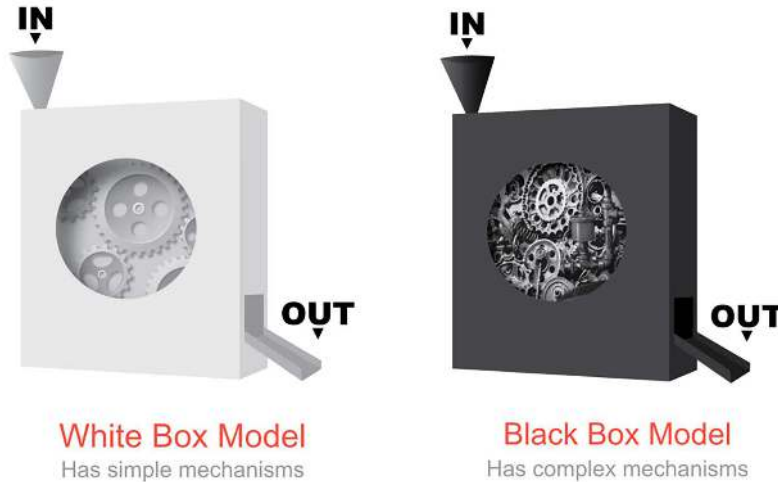
Figure 1.3: Visual comparison between white- and black-box models

Next, we will examine the property that separates white- and black-box models: explainability.

What is explainability?

Explainability encompasses everything interpretability is. The difference is that it goes deeper on the transparency requirement than interpretability because it demands human-friendly explanations for a model's inner workings and the model training process, and not just model inference. Depending on the application, this requirement might extend to various degrees of model, design, and algorithmic transparency. There are three types of transparency, outlined here:

- **Model transparency**: Being able to explain how a model is trained step by step. In the case of our simple weight prediction model, we can explain how the optimization method called **ordinary least squares** finds the β coefficient that minimizes errors in the model.

- **Design transparency**: Being able to explain choices made, such as model architecture and hyperparameters. For instance, we could justify these choices based on the size or nature of the training data. If we were performing a sales forecast and we knew that our sales were seasonal over the year, this could be a sound parameter choice. If we had doubts, we could always use some well-established statistical method to find seasonality patterns.

- **Algorithmic transparency**: Being able to explain automated optimizations such as grid search for hyperparameters, but note that the ones that can't be reproduced because of their random nature—such as random search for hyperparameter optimization, early stopping, and stochastic gradient descent—make the algorithm non-transparent.

Black-box models are called *opaque* simply because they lack *model transparency*, but for many models this is unavoidable, however justified the model choice might be. In many scenarios, even if you outputted the math involved in, for example, training a neural network or a random forest, it would raise more doubts than generate trust. There are at least a few reasons for this, outlined here:

- **Not "statistically grounded":** An opaque model training process maps an input to an optimal output, leaving behind what appears to be an arbitrary trail of parameters. These parameters are optimized to a cost function but are not grounded in statistical theory, which makes predictions hard to justify and explain in statistical terms.

- **Uncertainty and non-reproducibility:** Many opaque models are equally reproducible because they use random numbers to initialize their weights, regularize or optimize their hyperparameters, or make use of stochastic discrimination (such is the case for random forest algorithms).

- **Overfitting and the curse of dimensionality:** Many of these models operate in a high-dimensional space. This doesn't elicit trust because it's harder to generalize on a larger number of dimensions. After all, there's more opportunity to overfit a model the more dimensions you add.

- **The limitations of human cognition:** Transparent models are often used for smaller datasets with fewer dimensions. They also tend to not complicate the interactions between these dimensions more than necessary. This lack of complexity makes it easier to visualize what the model is doing and its outcomes. Humans are not very good at understanding many dimensions, so using transparent models tends to make this much easier to understand. That being said, even these models can get so complex they might become opaque. For instance, if a decision tree model is 100 levels deep or a linear regression model has 100 features, it's no longer easy for us to understand.

- **Occam's razor:** This is what is called the principle of simplicity or parsimony. It states that the simplest solution is usually the right one. Whether true or not, humans also have a bias for simplicity, and transparent models are known for—if anything—their simplicity.

Why and when does explainability matter?

Trustworthy and ethical decision-making is the main motivation for interpretability. Explainability has additional motivations such as causality, transferability, and informativeness. Therefore, there are many use cases in which total or nearly total transparency is valued, and rightly so. Some of these are outlined here:

- **Scientific research:** Reproducibility is essential for scientific research. Also, using statistically grounded optimization methods is especially desirable when causality needs to be established.

- **Clinical trials:** These must also produce reproducible findings and be statistically grounded. In addition to this, given the potential of overfitting, they must use the fewest dimensions possible and models that don't complicate them.

- **Consumer product safety testing:** Much as with clinical trials, when life-and-death safety is a concern, simplicity is preferred whenever possible.

- **Public policy and law:** This is a more nuanced discussion, as part of what legal scholars call **algorithmic governance**, and they have distinguished between **fishbowl transparency** and **reasoned transparency**. Fishbowl transparency seeks total explainability and it's closer to the rigor required for consumer product safety testing, whereas reasoned transparency is one where post-hoc interpretability, as defined earlier, would suffice. One day, the government could be entirely run by algorithms. When that happens, it's hard to tell which policies will align with which form of transparency, but there are many areas of public policy, such as criminal justice, where absolute transparency is necessary. However, whenever total transparency contradicts privacy or security objectives, a less rigorous form of transparency may be preferred.
- **Criminal investigation and regulatory compliance audits:** If something goes wrong, such as an accident at a chemical factory caused by a robot malfunction or a crash by an autonomous vehicle, an investigator would need a **decision trail**. This is to facilitate the assignment of accountability and legal liability. Even when no accident has happened, this kind of auditing can be performed when mandated by authorities. Compliance auditing applies to industries that are regulated, such as financial services, utilities, transportation, and healthcare. In many cases, fishbowl transparency is preferred.

A business case for interpretability

This section describes several practical business benefits of machine learning interpretability, such as better decisions, as well as being more trusted, ethical, and profitable.

Better decisions

Typically, machine learning models are trained and then evaluated against the desired metrics. If they pass quality control against a hold-out dataset, they are deployed. However, once tested in the real world, things can get wild, as in the following hypothetical scenarios:

- A high-frequency trading algorithm could single-handedly crash the stock market.
- Hundreds of smart home devices might inexplicably burst into unprompted laughter, terrifying their users.
- License-plate recognition systems could incorrectly read a new kind of license plate and fine the wrong drivers.
- A racially biased surveillance system could incorrectly detect an intruder, and because of this guards shoot an innocent office worker.
- A self-driving car could mistake snow for a pavement, crash into a cliff, and injure passengers.

Any system is prone to error, so this is not to say that interpretability is a cure-all. However, focusing on just optimizing metrics can be a recipe for disaster. In the lab, the model might generalize well, but if you don't know why the model is making the decisions, then you can miss an opportunity for improvement. For instance, knowing *what* a self-driving car identifies as a road is not enough, but knowing *why* could help improve the model. If, say, one of the reasons was that road is light-colored like the snow, this could be dangerous. Checking the model's assumptions and conclusions can lead to an improvement in the model by introducing winter road images into the dataset or feeding real-time weather data into the model. Also, if this doesn't work, maybe an algorithmic fail-safe can stop it from acting on a decision that it's not entirely confident about.

One of the main reasons why a focus on machine learning interpretability leads to better decision-making was mentioned earlier when we discussed completeness. If we think a model is complete, what is the point of making it better? Furthermore, if we don't question the model's reasoning, then our understanding of the problem must be complete. If this is the case, perhaps we shouldn't be using machine learning to solve the problem in the first place! Machine learning creates an algorithm that would otherwise be too complicated to program in *if-else* statements, precisely to be used for cases where our understanding of the problem is incomplete!

It turns out that when we predict or estimate something, especially with a high level of accuracy, we think we control it. This is what is called the **illusion of control bias**. We can't underestimate the complexity of a problem just because, in aggregate, the model gets it right almost all the time. Even for a human, the difference between snow and concrete pavement can be blurry and difficult to explain. How would you even begin to describe this difference in such a way that it is always accurate? A model can learn these differences, but it doesn't make it any less complex. Examining a model for points of failure and continuously being vigilant for outliers requires a different outlook, whereby we admit that we can't control the model, but we can try to understand it through interpretation.

The following are some additional decision biases that can adversely impact a model, and serve as reasons why interpretability can lead to better decision-making:

- **Conservatism bias:** When we get new information, we don't change our prior beliefs. With this bias, entrenched pre-existing information trumps new information, but models ought to evolve. Hence, an attitude that values questioning prior assumptions is a healthy one to have.
- **Salience bias:** Some prominent or more visible things may stand out more than others, but statistically speaking, they should get equal attention to others. This bias could inform our choice of features, so an interpretability mindset can expand our understanding of a problem to include other less perceived features.
- **Fundamental attribution error:** This bias causes us to attribute outcomes to behavior rather than circumstances, character rather than situations, and nature rather than nurture. Interpretability asks us to explore deeper and look for the less obvious relationships between our variables or those that could be missing.

One crucial benefit of model interpretation is locating *outliers*. These outliers could be a potential new source of revenue or a liability waiting to happen. Knowing this can help us to prepare and strategize accordingly.

More trusted brands

Trust is defined as a belief in the reliability, ability, or credibility of something or someone. In the context of organizations, trust is their reputation; and in the unforgiving court of public opinion, all it takes is one accident, controversy, or fiasco to lose public confidence. This, in turn, can cause investor confidence to wane.

Let's consider what happened to Boeing after the 737 MAX debacle or Facebook after the Cambridge Analytica elections scandal. In both cases, the technological failures were shrouded in mystery, leading to massive public distrust.

And these were examples of, for the most part, decisions made by people. With decisions made exclusively by machine learning models, this could get worse because it is easy to drop the ball and keep the accountability in the model's corner. For instance, if you started to see offensive material in your Facebook feed, Facebook could say it's because its model was trained with *your data* such as your comments and likes, so it's really a reflection of *what you want to see*. Not their fault—your fault. If the police targeted our neighborhood for aggressive policing because it uses PredPol, an algorithm that predicts where and when crimes will occur, it could blame the algorithm. On the other hand, the makers of this algorithm could blame the police because the software is trained on their police reports. This generates a potentially troubling feedback loop, not to mention an accountability gap. And if some pranksters or hackers physically place strange textured meshes onto a highway (see `https://arxiv.org/pdf/2101.06784.pdf`), this could cause a Tesla self-driving car to veer into the wrong lane. Is this Tesla's fault for not anticipating this possibility, or the hackers' for throwing a monkey wrench into their model? This is called an **adversarial attack**, and we discuss this in *Chapter 13*, *Adversarial Robustness*.

It is undoubtedly one of the goals of machine learning interpretability to make models better at making decisions. But even when they fail, you can show that you tried. Trust is not lost entirely because of the failure itself but because of the lack of accountability, and even in cases where it is not fair to accept all the blame, some accountability is better than none. For instance, in the previous set of examples, Facebook could look for clues as to why offensive material is shown more often and then commit to finding ways to make it happen less, even if this means making less money. PredPol could find other sources of crime-rate datasets that are potentially less biased, even if they are smaller. They could also use techniques to mitigate bias in existing datasets (these are covered in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*). And Tesla could audit its systems for adversarial attacks, even if this delays the shipment of its cars. All of these are interpretability solutions. Once they become common practice, they can lead to an increase in not only public trust—be it from users and customers—but also internal stakeholders such as employees and investors.

Many public relation AI blunders have occurred over the past couple of years. Due to trust issues, many AI-driven technologies are losing public support, to the detriment of both companies that monetize AI and users that could benefit from them. This, in part, requires a legal framework at a national or global level and, at the organizational end for those that deploy these technologies, more accountability.

More ethical

There are three schools of thought for ethics: utilitarians focus on consequences, deontologists are concerned with duty, and teleologicalists are more interested in overall moral character. So, this means that there are different ways to examine ethical problems. For instance, there are useful lessons to draw from all of them. There are cases in which you want to produce the greatest amount of "good," despite some harm being produced in the process. Other times, ethical boundaries must be treated as lines in the sand you mustn't cross. And at other times, it's about developing a righteous disposition, much like many religions aspire to do. Regardless of the school of ethics that we align with, our notion of what it is evolves with time because it mirrors our current values. At this moment, in Western cultures, these values include the following:

- Human welfare
- Ownership and property

- Privacy
- Freedom from bias
- Universal usability
- Trust
- Autonomy
- Informed consent
- Accountability
- Courtesy
- Environmental sustainability

Ethical transgressions are cases whereby you cross the moral boundaries that these values seek to uphold, be it by discriminating against someone or polluting their environment, whether it's against the law or not. Ethical dilemmas occur when you have a choice between options that lead to transgressions, so you have to choose between one and another.

The first reason machine learning is related to ethics is that technologies and ethical dilemmas have an intrinsically linked history.

Even the first widely adopted tools made by humans brought progress but also caused harm, such as accidents, war, and job losses. This is not to say that technology is always bad but that we lack the foresight to measure and control its consequences over time. In AI's case, it is not clear what the harmful long-term effects are. What we can anticipate is that there will be a major loss of jobs and an immense demand for energy to power our data centers, which could put stress on the environment. There's speculation that AI could create an "algocratic" surveillance state run by algorithms, infringing on values such as privacy, autonomy, and ownership. Some readers might point to examples of this already happening.

The second reason is even more consequential than the first. It's that predictive modeling is a technological first for humanity: machine learning is a technology that can make decisions for us, and these decisions can produce individual ethical transgressions that are hard to trace. The problem with this is that accountability is essential to morality because you have to know who to blame for human dignity, atonement, closure, or criminal prosecution. However, many technologies have accountability issues to begin with, because moral responsibility is often shared in any case. For instance, maybe the reason for a car crash was partly due to the driver, mechanic, and car manufacturer. The same can happen with a machine learning model, except it gets trickier. After all, a model's programming has no programmer because the "programming" was learned from data, and there are things a model can learn from data that can result in ethical transgressions. Top among them are biases such as the following:

- **Sample bias:** When your data, the sample, doesn't represent the environment accurately, also known as the population
- **Exclusion bias:** When you omit features or groups that could otherwise explain a critical phenomenon with the data
- **Prejudice bias:** When stereotypes influence your data, either directly or indirectly
- **Measurement bias:** When faulty measurements distort your data

Interpretability comes in handy to mitigate bias, as seen in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*, or even placing guardrails on the right features, which may be a source of bias. This is covered in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*. As explained in this chapter, explanations go a long way in establishing accountability, which is a moral imperative. Also, by explaining the reasoning behind models, you can find ethical issues before they cause any harm. But there are even more ways in which models' potentially worrisome ethical ramifications can be controlled, and this has less to do with interpretability and more to do with design. There are frameworks such as **human-centered design, value-sensitive design**, and **techno-moral virtue ethics** that can be used to incorporate ethical considerations into every technological design choice. An article by Kirsten Martin (`https://doi.org/10.1007/s10551-018-3921-3`) also proposes a specific framework for algorithms. This book won't delve into algorithm design aspects too much, but for those readers interested in the larger umbrella of ethical AI, this article is an excellent place to start.

Organizations should take the ethics of algorithmic decision-making seriously because ethical transgressions have monetary and reputation costs. But also, AI left to its own devices could undermine the very values that sustain democracy and the economy that allows businesses to thrive.

More profitable

As seen already in this section, interpretability improves algorithmic decisions, boosting trust and mitigating ethical transgressions.

When we leverage previously unknown opportunities and mitigate threats such as accidental failures through better decision-making, we are likely to improve the bottom line; and if we increase trust in an AI-powered technology, we are likely to increase its use and enhance overall brand reputation, which also has a beneficial impact on profits. On the other hand, ethical transgressions can occur by design or by accident, and when they are discovered, they adversely impact both profits and reputation.

When businesses incorporate interpretability into their machine learning workflows, it's a virtuous cycle, and it results in higher profitability. In the case of a non-profit or government, profits might not be a motive. Still, finances are undoubtedly involved because lawsuits, lousy decision-making, and tarnished reputations are expensive. Ultimately, technological progress is contingent not only on the engineering and scientific skills and materials that make it possible but its voluntary adoption by the general public.

Summary

This chapter has shown us what machine learning interpretation is and what it is not, and the importance of interpretability. In the next chapter, we will learn what can make machine learning models so challenging to interpret, and how you would classify interpretation methods in both category and scope.

Image sources

- Martin, K. (2019). *Ethical Implications and Accountability of Algorithms*. Journal of Business Ethics 160. 835–850. `https://doi.org/10.1007/s10551-018-3921-3`

Dataset sources

- Statistics Online Computational Resource, University of Southern California. (1993). Growth Survey of 25,000 children from birth to 18 years of age recruited from Maternal and Child Health Centers. Originally retrieved from `http://www.socr.ucla.edu/`

Further reading

- Lipton, Zachary (2017). *The Mythos of Model Interpretability*. ICML 2016 Human Interpretability in Machine Learning Workshop: `https://doi.org/10.1145/3236386.3241340`
- Roscher, R., Bohn, B., Duarte, M.F. & Garcke, J. (2020). *Explainable Machine Learning for Scientific Insights and Discoveries*. IEEE Access, 8, 42200-42216: `https://dx.doi.org/10.1109/ACCESS.2020.2976199`
- Doshi-Velez, F. & Kim, B. (2017). *Towards A Rigorous Science of Interpretable Machine Learning*: `http://arxiv.org/abs/1702.08608`
- Arrieta, A.B., Diaz-Rodriguez, N., Ser, J.D., Bennetot, A., Tabik, S., Barbado, A., Garc'ia, S., Gil-L'opez, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). *Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI*: `https://arxiv.org/abs/1910.10045`
- Coglianese, C. & Lehr, D. (2019). *Transparency and algorithmic governance*. Administrative Law Review, 71, 1-4: `https://ssrn.com/abstract=3293008`
- Weller, Adrian. (2019) *Transparency: Motivations and Challenges*. arXiv:1708.01870 [Cs]: `http://arxiv.org/abs/1708.01870`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

2

Key Concepts of Interpretability

This book covers many model interpretation methods. Some produce metrics, others create visuals, and some do both; some depict models broadly and others granularly. In this chapter, we will learn about two methods, feature importance and decision regions, as well as the taxonomies used to describe these methods. We will also detail what elements hinder machine learning interpretability as a primer to what lies ahead.

The following are the main topics we are going to cover in this chapter:

- Learning about interpretation method types and scopes
- Appreciating what hinders machine learning interpretability

Let's start with our technical requirements.

Technical requirements

Although we began the book with a "toy example," we will be leveraging real datasets throughout this book to be used in specific interpretation use cases. These come from many different sources and are often used only once.

To avoid that, readers spend a lot of time downloading, loading, and preparing datasets for single examples; there's a library called `mldatasets` that takes care of most of this. Instructions on how to install this library are located in the *Preface*. In addition to `mldatasets`, this chapter's examples also use the `pandas`, `numpy`, `statsmodel`, `sklearn`, `seaborn`, and `matplotlib` libraries.

> The code for this chapter is located here: `https://packt.link/DgnVj`.

The mission

Imagine you are an analyst for a national health ministry, and there's a **Cardiovascular Diseases** (**CVDs**) epidemic. The minister has made it a priority to reverse the growth and reduce the caseload to a 20-year low. To this end, a task force has been created to find clues in the data to ascertain the following:

- What risk factors can be addressed.
- If future cases can be predicted, interpret predictions on a case-by-case basis.

You are part of this task force!

Details about CVD

Before we dive into the data, we must gather some important details about CVD in order to do the following:

- Understand the problem's context and relevance.
- Extract domain knowledge information that can inform our data analysis and model interpretation.
- Relate an expert-informed background to a dataset's features.

CVDs are a group of disorders, the most common of which is coronary heart disease (also known as *Ischaemic Heart Disease*). According to the World Health Organization, CVD is the leading cause of death globally, killing close to 18 million people annually. Coronary heart disease and strokes (which are, for the most part, a byproduct of CVD) are the most significant contributors to that. It is estimated that 80% of CVD is made up of modifiable risk factors. In other words, some of the preventable factors that cause CVD include the following:

- Poor diet
- Smoking and alcohol consumption habits
- Obesity
- Lack of physical activity
- Poor sleep

Also, many of the risk factors are non-modifiable and, therefore, known to be unavoidable, including the following:

- Genetic predisposition
- Old age
- Male (varies with age)

We won't go into more domain-specific details about CVD because it is not required to make sense of the example. However, *it can't be stressed enough how central domain knowledge is to model interpretation*. So, if this example was your job and many lives depended on your analysis, it would be advisable to read the latest scientific research on the subject and consult with domain experts to inform your interpretations.

The approach

Logistic regression is one common way to rank risk factors in medical use cases. Unlike linear regression, it doesn't try to predict a continuous value for each of our observations, but it predicts a probability score that an observation belongs to a particular class. In this case, what we are trying to predict is, given x data for each patient, what is the y probability, from 0 to 1, that they have CVD?

Preparations

We will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/tree/main/02/CVD.ipynb`.

Loading the libraries

To run this example, we need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `statsmodels` to fit the logistic regression model
- `sklearn` (scikit-learn) to split the data
- `matplotlib` and `seaborn` to visualize the interpretations

We should load all of them first:

```python
import math
import mldatasets
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
```

Understanding and preparing the data

The data to be used in this example should then be loaded into a DataFrame we call `cvd_df`:

```python
cvd_df = mldatasets.load("cardiovascular-disease")
```

From this, we should get 70,000 records and 12 columns. We can take a peek at what was loaded with `info()`:

```python
cvd_df.info()
```

The preceding command will output the names of each column with its type and how many non-null records it contains:

```
RangeIndex: 70000 entries, 0 to 69999
Data columns (total 12 columns):
age            70000 non-null int64
gender         70000 non-null int64
height         70000 non-null int64
weight         70000 non-null float64
ap_hi          70000 non-null int64
ap_lo          70000 non-null int64
cholesterol    70000 non-null int64
gluc           70000 non-null int64
smoke          70000 non-null int64
alco           70000 non-null int64
active         70000 non-null int64
cardio         70000 non-null int64
dtypes: float64(1), int64(11)
```

The data dictionary

To understand what was loaded, the following is the data dictionary, as described in the source:

- `age`: Of the patient in days (objective feature)
- `height`: In centimeters (objective feature)
- `weight`: In kg (objective feature)
- `gender`: A binary where 1: female, 2: male (objective feature)
- `ap_hi`: Systolic blood pressure, which is the arterial pressure exerted when blood is ejected during ventricular contraction. Normal value: < 120 mmHg (objective feature)
- `ap_lo`: Diastolic blood pressure, which is the arterial pressure in between heartbeats. Normal value: < 80 mmHg (objective feature)
- `cholesterol`: An ordinal where 1: normal, 2: above normal, and 3: well above normal (objective feature)
- `gluc`: An ordinal where 1: normal, 2: above normal, and 3: well above normal (objective feature)
- `smoke`: A binary where 0: non-smoker and 1: smoker (subjective feature)
- `alco`: A binary where 0: non-drinker and 1: drinker (subjective feature)

- • `active`: A binary where 0: non-active and 1: active (subjective feature)
- • `cardio`: A binary where 0: no CVD and 1: has CVD (objective and target feature)

It's essential to understand the data generation process of a dataset, which is why the features are split into two categories:

- • **Objective:** A feature that is a product of official documents or a clinical examination. It is expected to have a rather insignificant margin of error due to clerical or machine errors.
- • **Subjective:** Reported by the patient and not verified (or unverifiable). In this case, due to lapses of memory, differences in understanding, or dishonesty, it is expected to be less reliable than objective features.

At the end of the day, trusting the model is often about trusting the data used to train it, so how much patients lie about smoking can make a difference.

Data preparation

For the sake of interpretability and model performance, there are several data preparation tasks that we can perform, but the one that stands out right now is `age`. Age is not something we usually measure in days. In fact, for health-related predictions like this one, we might even want to bucket them into **age groups** since health differences observed between individual year-of-birth cohorts aren't as evident as those observed between generational cohorts, especially when cross tabulating with other features like lifestyle differences. For now, we will convert all ages into years:

```
cvd_df['age'] = cvd_df['age'] / 365.24
```

The result is a more understandable column because we expect age values to be between 0 and 120. We took existing data and transformed it. This is an example of **feature engineering**, which is when we use the domain knowledge of our data to create features that better represent our problem, thereby improving our models. We will discuss this further in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*. There's value in performing feature engineering simply to make model outcomes more *interpretable* as long as this doesn't significantly hurt model performance. In fact, it might improve predictive performance. Note that there was no loss in data in the feature engineering performed on the age column, as the decimal value for years is maintained.

Now we are going to take a peek at what the summary statistics are for each one of our features using the `describe()` method:

```
cvd_df.describe(percentiles=[.01,.99]).transpose()
```

Figure 2.1 shows the summary statistics outputted by the preceding code. It includes the 1% and 99% percentiles, which tell us what are among the highest and lowest values for each feature:

	count	mean	std	min	1%	50%	99%	max
age	70000.00	53.30	6.76	29.56	39.61	53.95	64.31	64.92
gender	70000.00	1.35	0.48	1.00	1.00	1.00	2.00	2.00
height	70000.00	164.36	8.21	55.00	147.00	165.00	184.00	250.00
weight	70000.00	74.21	14.40	10.00	48.00	72.00	117.00	200.00
ap_hi	70000.00	128.82	154.01	-150.00	90.00	120.00	180.00	16020.00
ap_lo	70000.00	96.63	188.47	-70.00	60.00	80.00	1000.00	11000.00
cholesterol	70000.00	1.37	0.68	1.00	1.00	1.00	3.00	3.00
gluc	70000.00	1.23	0.57	1.00	1.00	1.00	3.00	3.00
smoke	70000.00	0.09	0.28	0.00	0.00	0.00	1.00	1.00
alco	70000.00	0.05	0.23	0.00	0.00	0.00	1.00	1.00
active	70000.00	0.80	0.40	0.00	0.00	1.00	1.00	1.00
cardio	70000.00	0.50	0.50	0.00	0.00	0.00	1.00	1.00

Figure 2.1: Summary statistics for the dataset

In *Figure 2.1*, `age` appears valid because it ranges between 29 and 65 years, which is not out of the ordinary, but there are some anomalous outliers for `ap_hi` and `ap_lo`. Blood pressure can't be negative, and the highest ever recorded was `370`. Keeping these outliers in there can lead to poor model performance and interpretability. Given that the 1% and 99% percentiles still show values in normal ranges according to *Figure 2.1*, there's close to 2% of records with invalid values. If you dig deeper, you'll realize it's closer to 1.8%.

```
incorrect_l = cvd_df[
    (cvd_df['ap_hi']>370)
    | (cvd_df['ap_hi']<=40)
    | (cvd_df['ap_lo'] > 370)
    | (cvd_df['ap_lo'] <= 40)
].index
print(len(incorrect_l) / cvd_df.shape[0])
```

There are many ways we could handle these incorrect values, but because they are relatively few records and we lack the domain expertise to guess if they were mistyped (and correct them accordingly), we will delete them:

```
cvd_df.drop(incorrect_l, inplace=True)
```

For good measure, we ought to make sure that `ap_hi` is always higher than `ap_lo`, so any record with that discrepancy should also be dropped:

```
cvd_df = cvd_df[cvd_df['ap_hi'] >=\
                cvd_df['ap_lo']].reset_index(drop=True)
```

Now, in order to fit a logistic regression model, we must put all objective, examination, and subjective features together as *X* and the target feature alone as *y*. After this, we split *X* and *y* into training and test datasets, but make sure to include `random_state` for reproducibility:

```
y = cvd_df['cardio']
X = cvd_df.drop(['cardio'], axis=1).copy()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.15, random_state=9
)
```

The scikit-learn `train_test_split` function puts 15% of the observations in the test dataset and the remainder in the train dataset, so you end up with *X* and *y* pairs for both.

Now that we have our data ready for training, let's train a model and interpret it.

Interpretation method types and scopes

Now that we have prepared our data and split it into training/test datasets, we can fit the model using the training data and print a summary of the results:

```
log_model = sm.Logit(y_train, sm.add_constant(X_train))
log_result = log_model.fit()
print(log_result.summary2())
```

Printing `summary2` on the fitted model produces the following output:

```
Optimization terminated successfully.
         Current function value: 0.561557
         Iterations 6
                       Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.190
Dependent Variable: cardio           AIC:              65618.3485
Date:               2020-06-10 09:10 BIC:              65726.0502
No. Observations:   58404            Log-Likelihood:   -32797.
Df Model:           11               LL-Null:          -40481.
Df Residuals:       58392            LLR p-value:      0.0000
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
-----------------------------------------------------------------
```

```
                 Coef.    Std.Err.     z      P>|z|    [0.025    0.975]
    ----------------------------------------------------------------
    const          -11.1730    0.2504  -44.6182  0.0000  -11.6638  -10.6822
    age              0.0510    0.0015   34.7971  0.0000    0.0482    0.0539
    gender          -0.0227    0.0238   -0.9568  0.3387   -0.0693    0.0238
    height          -0.0036    0.0014   -2.6028  0.0092   -0.0063   -0.0009
    weight           0.0111    0.0007   14.8567  0.0000    0.0096    0.0125
    ap_hi            0.0561    0.0010   56.2824  0.0000    0.0541    0.0580
    ap_lo            0.0105    0.0016    6.7670  0.0000    0.0075    0.0136
    cholesterol      0.4931    0.0169   29.1612  0.0000    0.4600    0.5262
    gluc            -0.1155    0.0192   -6.0138  0.0000   -0.1532   -0.0779
    smoke           -0.1306    0.0376   -3.4717  0.0005   -0.2043   -0.0569
    alco            -0.2050    0.0457   -4.4907  0.0000   -0.2945   -0.1155
    active          -0.2151    0.0237   -9.0574  0.0000   -0.2616   -0.1685
    ================================================================
```

The preceding summary helps us to understand which *X* features contributed the most to the *y* CVD diagnosis using the model coefficients (labeled `Coef.` in the table). Much like with linear regression, the coefficients are weights applied to the predictors. However, the linear combination exponent is a **logistic function**. This makes the interpretation more difficult. We explain this function further in *Chapter 3*, *Interpretation Challenges*.

We can tell by looking at it that the features with the absolute highest values are `cholesterol` and `active`, but it's not very intuitive in terms of what this means. A more interpretable way of looking at these values is revealed once we calculate the exponential of these coefficients:

```python
np.exp(log_result.params).sort_values(ascending=False)
```

The preceding code outputs the following:

```
cholesterol    1.637374
ap_hi          1.057676
age            1.052357
weight         1.011129
ap_lo          1.010573
height         0.996389
gender         0.977519
gluc           0.890913
smoke          0.877576
alco           0.814627
active         0.806471
const          0.000014
dtype: float64
```

Why the exponential? The coefficients are the **log odds**, which are the logarithms of the *odds*. Also, *odds* are the probability of a positive case over the probability of a negative case, where the **positive case** is the label we are trying to predict. It doesn't necessarily indicate what is favored by anyone. For instance, if we are trying to predict the odds of a rival team winning the championship today, the positive case would be that they own, regardless of whether we favor them or not. Odds are often expressed as a ratio. The news could say the probability of them winning today is 60% or say the odds are 3:2 or 3/2 = 1.5. In log odds form, this would be 0.176, which is the logarithm of 1.5. They are basically the same thing but expressed differently. An exponential function is the inverse of a logarithm, so it can take any *log odds* and return the *odds*, as we have done.

Back to our CVD case. Now that we have the odds, we can interpret what it means. For example, what do the odds mean in the case of cholesterol? It means that the odds of CVD increase by a factor of 1.64 for each additional unit of cholesterol, provided every other feature stays unchanged. Being able to explain the impact of a feature on the model in such tangible terms is one of the advantages of an *intrinsically interpretable* model such as logistic regression.

Although the *odds* provide us with useful information, they don't tell us what matters the most and, therefore, by themselves, cannot be used to measure feature importance. But how could that be? If something has higher odds, then it must matter more, right? Well, for starters, they all have different scales, so that makes a huge difference. This is because if we measure the odds of how much something increases, we have to know by how much it typically increases because that provides context. For example, we could say that the odds of a specific species of butterfly living one day more are 0.66 after their first eggs hatch. This statement is meaningless unless we know the lifespan and reproductive cycle of this species.

To provide context to our odds, we can easily calculate the standard deviation of our features using the `np.std` function:

```
np.std(X_train, 0)
```

The following series is what is outputted by the `np.std` function:

```
age             6.757537
gender          0.476697
height          8.186987
weight         14.335173
ap_hi          16.703572
ap_lo           9.547583
cholesterol     0.678878
gluc            0.571231
smoke           0.283629
alco            0.225483
active          0.397215
dtype: float64
```

As we can tell by the output, binary and ordinal features only typically vary by one at most, but continuous features, such as `weight` or `ap_hi`, can vary 10–20 times more, as evidenced by the standard deviation of the features.

Another reason why *odds* cannot be used to measure feature importance is that despite favorable odds, sometimes features are not statistically significant. They are entangled with other features in such a way they might appear to be significant, but we can prove that they aren't. This can be seen in the summary table for the model, under the `P>|z|` column. This value is called the **p-value**, and when it's less than 0.05, we reject the null hypothesis that states that the coefficient is equal to zero. In other words, the corresponding feature is statistically significant. However, when it's above this number, especially by a large margin, there's no statistical evidence that it affects the predicted score. Such is the case with `gender`, at least in this dataset.

If we are trying to obtain what features matter most, one way to approximate this is to multiply the coefficients by the standard deviations of the features. Incorporating the standard deviations accounts for differences in variances between features. Hence, it is better if we get `gender` out of the way too while we are at it:

```
coefs = log_result.params.drop(labels=['const','gender'])
stdv = np.std(X_train, 0).drop(labels='gender')
abs(coefs * stdv).sort_values(ascending=False)
```

The preceding code produced this output:

```
ap_hi         0.936632
age           0.344855
cholesterol   0.334750
weight        0.158651
ap_lo         0.100419
active        0.085436
gluc          0.065982
alco          0.046230
smoke         0.037040
height        0.029620
```

The preceding table can be interpreted as an **approximation of risk factors** from high to low according to the model. It is also a **model-specific** feature importance method, in other words, a **global model (modular) interpretation method**. There are a lot of new concepts to unpack here, so let's break them down.

Model interpretability method types

There are two model interpretability method types:

- **Model-specific:** When the method can only be used for a specific model class, then it's model-specific. The method detailed in the previous example can only work with logistic regression because it uses its coefficients.

• **Model-agnostic:** These are methods that can work with any model class. We cover these in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, and the next two chapters.

Model interpretability scopes

There are several model interpretability scopes:

• **Global holistic interpretation:** We can explain how a model makes predictions simply because we can comprehend the entire model at once with a complete understanding of the data, and it's a trained model. For instance, the simple linear regression example in *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, can be visualized in a two-dimensional graph. We can conceptualize this in memory, but this is only possible because the simplicity of the model allows us to do so, and it's not very common nor expected.

• **Global modular interpretation:** In the same way that we can explain the role of *parts* of an internal combustion engine in the *whole* process of turning fuel into movement, we can also do so with a model. For instance, in the CVD risk factor example, our feature importance method tells us that `ap_hi` (systolic blood pressure), `age`, `cholesterol`, and `weight` are the *parts* that impact the *whole* the most. Feature importance is only one of many global modular interpretation methods but arguably the most important one. *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, goes into more detail on feature importance.

• **Local single-prediction interpretation:** We can explain why a single prediction was made. The next example will illustrate this concept and *Chapter 5*, *Local Model-Agnostic Interpretation Methods*, will go into more detail.

• **Local group-prediction interpretation:** The same as single-prediction, except that it applies to groups of predictions.

Congratulations! You've already determined the risk factors with a **global model interpretation method**, but the health minister also wants to know whether the model can be used to interpret individual cases. So, let's look into that.

Interpreting individual predictions with logistic regression

What if we used the model to predict CVD for the entire test dataset? We could do so like this:

```
y_pred = log_result.predict(sm.add_constant(X_test)).to_numpy()
print(y_pred)
```

The resulting array is the probabilities that each test case is positive for CVD:

```
[0.40629892 0.17003609 0.13405939 ... 0.95575283 0.94095239 0.91455717]
```

Let's take one of the positive cases; test case #2872:

```
print(y_pred[2872])
```

We know that it predicted positive for CVD because the score exceeds 0.5.

And these are the details for test case #2872:

```
print(X_test.iloc[2872])
```

The following is the output:

```
age              60.521849
gender            1.000000
height          158.000000
weight           62.000000
ap_hi           130.000000
ap_lo            80.000000
cholesterol       1.000000
gluc              1.000000
smoke             0.000000
alco              0.000000
active            1.000000
Name: 46965, dtype: float64
```

So, by the looks of the preceding series, we know that the following applies to this individual:

- A borderline high `ap_hi` (systolic blood pressure) since anything above or equal to 130 is considered high according to the **American Heart Association** (**AHA**).
- Normal `ap_lo` (diastolic blood pressure) also according to AHA. Having high systolic blood pressure and normal diastolic blood pressure is what is known as *isolated systolic hypertension*. It could be causing a positive prediction, but `ap_hi` is borderline; therefore, the condition of *isolated systolic hypertension* is borderline.
- `age` is not too old, but among the oldest in the dataset.
- `cholesterol` is normal.
- `weight` also appears to be in the healthy range.

There are also no other risk factors: glucose is normal, the individual does not smoke nor drink alcohol, and does not live a sedentary lifestyle, as the individual is active. It is not clear exactly why it's positive. Are the age and borderline *isolated systolic hypertension* enough to tip the scales? It's tough to understand the reasons for the prediction without putting all the predictions into context, so let's try to do that!

But how do we put everything in context at the same time? We can't possibly visualize how one prediction compares with the other 10,000 for every single feature and their respective predicted CVD diagnosis. Unfortunately, humans can't process that level of dimensionality, even if it were possible to visualize a ten-dimensional hyperplane!

However, we can do it for two features at a time, resulting in a graph that conveys where the decision boundary for the model lies for those features. On top of that, we can overlay what the predictions were for the test dataset based on all the features. This is to visualize the discrepancy between the effect of two features and all eleven features.

This graphical interpretation method is what is termed a **decision boundary**. It draws boundaries for the classes, leaving areas that belong to one class or another. Such areas are called **decision regions**. In this case, we have two classes, so we will see a graph with a single boundary between `cardio=0` and `cardio=1`, only concerning the two features we are comparing.

We have managed to visualize the two decision-based features at a time, with one big assumption that if all the other features are held constant, we can observe only two in isolation. This is also known as the **ceteris paribus** assumption and is critical in a scientific inquiry, allowing us to *control* some variables in order to *observe* others. One way to do this is to fill them with a value that won't affect the outcome. Using the table of odds we produced, we can tell whether a feature increases as it will increase the odds of CVD. So, in aggregates, a lower value is less risky for CVD.

For instance, `age=30` is the least risky value of those present in the dataset for `age`. It can also go in the opposite direction, so `active=1` is known to be less risky than `active=0`. We can come up with optimal values for the remainder of the features:

- `height=165`.
- `weight=57` (optimal for that `height`).
- `ap_hi=110`.
- `ap_lo=70`.
- `smoke=0`.
- `cholesterol=1` (this means normal).
- `gender` can be coded for male or female, which doesn't matter because the odds for gender (`0.977519`) are so close to 1.

The following `filler_feature_values` dictionary exemplifies what should be done with the features matching their index to their least risky values:

```
filler_feature_values = {
    "age": 30,
    "gender": 1,
    "height": 165,
    "weight": 57,
    "ap_hi": 110,
    "ap_lo": 70,
    "cholesterol": 1,
    "gluc": 1,
    "smoke": 0,
    "alco":0,
    "active":1
}
```

The next thing to do is to create a (1,12) shaped NumPy array with test case #2872 so that the plotting function can highlight it. To this end, we first convert it into NumPy and then prepend the *constant* of 1, which must be the first feature, and then reshape it so that it meets the (1,12) dimensions. The reason for the constant is that in statsmodels, we must explicitly define the **intercept**. For this reason, the logistic model has an additional 0 feature, which always equals 1.

```
X_highlight = np.reshape(
    np.concatenate(([1], X_test.iloc[2872].to_numpy())), (1, 12))
print(X_highlight)
```

The following is the output:

```
[[  1.        60.52184865    1.        158.         62.        130.
   80.         1.            1.          0.          0.          1.    ]]
```

We are good to go now! Let's visualize some decision region plots! We will compare the feature that is thought to be the highest *risk factor*, ap_hi, with the following four most important risk factors: age, cholesterol, weight, and ap_lo.

The following code will generate the plots in *Figure 2.2*:

```
plt.rcParams.update({'font.size': 14})
fig, axarr = plt.subplots(2, 2, figsize=(12,8), sharex=True,
                          sharey=False)
mldatasets.create_decision_plot(
    X_test,
    y_test,
    log_result,
    ["ap_hi", "age"],
    None,
    X_highlight,
    filler_feature_values,
    ax=axarr.flat[0]
)
mldatasets.create_decision_plot(
    X_test,
    y_test,
```

```
        log_result,
        ["ap_hi", "cholesterol"],
         None,
        X_highlight,
        filler_feature_values,
        ax=axarr.flat[1]
    )
    mldatasets.create_decision_plot(
        X_test,
        y_test,
        log_result,
        ["ap_hi", "ap_lo"],
        None,
        X_highlight,
        filler_feature_values,
        ax=axarr.flat[2],
    )
    mldatasets.create_decision_plot(
        X_test,
        y_test,
        log_result,
        ["ap_hi", "weight"],
        None,
        X_highlight,
        filler_feature_values,
        ax=axarr.flat[3],
    )
    plt.subplots_adjust(top=1, bottom=0, hspace=0.2, wspace=0.2)
    plt.show()
```

In the plot in *Figure 2.2*, the circle represents test case #2872. In all the plots bar one, this test case is on the negative (left-hand side) decision region, representing `cardio=0` classification. The borderline high `ap_hi` (systolic blood pressure) and the relatively high `age` are barely enough for a positive prediction in the top-left chart. Still, in any case, for test case #2872, we have predicted a 57% score for CVD, so this could very well explain most of it.

Not surprisingly, by themselves, `ap_hi` and a healthy `cholesterol` value are not enough to tip the scales in favor of a definitive CVD diagnosis according to the model because it's decidedly in the negative decision region, and neither is a normal `ap_lo` (diastolic blood pressure). You can tell from these three charts that although there's some overlap in the distribution of squares and triangles, there is a tendency for more triangles to gravitate toward the positive side as the *y*-axis increases, while fewer squares populate this region:
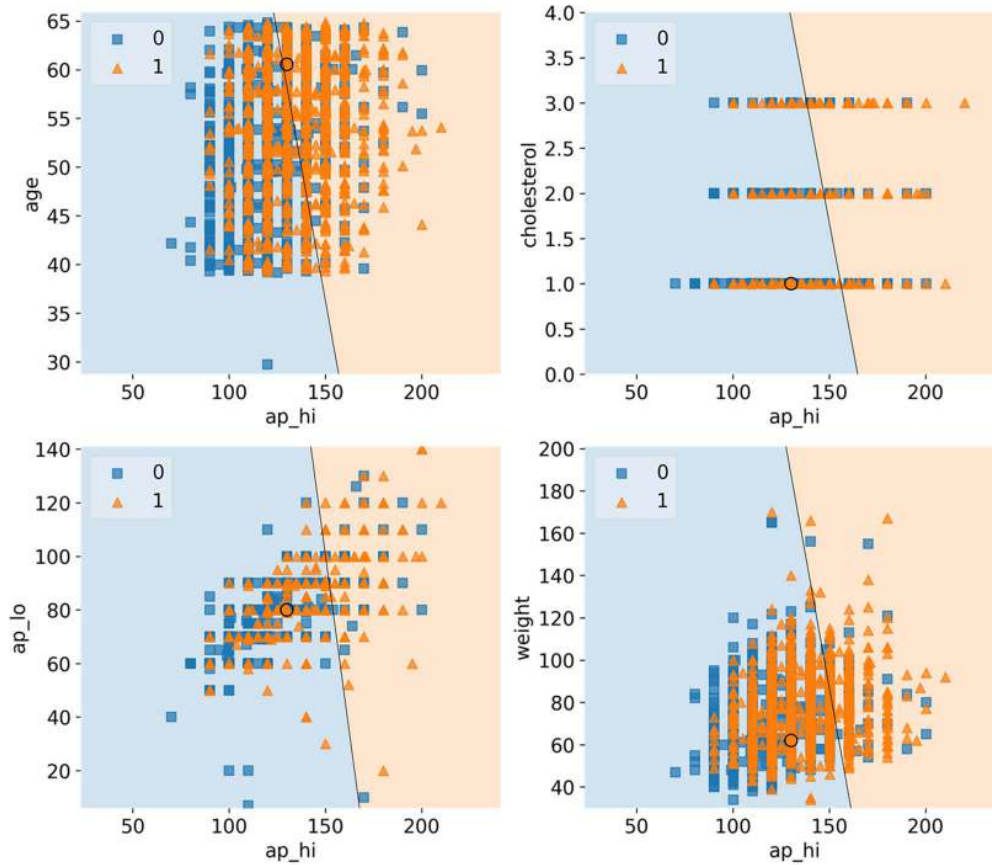
Figure 2.2: The decision regions for ap_hi and other top risk factors, with test case #2872

The overlap across the decision boundary is expected because, after all, these squares and triangles are based on the effects of **all** features. Still, you expect to find a somewhat consistent pattern. The chart with `ap_hi` versus `weight` doesn't have this pattern vertically as `weight` increases, which suggests something is missing in this story... Hold that thought because we are going to investigate that in the next section!

Congratulations! You have completed the second part of the minister's request.

Decision region plotting, a **local model interpretation method**, provided the health ministry with a tool to interpret individual case predictions. You could now extend this to explain several cases at a time, or plot all-important feature combinations to find the ones where the circle is decidedly in the positive decision region. You can also change some of the filler variables one at a time to see how they make a difference. For instance, what if you increase the filler age to the median age of 54 or even to the age of test case #2872? Would a borderline high ap_hi and healthy cholesterol now be enough to tip the scales? We will answer this question later, but first, let's understand what can make machine learning interpretation so difficult.

Appreciating what hinders machine learning interpretability

In the last section, we were wondering why the chart with ap_hi versus weight didn't have a conclusive pattern. It could very well be that although weight is a risk factor, there are other critical *mediating variables* that could explain the increased risk of CVD. A **mediating variable** is one that influences the strength between the independent and target (*dependent*) variable. We probably don't have to think too hard to find what is missing. In *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, we performed linear regression on weight and height because there's a linear relationship between these variables. In the context of human health, weight is not nearly as *meaningful* without height, so you need to look at both.

Perhaps if we plot the decision regions for these two variables, we will get some clues. We can plot them with the following code:

```python
fig, ax = plt.subplots(1,1, figsize=(12,8))
mldatasets.create_decision_plot(
    X_test,
    y_test,
    log_result,
    [3, 4],
    ['height [cm]',
    'weight [kg]'],
    X_highlight,
    filler_feature_values,
    filler_feature_ranges,
    ax=ax
)
plt.show()
```

The preceding snippet will generate the plot in *Figure 2.3*:
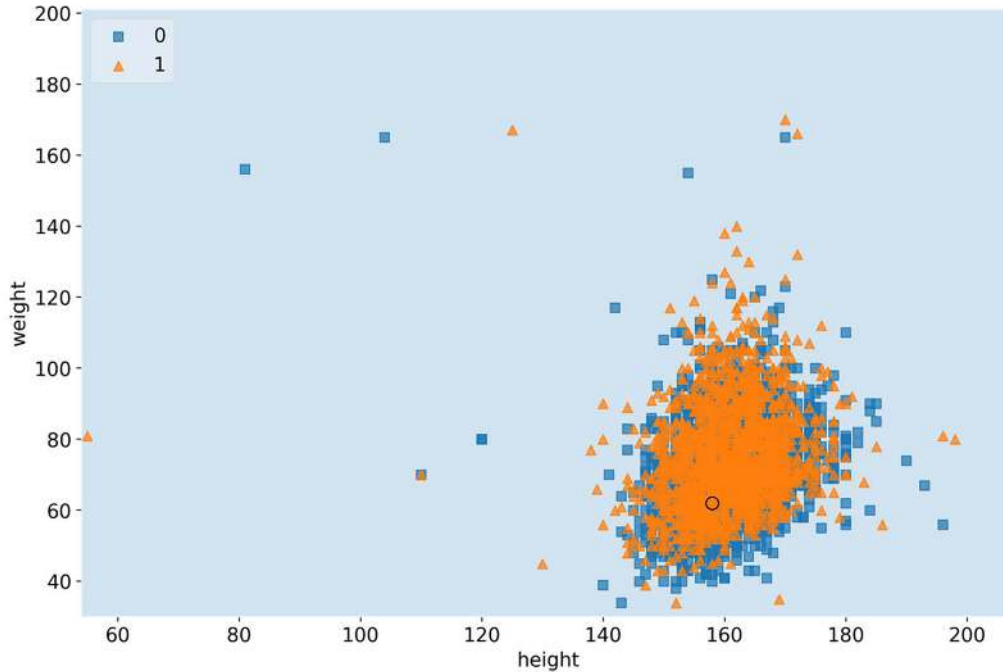
Figure 2.3: The decision regions for weight and height, with test case #2872

No decision boundary was ascertained in *Figure 2.3* because if all other variables are held constant (at a less risky value), no `height` and `weight` combination is enough to predict CVD. However, we can tell that there is a pattern for the orange triangles, mostly located in one ovular area. This provides exciting insight that even though we expect `weight` to increase when `height` increases, the concept of an inherently unhealthy `weight` value is not one that increases linearly with `height`.

In fact, for almost two centuries, this relationship has been mathematically understood by the name **body mass index** (**BMI**):

$$\text{BMI} = \frac{\text{weight}_{kg}}{\text{height}_m^2}$$

Before we discuss BMI further, you must consider complexity. Dimensionality aside, there are chiefly three things that introduce complexity that makes interpretation difficult:

1. Non-linearity
2. Interactivity
3. Non-monotonicity

Non-linearity

Linear equations such as $y = a + bx + cz$ are easy to understand. They are additive, so it is easy to separate and quantify the effects of each of its terms (*a*, *bx*, and *cz*) from the outcome of the model (*y*). Many model classes have linear equations incorporated in the math. These equations can both be used to fit the data to the model and describe the model.

However, there are model classes that are inherently non-linear because they introduce non-linearity in their training. Such is the case for *deep learning* models because they have non-linear activation functions such as *sigmoid*. However, logistic regression is considered a **Generalized Linear Model** (**GLM**) because it's additive. In other words, the outcome is a sum of weighted inputs and parameters. We will discuss GLMs further in *Chapter 3*, *Interpretation Challenges*.

However, even if your model is linear, the relationships between the variables may not be linear, which can lead to poor performance and interpretability. What you can do in these cases is adopt either of the following approaches:

- *Use a non-linear model class*, which will fit these non-linear feature relationships much better, possibly improving model performance. Nevertheless, as we will explore in more detail in the next chapter, this can make the model less interpretable.

- *Use domain knowledge to engineer a feature that can help "linearize" it*. For instance, if you had a feature that increased exponentially against another, you can engineer a new variable with the logarithm of that feature. In the case of our CVD prediction, we know BMI is a better way to understand weight in the company of height. Best of all, it's not an *arbitrary* made-up feature, so it's easier to interpret. We can prove this point by making a copy of the dataset, engineering the BMI feature in it, training the model with this extra feature, and performing local model interpretation. The following code snippet does just that:

```
X2 = cvd_df.drop(['cardio'], axis=1).copy()
X2["bmi"] = X2["weight"] / (X2["height"]/100)**2
```

To illustrate this new feature, let's plot `bmi` against both `weight` and `height` using the following code:

```
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(15,4))
sns.regplot(x="weight", y="bmi", data=X2, ax=ax1)
sns.regplot(x="height", y="bmi", data=X2, ax=ax2)
sns.regplot(x="height", y="weight", data=X2, ax=ax3)
plt.subplots_adjust(top = 1, bottom=0, hspace=0.2, wspace=0.3)
plt.show()
```

Figure 2.4 is produced with the preceding code:
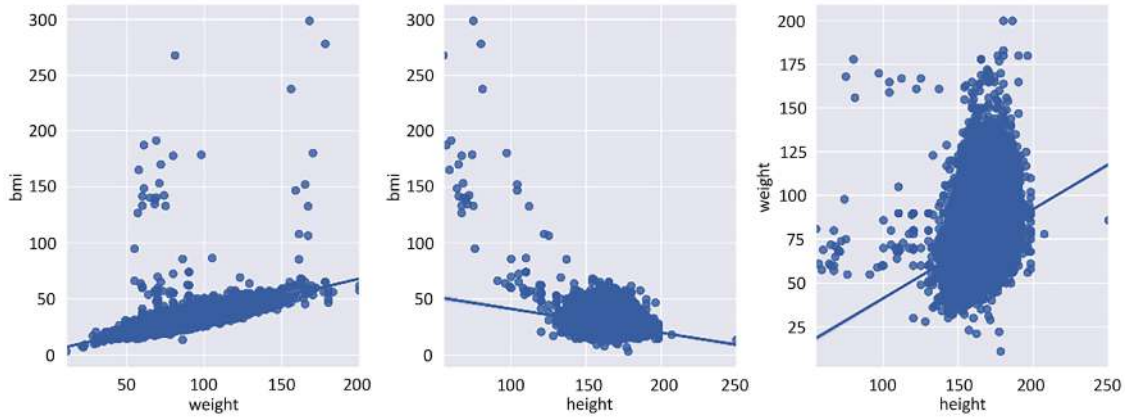
Figure 2.4: Bivariate comparison between weight, height, and bmi

As you can appreciate by the plots in *Figure 2.4*, there is a more definite linear relationship between `bmi` and `weight` than between `height` and `weight` and, even, between `bmi` and `height`.

Let's fit the new model with the extra feature using the following code snippet:

```
X2 = X2.drop(['weight','height'], axis=1)
X2_train, X2_test,__,_ = train_test_split(
  X2, y, test_size=0.15, random_state=9)
log_model2 = sm.Logit(y_train, sm.add_constant(X2_train))
log_result2 = log_model2.fit()
```

Now, let's see whether test case #2872 is in the positive decision region when comparing `ap_hi` to `bmi` if we keep `age` constant at `60`:

```
filler_feature_values2 = {
    "age": 60, "gender": 1, "ap_hi": 110,
    "ap_lo": 70, "cholesterol": 1, "gluc": 1,
    "smoke": 0, "alco":0, "active":1, "bmi":20
}

X2_highlight = np.reshape(
    np.concatenate(([1],X2_test.iloc[2872].to_numpy())), (1, 11)
)
fig, ax = plt.subplots(1,1, figsize=(12,8))

mldatasets.create_decision_plot(
    X2_test, y_test, log_result2,
    ["ap_hi", "bmi"], None, X2_highlight,
```

```
        filler_feature_values2, ax=ax)
    plt.show()
```

The preceding code plots decision regions in *Figure 2.5*:
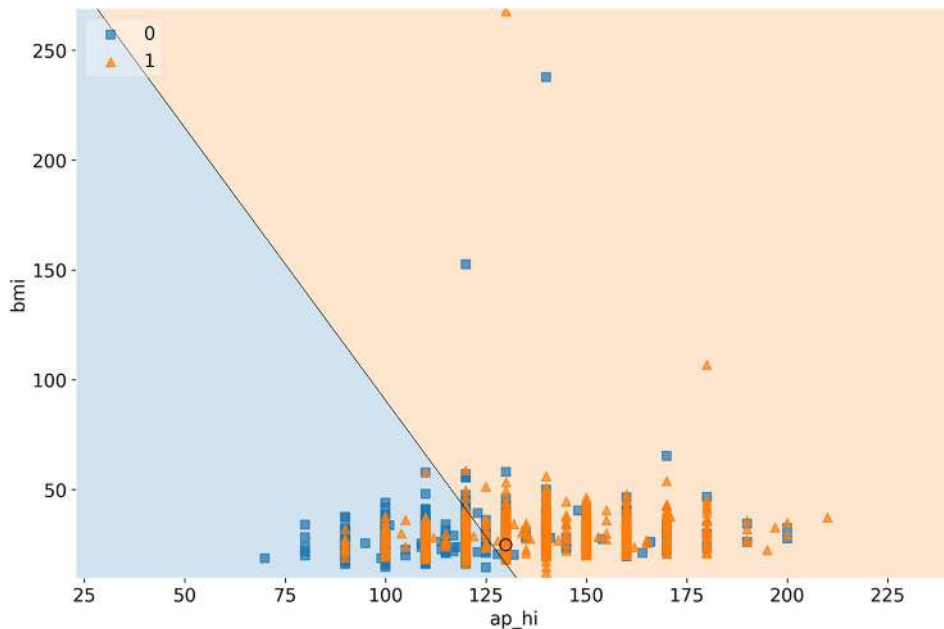
Figure 2.5: The decision regions for ap_hi and bmi, with test case #2872

Figure 2.5 shows that controlling for `age`, `ap_hi`, and `bmi` can help explain the positive prediction for CVD because the circle is in the positive decision region. Please note that there are some likely anomalous `bmi` outliers (the highest BMI ever recorded was 204), so there are probably some incorrect weights or heights in the dataset.

WHAT'S THE PROBLEM WITH OUTLIERS?

Outliers can be **influential** or **high leverage** and, therefore, affect the model when trained with these included. Even if they don't, they can make interpretation more difficult. If they are **anomalous**, then you should remove them, as we did with blood pressure at the beginning of this chapter. And sometimes, they can hide in plain sight because they are only perceived as *anomalous* in the context of other features. In any case, there are practical reasons why outliers are problematic, such as making plots like the preceding one "zoom out" to be able to fit them while not letting you appreciate the decision boundary where it matters. And there are also more profound reasons, such as losing trust in the data, thereby tainting trust in the models that were trained on that data, or making the model perform worse. This sort of problem is to be expected with real-world data. Even though we haven't done it in this chapter for the sake of expediency, it's essential to begin every project by thoroughly exploring the data, treating missing values and outliers, and doing other data housekeeping tasks.

Interactivity

When we created `bmi`, we didn't only linearize a non-linear relationship, but we also created interactions between two features. `bmi` is, therefore, an **interaction feature**, but this was informed by domain knowledge. However, many model classes do this automatically by permutating all kinds of operations between features. After all, features have *latent* relationships between one another, much like `height` and `width`, and `ap_hi` and `ap_lo`. Therefore, automating the process of looking for them is not always a bad thing. In fact, it can even be absolutely necessary. This is the case for many deep learning problems where the data is unstructured and, therefore, part of the task of training the model is looking for the latent relationships to make sense of it.

However, for structured data, even though interactions can be significant for model performance, they can hurt interpretability by adding potentially unnecessary complexity to the model and also finding latent relationships that *don't mean anything* (which is called a **spurious relationship** or **correlation**).

Non-monotonicity

Often, a variable has a meaningful and consistent relationship between a feature and the target variable. So, we know that as `age` increases, the risk of CVD (`cardio`) must increase. There is no point at which you reach a certain age and this risk drops. Maybe the risk slows down, but it does not drop. We call this **monotonicity**, and functions that are *monotonic* are either always increasing or decreasing throughout their entire domain.

Please note that **all** linear relationships are monotonic, but not all monotonic relationships are necessarily linear. This is because they don't have to be a straight line. A common problem in machine learning is that a model doesn't know about a monotonic relationship that we expect because of our domain expertise. Then, because of noise and omissions in the data, the model is trained in such a way in which there are ups and downs where you don't expect them.

Let's propose a hypothetical example. Let's imagine that due to a lack of availability of data for 57–60-year-olds, and because the few cases we did have for this range were negative for CVD, the model could learn that this is where you would expect a drop in CVD risk. Some model classes are inherently monotonic, such as logistic regression, so they can't have this problem, but many others do. We will examine this in more detail in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*:
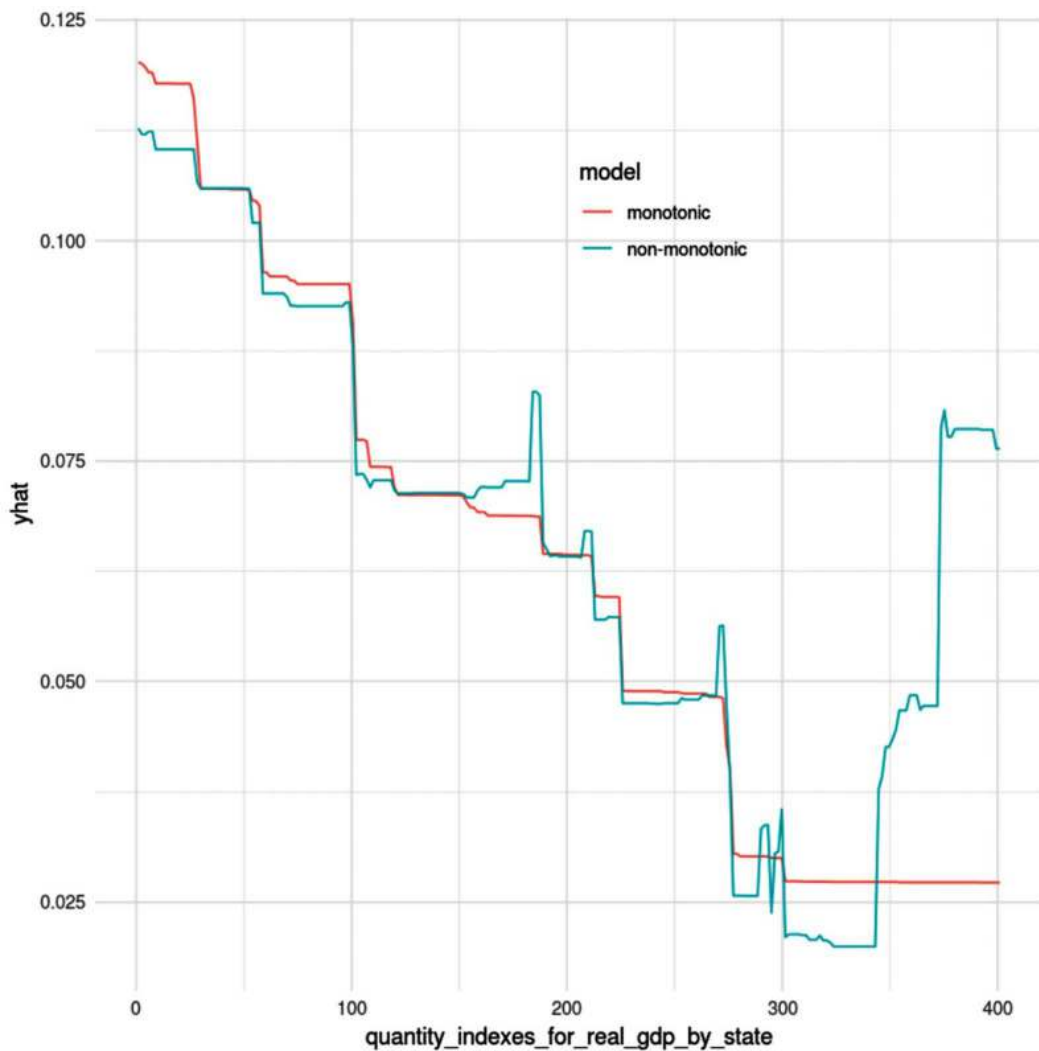
Figure 2.6: A partial dependence plot between a target variable (yhat) and a predictor with monotonic and non-monotonic models

Figure 2.6 is what is called a **Partial Dependence Plot** (**PDP**), from an unrelated example. PDPs are a concept we will study in further detail in *Chapter 4, Global Model-Agnostic Interpretation Methods,* but what is important to grasp from it is that the prediction yhat is supposed to decrease as the feature quantity_indexes_for_real_gdp_by_state increases. As you can tell by the lines, in the monotonic model, it consistently decreases, but in the non-monotonic one, it has jagged peaks as it decreases, and then increases at the very end.

Mission accomplished

The first part of the mission was to understand risk factors for cardiovascular disease, and you've determined that the top four risk factors are systolic blood pressure (`ap_hi`), `age`, `cholesterol`, and `weight` according to the logistic regression model, of which only `age` is non-modifiable. However, you also realized that systolic blood pressure (`ap_hi`) is not as meaningful on its own since it relies on diastolic blood pressure (`ap_lo`) for interpretation. The same goes for `weight` and `height`. We learned that the interaction of features plays a crucial role in interpretation, and so does their relationship with each other and the target variable, whether linear or monotonic. Furthermore, the data is only a representation of the truth, which can be wrong. After all, we found *anomalies* that, left unchecked, can bias our model.

Another source of bias is how the data was collected. After all, you can wonder why the model's top features were all objective and examination features. Why isn't smoking or drinking a larger factor? To verify whether there was *sample bias* involved, you would have to compare with other more trust-worthy datasets to check whether your dataset is underrepresenting drinkers and smokers. Or maybe the bias was introduced by the question that asked whether they smoked now, and not whether they had ever smoked for an extended period.

Another type of bias that we could address is *exclusion bias*—our data might be missing information that explains the truth that the model is trying to depict. For instance, we know through medical research that blood pressure issues such as isolated systolic hypertension, which increases CVD risk, are caused by underlying conditions such as diabetes, hyperthyroidism, arterial stiffness, and obesity, to name a few. The only one of these conditions that we can derive from the data is obesity and not the other ones. If we want to be able to interpret a model's predictions well, we need to have all relevant features. Otherwise, there will be gaps we cannot explain. Maybe once we add them, they won't make much of a difference, but that's what the methods we will learn in *Chapter 10*, *Feature Selection and Engineering for Interpretability*, are for.

The second part of the mission was to be able to interpret individual model predictions. We can do this well enough by plotting decision regions. It's a simple method, but it has many limitations, especially in situations where there are more than a handful of features, and they tend to interact a lot with each other. *Chapter 5*, *Local Model-Agnostic Interpretation Methods*, and *Chapter 6*, *Anchors and Counterfactual Explanations*, will cover local interpretation methods in more detail. However, the decision region plot method helps illustrate many of the concepts surrounding decision boundaries, which we will discuss in those chapters.

Summary

In this chapter, we covered two model interpretation methods: feature importance and decision boundaries. We also learned about model interpretation method types and scopes and the three elements that impact interpretability in machine learning. We will keep mentioning these fundamental concepts in subsequent chapters. For a machine learning practitioner, it is paramount to be able to spot them so that we can know what tools to leverage to overcome interpretation challenges. In the next chapter, we will dive deeper into this topic.

Further reading

- Molnar, Christoph. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2019: `https://christophm.github.io/interpretable-ml-book/`

- *Mlextend Documentation. Plotting Decision Regions*: `http://rasbt.github.io/mlxtend/user_guide/plotting/plot_decision_regions/`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

3

Interpretation Challenges

In this chapter, we will discuss the traditional methods used for machine learning interpretation for both regression and classification. This includes model performance evaluation methods such as RMSE, R-squared, AUC, ROC curves, and the many metrics derived from confusion matrices. We will then examine the limitations of these performance metrics and explain what exactly makes "white-box" models intrinsically interpretable and why we cannot always use white-box models. To answer these questions, we'll consider the trade-off between prediction performance and model interpretability. Finally, we will discover some new "glass-box" models such as **Explainable Boosting Machines** (**EBMs**) and GAMI-Net that attempt to not compromise on this trade-off between predictive performance and interpretability.

The following are the main topics that will be covered in this chapter:

- Reviewing traditional model interpretation methods
- Understanding the limitations of traditional model interpretation methods
- Studying intrinsically interpretable (white-box) models
- Recognizing the trade-off between performance and interpretability
- Discovering newer interpretable (glass-box) models

Technical requirements

From *Chapter 2*, *Key Concepts of Interpretability*, onward, we are using a custom `mldatasets` library to load our datasets. Instructions on how to install this library can be found in the *Preface*. In addition to `mldatasets`, this chapter's examples also use the `pandas`, `numpy`, `sklearn`, `rulefit`, `interpret`, `statsmodels`, `matplotlib`, and `gaminet` libraries.

The code for this chapter is located here: `packt.link/swCyB`.

The mission

Picture yourself, a data science consultant, in a conference room in Fort Worth, Texas, during early January 2019. In this conference room, executives for one of the world's largest airlines, **American Airlines** (**AA**), are briefing you on their **On-Time Performance** (**OTP**). OTP is a widely accepted **Key Performance Indicator** (**KPI**) for flight punctuality. It is measured as the percentage of flights that arrived within 15 minutes of the scheduled arrival. It turns out that AA has achieved an OTP of just over 80% for 3 years in a row, which is acceptable, and a significant improvement, but they are still ninth in the world and fifth in North America. To brag about it next year in their advertising, they aspire to achieve, at least, number one in North America for 2019, besting their biggest rivals.

On the financial front, it is estimated that delays cost the airline close to $2 billion, so reducing this by 25–35% to be on parity with their competitors could produce sizable savings. And it is estimated that it costs passengers just as much due to tens of millions of lost hours. A reduction in delays would result in happier customers, which could lead to an increase in ticket sales.

Your task is to create models that can accurately predict delays for domestic flights only. What they hope to gain from the models is the following:

- To understand what factors impacted domestic arrival delays the most in 2018
- To anticipate a delay caused by the airline in midair with enough accuracy to mitigate some of these factors in 2019

But not all delays are made equal. The **International Air Transport Association** (**IATA**) has over 80 delay codes ranging from 14 (*oversales booking errors*) to 75 (*de-icing of aircraft, removal of ice/snow, frost prevention*). Some are preventable, and others unavoidable.

The airline executives told you that the airline is not, for now, interested in predicting delays caused by events out of their control, such as extreme weather, security events, and air traffic control issues. They are also not interested in delays caused by late arrivals from previous flights using the same aircraft because this was not the root cause. Nevertheless, they would like to know the effect of a busy hub on avoidable delays even if this has to do with congestion because, after all, perhaps there's something they can do with flight scheduling or flight speed, or even gate selection. And while they understand that international flights occasionally impact domestic flights, they hope to tackle the sizeable local market first.

Executives have provided you with a dataset from the United States Department of Transportation *Bureau of Transportation Statistics* with all 2018 AA domestic flights.

The approach

Upon careful consideration, you have decided to approach this both as a regression problem and a classification problem. Therefore, you will produce models that predict minutes delayed as well as models that classify whether flights were delayed by more than 15 minutes. For interpretation, using both will enable you to use a wider variety of methods and expand your interpretation accordingly. So we will approach this example by taking the following steps:

1. Predicting minutes delayed with various regression methods
2. Classifying flights as delayed or not delayed with various classification methods

These steps in the *Reviewing traditional model interpretation methods* section are followed by conclusions spread out in the rest of the sections of this chapter.

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/03/FlightDelays.ipynb`.

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `sklearn` (scikit-learn), `rulefit`, `statsmodels`, `interpret`, `tf`, and `gaminet` to fit models and calculate performance metrics
- `matplotlib` to create visualizations

Load these libraries as seen in the following snippet:

```python
import math
import mldatasets
import pandas as pd
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler,\
                                   MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics, linear_model, tree, naive_bayes,\
                    neighbors, ensemble, neural_network, svm
from rulefit import RuleFit
import statsmodels.api as sm
from interpret.glassbox import ExplainableBoostingClassifier
from interpret import show
from interpret.perf import ROC
import tensorflow as tf
from gaminet import GAMINet
from gaminet.utils import plot_trajectory, plot_regularization,\
                    local_visualize, global_visualize_density,\
                    feature_importance_visualize
import matplotlib.pyplot as plt
```

Understanding and preparing the data

We then load the data as shown:

```
aad18_df = mldatasets.load("aa-domestic-delays-2018")
```

There should be nearly 900,000 records and 23 columns. We can take a peek at what was loaded like this:

```
print(aad18_df.info())
```

The following is the output:

```
RangeIndex: 899527 entries, 0 to 899526
Data columns (total 23 columns):
FL_NUM                 899527 non-null int64
ORIGIN                 899527 non-null object
DEST                   899527 non-null object
PLANNED_DEP_DATETIME   899527 non-null object
CRS_DEP_TIME           899527 non-null int64
DEP_TIME               899527 non-null float64
DEP_DELAY              899527 non-null float64
DEP_AFPH               899527 non-null float64
DEP_RFPH               899527 non-null float64
TAXI_OUT               899527 non-null float64
WHEELS_OFF             899527 non-null float64
    :          :  :    :
WEATHER_DELAY          899527 non-null float64
NAS_DELAY              899527 non-null float64
SECURITY_DELAY         899527 non-null float64
LATE_AIRCRAFT_DELAY    899527 non-null float64
dtypes: float64(17), int64(3), object(3)
```

Everything seems to be in order because all columns are there and there are no `null` values.

The data dictionary

Let's examine the data dictionary.

General features are as follows:

- `FL_NUM`: Flight number.
- `ORIGIN`: Starting airport code (IATA).
- `DEST`: Destination airport code (IATA).

Departure features are as follows:

- `PLANNED_DEP_DATETIME`: The planned date and time of the flight.
- `CRS_DEP_TIME`: The planned departure time.

- `DEP_TIME`: The actual departure time.
- `DEP_AFPH`: The number of actual flights per hour occurring during the interval in between the planned and actual departure from the origin airport (factoring in 30 minutes of padding). The feature tells you how busy the origin airport was during takeoff.
- `DEP_RFPH`: The departure relative flights per hour is the ratio of actual flights per hour over the median number of flights per hour that occur at the origin airport at that time of day, day of the week, and month of the year. The feature tells you how *relatively* busy the origin airport was during takeoff.
- `TAXI_OUT`: The time duration elapsed between the departure from the origin airport gate and wheels off.
- `WHEELS_OFF`: The point in time that the aircraft's wheels leave the ground.

In-flight features are as follows:

- `CRS_ELAPSED_TIME`: The planned amount of time needed for the flight trip.
- `PCT_ELAPSED_TIME`: The ratio of actual flight time over planned flight time to gauge the plane's relative speed.
- `DISTANCE`: The distance between two airports.

Arrival features are as follows:

- `CRS_ARR_TIME`: The planned arrival time.
- `ARR_AFPH`: The number of actual flights per hour occurring during the interval between the planned and actual arrival time at the destination airport (factoring in 30 minutes of padding). The feature tells you how busy the destination airport was during landing.
- `ARR_RFPH`: The arrival relative flights per hour is the ratio of actual flights per hour over the median number of flights per hour that occur at the destination airport at that time of day, day of the week, and month of the year. The feature tells you how *relatively* busy the destination airport was during landing.

Delay features are as follows:

- `DEP_DELAY`: The total delay on departure in minutes.
- `ARR_DELAY`: The total delay on arrival in minutes can be subdivided into any or all of the following:

    a. `CARRIER_DELAY`: The delay in minutes caused by circumstances within the airline's control (for example, maintenance or crew problems, aircraft cleaning, baggage loading, fueling, and so on).

    b. `WEATHER_DELAY`: The delay in minutes caused by significant meteorological conditions (actual or forecasted).

    c. `NAS_DELAY`: The delay in minutes mandated by a national aviation system such as non-extreme weather conditions, airport operations, heavy traffic volume, and air traffic control.

d.   `SECURITY_DELAY`: The delay in minutes caused by the evacuation of a terminal or concourse, re-boarding of an aircraft because of a security breach, faulty screening equipment, or long lines above 29 minutes in screening areas.

e.   `LATE_AIRCRAFT_DELAY`: The delay in minutes caused by a previous flight with the same aircraft that arrived late.

Data preparation

For starters, `PLANNED_DEP_DATETIME` must be a datetime data type:

```
aad18_df['PLANNED_DEP_DATETIME'] =\
pd.to_datetime(aad18_df['PLANNED_DEP_DATETIME'])
```

The exact day and time of a flight don't matter, but maybe the month and day of the week do because of weather and seasonal patterns that can only be appreciated at this level of granularity. Also, the executives mentioned weekends and winters being especially bad for delays. Therefore, we will create features for the month and day of the week:

```
aad18_df['DEP_MONTH'] = aad18_df['PLANNED_DEP_DATETIME'].dt.month
aad18_df['DEP_DOW'] = aad18_df['PLANNED_DEP_DATETIME'].dt.dayofweek
```

We don't need the `PLANNED_DEP_DATETIME` column so let's drop it like this:

```
aad18_df = aad18_df.drop(['PLANNED_DEP_DATETIME'], axis=1)
```

It is essential to record whether the arrival or destination airport is a hub. AA, in 2019, had 10 hubs: Charlotte, Chicago–O'Hare, Dallas/Fort Worth, Los Angeles, Miami, New York–JFK, New York–LaGuardia, Philadelphia, Phoenix–Sky Harbor, and Washington–National. Therefore, we can encode which `ORIGIN` and `DEST` airports are AA hubs using their IATA codes, and get rid of columns with codes since they are too specific (`FL_NUM`, `ORIGIN`, and `DEST`):

```
#Create list with 10 hubs (with their IATA codes)
hubs = ['CLT', 'ORD', 'DFW', 'LAX', 'MIA', 'JFK', 'LGA', 'PHL',
        'PHX', 'DCA']
#Boolean series for if ORIGIN or DEST are hubsis_origin_hub =
aad18_df['ORIGIN'].isin(hubs)
is_dest_hub = aad18_df['DEST'].isin(hubs)
#Use boolean series to set ORIGIN_HUB and DEST_HUB
aad18_df['ORIGIN_HUB'] = 0
aad18_df.loc[is_origin_hub, 'ORIGIN_HUB'] = 1
aad18_df['DEST_HUB'] = 0
aad18_df.loc[is_dest_hub, 'DEST_HUB'] = 1
#Drop columns with codes
aad18_df = aad18_df.drop(['FL_NUM', 'ORIGIN', 'DEST'], axis=1)
```

After all these operations, we have a fair number of useful features, but we are yet to determine the target feature. There are two columns that could serve this purpose. We have `ARR_DELAY`, which is the total number of minutes delayed regardless of the reason, and then there's `CARRIER_DELAY`, which is just the total number of those minutes that can be attributed to the airline. For instance, look at the following sample of flights delayed over 15 minutes (which is considered late according to the airline's definition):

```
aad18_df.loc[aad18_df['ARR_DELAY'] > 15,\
['ARR_DELAY','CARRIER_DELAY']].head(10)
```

The preceding code outputs *Figure 3.1*:

	ARR_DELAY	CARRIER_DELAY
8	168.000000	136.000000
16	20.000000	5.000000
18	242.000000	242.000000
19	62.000000	62.000000
22	19.000000	19.000000
26	26.000000	0.000000
29	77.000000	77.000000
32	19.000000	19.000000
33	18.000000	1.000000
40	36.000000	16.000000

Figure 3.1: Sample observations with arrival delays over 15 minutes

Of all the delays in *Figure 3.1*, one of them (#26) wasn't at all the responsibility of the airline because only 0 minutes could be attributed to the airline. Four of them were partially the responsibility of the airline (#8, #16, #33, and #40), two of which were over 15 minutes late due to the airline (#8 and #40). The rest of them were entirely the airline's fault. We can tell that although the total delay is useful information, the airline executives were only interested in delays caused by the airline so `ARR_DELAY` can be discarded. Furthermore, there's another more important reason it should be discarded, and it's that if the task at hand is to predict a delay, we cannot use pretty much the very same delay (minus the portions not due to the airline) to predict it. For this very same reason, it is best to remove `ARR_DELAY`:

```
aad18_df = aad18_df.drop(['ARR_DELAY'], axis=1)
```

Finally, we can put the target feature alone as y and all the rest as X. After this, we split y and X into train and test datasets. Please note that the target feature (y) stays the same for regression, so we split it into y_train_reg and y_test_reg. However, for classification, we must make binary versions of these labels denoting whether it's more than 15 minutes late or not, called y_train_class and y_test_class. Please note that we are setting a fixed random_state for reproducibility:

```python
rand = 9
np.random.seed(rand)
y = aad18_df['CARRIER_DELAY']
X = aad18_df.drop(['CARRIER_DELAY'], axis=1).copy()
X_train, X_test, y_train_reg, y_test_reg = train_test_split(
    X, y, test_size=0.15, random_state=rand
)
y_train_class = y_train_reg.apply(lambda x: 1 if x > 15 else 0)
y_test_class = y_test_reg.apply(lambda x: 1 if x > 15 else 0)
```

To examine how linearly correlated the features are to the target CARRIER_DELAY, we can compute Pearson's correlation coefficient, turn coefficients to absolute values (because we aren't interested in whether they are positively or negatively correlated), and sort them in descending order:

```python
corr = aad18_df.corr()
abs(corr['CARRIER_DELAY']).sort_values(ascending=False)
```

As you can tell from the output, only one feature (DEP_DELAY) is highly correlated. The others aren't:

```
CARRIER_DELAY         1.000000
DEP_DELAY             0.703935
ARR_RFPH              0.101742
LATE_AIRCRAFT_DELAY   0.083166
DEP_RFPH              0.058659
ARR_AFPH              0.035135
DEP_TIME              0.030941
NAS_DELAY             0.026792
    :           :
WEATHER_DELAY         0.003002
SECURITY_DELAY        0.000460
```

However, this is only *linearly* correlated and on a one-by-one basis. It doesn't mean that they don't have a non-linear relationship, or that several features interacting together wouldn't impact the target. In the next section, we will discuss this further.

Reviewing traditional model interpretation methods

To explore as many model classes and interpretation methods as possible, we will fit the data into regression and classification models.

Predicting minutes delayed with various regression methods

To compare and contrast regression methods, we will first create a dictionary named `reg_models`. Each model is its own dictionary and the function that creates it is the `model` attribute. This structure will be used later to neatly store the fitted model and its metrics. Model classes in this dictionary have been chosen to represent several model families and to illustrate important concepts that we will discuss later:

```python
reg_models = {
    #Generalized Linear Models (GLMs)
    'linear':{'model': linear_model.LinearRegression()},
    'linear_poly':{
        'model':make_pipeline(
            PolynomialFeatures(degree=2),
            linear_model.LinearRegression(fit_intercept=False)
        )
    },
    'linear_interact':{
        'model':make_pipeline(
            PolynomialFeatures(interaction_only=True),
            linear_model.LinearRegression(fit_intercept=False)
        )
    },
    'ridge':{
        'model': linear_model.RidgeCV(
            alphas=[1e-3, 1e-2, 1e-1, 1])
    },
    #Trees
    'decision_tree':{
        'model': tree.DecisionTreeRegressor(
            max_depth=7, random_state=rand
        )
    },
    #RuleFit
    'rulefit':{
        'model': RuleFit(
            max_rules=150,
            rfmode='regress',
            random_state=rand
        )
    },
    #Nearest Neighbors
```

```
    'knn':{'model': neighbors.KNeighborsRegressor(n_neighbors=7)},
    #Ensemble Methods
    'random_forest':{
        'model':ensemble.RandomForestRegressor(
            max_depth=7, random_state=rand)
    },
    #Neural Networks
    'mlp':{
        'model':neural_network.MLPRegressor(
            hidden_layer_sizes=(21,),
            max_iter=500,
            early_stopping=True,
            random_state=rand
        )
    }
}
```

Before we start fitting the data to these models, we will briefly explain them one by one:

- `linear`: **Linear regression** was the first model class we discussed. For better or for worse, it makes several assumptions about the data. Chief among them is the assumption that the prediction must be a linear combination of *X* features. This, naturally, limits the capacity to discover non-linear relationships and interactions among the features.

- `linear_poly`: **Polynomial regression** extends linear regression by adding polynomial features. In this case, as indicated by `degree=2`, the polynomial degree is two, so it's quadratic. This means, in addition to having all features in their monomial form (for example, `DEP_FPH`), it also has them in a quadratic form (for example, `DEP_FPH²`), plus the many interaction terms for all of the 21 features. In other words, for `DEP_FPH`, there would be interaction terms such as `DEP_FPH × DISTANCE`, `DEP_FPH × DELAY`, and so on for the rest of the features.

- `linear_interact`: This is just like the **polynomial regression** model but without the quadratic terms – in other words, only the interactions, as `interaction_only=True` would suggest. It's useful because there is no reason to believe any of our features have a relationship that is better fitted with quadratic terms. Still, perhaps it's the interaction with other features that makes an impact.

- `ridge`: **Ridge regression** is a variation of linear regression. However, even though the method behind linear regression, called **ordinary least squares** (**OLS**), does a pretty good job of reducing the error and fitting the model to the features, it does it without considering **overfitting**. The problem here is that OLS treats all features equally, so the model becomes more complex as each variable is added. As the word *overfitting* suggests, the resulting model fits the training data too well, resulting in the lowest bias but the highest variance. There's a sweet spot in this **trade-off between bias and variance**, and one way of getting to this spot is by reducing the complexity added by the introduction of too many features. Linear regression is not equipped to do so on its own.

This is where ridge regression comes along, with our friend **regularization**. It does this by shrinking coefficients that don't contribute to the outcome with a penalty term called the **L2 norm**. It penalizes complexity, thus constraining the algorithm from overfitting. In this example, we use a cross-validated version of `ridge` (`RidgeCV`) that tests several regularization strengths (`alphas`).

- `decision_tree`: A **decision tree** is precisely as the name suggests. Imagine a tree-like structure where at every point that branches subdivide to form more branches, there is a "test" performed on a feature, partitioning the datasets into each branch. When branches stop subdividing, they become leaves, and at every leaf, there's *a decision*, be it to assign a *class* for classification or a fixed value for regression. We are limiting this tree to `max_depth=7` to prevent overfitting because the larger the tree, the better it will fit our training data, and the less likely the tree will generalize to non-training data.

- `rule_fit`: **RuleFit** is a regularized linear regression expanded to include feature interactions in the form of rules. The rules are formed by traversing a decision tree, except it discards the leaves and keeps the feature interactions found traversing the branches toward these leaves. It uses **LASSO Regression**, which, like ridge, uses regularization, but instead of using the **L2 norm**, it uses the **L1 norm**. The result is that useless features end up with a coefficient of zero and do not just converge to zero, as they do with L2, which makes it easy for the algorithm to filter them out. We are limiting the rules to 150 (`max_rules=150`) and the attribute `rfmode='regress'` tells RuleFit that this is a regression problem, since it can also be used for classification. Unlike all other models used here, this isn't a scikit-learn one but was created by Christoph Molnar adapting a paper called *Predictive learning via rule ensembles*.

- `knn`: **k-Nearest Neighbors** (**k-NN**) is a simple method based on the *locality* assumption, which is that data points that are close to each other are similar. In other words, they must have similar predicted values, and, in practice, this isn't a bad guess, so it takes data points nearest to the point you want to predict and derives a prediction based on that. In this case, `n_neighbors=7` so k = 7. It's an **instance-based machine learning model**, also known as a **lazy learner** because it simply stores the training data. During inference, it employs training data to calculate the similarity with points and generate a prediction based on that. This is opposed to what model-based machine learning techniques, or **eager learners**, do, which is to use training data to learn formulas, parameters, coefficients, or bias/weights, which they then leverages to make a prediction during inference.

- `random_forest`: Imagine not one but hundreds of decision trees trained on random combinations of the features and random samples of data. **Random forest** takes an average of these randomly generated decision trees to create the best tree. This concept of training less effective models in parallel and combining them using an averaging process is called **bagging**. It is an **ensemble** method because it combines more than one model (usually called **weak learners**) into a **strong learner**. In addition to *bagging*, there are two other ensemble techniques, called **boosting** and **stacking**. For bagging deeper, trees are better because they reduce variance, so this is why we are using `max_depth=7`.

- **mlp: A multi-layer perceptron** is a "vanilla" feedforward (sequential) neural network, so it uses non-linear activation functions (`MLPRegressor` uses *ReLU* by default), stochastic gradient descent, and backpropagation. In this case, we are using 21 neurons in the first and only hidden layer, hence `hidden_layer_sizes=(21,)`, running training for 500 epochs (`max_iter=500`), and terminating training when the validation score is not improving (`early_stopping=True`).

If you are unfamiliar with some of these models, don't fret! We will cover them in more detail later in this chapter and the book. Also, please note that some of these models have a random process somewhere. To ensure reproducibility, we have set `random_state`. It is best to always set this; otherwise, it will randomly set it every single time, which will make your results hard to reproduce.

Now, let's iterate over our dictionary of models (`reg_models`), fit them to the training data, and predict and compute two metrics based on the quality of these predictions. We'll then save the fitted model, test predictions, and metrics in the dictionary for later use. Note that `rulefit` only accepts `numpy` arrays, so we can't `fit` it in the same way. Also, note that `rulefit` and `mlp` take longer than the rest to train, so this can take a few minutes to run:

```
for model_name in reg_models.keys():
    if model_name != 'rulefit':
        fitted_model = reg_models[model_name]\
        ['model'].fit(X_train, y_train_reg)
    else :
        fitted_model = reg_models[model_name]['model'].\
        fit(X_train.values, y_train_reg.values, X_test.columns
        )
        y_train_pred = fitted_model.predict(X_train.values)
        y_test_pred = fitted_model.predict(X_test.values)

    reg_models[model_name]['fitted'] = fitted_model
    reg_models[model_name]['preds'] = y_test_pred
    reg_models[model_name]['RMSE_train'] =\
    math.sqrt(
        metrics.mean_squared_error(y_train_reg, y_train_pred)
    )
    reg_models[model_name]['RMSE_test'] =\
    math.sqrt(metrics.mean_squared_error(y_test_reg, y_test_pred)
    )
    reg_models[model_name]['R2_test'] =\
    metrics.r2_score(y_test_reg, y_test_pred)
```

We can now convert the dictionary to a `DataFrame` and display the metrics in a sorted and color-coded fashion:

```
reg_metrics = pd.DataFrame.from_dict(
    reg_models, 'index')[['RMSE_train', 'RMSE_test', 'R2_test']]
]
```

```
reg_metrics.sort_values(by='RMSE_test').style.format(
    {'RMSE_train': '{:.2f}', 'RMSE_test': '{:.2f}',
     'R2_test': '{:.3f}'}
).background_gradient(
    cmap='viridis_r', low=0.1, high=1,
    subset=['RMSE_train', 'RMSE_test']
).background_gradient(
    cmap='plasma', low=0.3, high=1, subset=['R2_test']
)
```

The preceding code outputs *Figure 3.2*. Please note that color-coding doesn't work in all Jupyter Notebook implementations:

	RMSE_train	RMSE_test	R2_test
mlp	3.24	3.31	0.987
random_forest	5.14	6.09	0.956
linear_poly	6.21	6.34	0.952
linear_interact	6.45	6.56	0.949
decision_tree	6.54	7.46	0.934
linear	7.82	7.88	0.926
ridge	7.83	7.90	0.926
knn	7.36	9.26	0.898
rulefit	9.17	9.31	0.897

Figure 3.2: Regression metrics for our models

To interpret the metrics in *Figure 3.2*, we ought to first understand what they mean, both in general and in the context of this regression exercise:

- **RMSE: Root Mean Square Error** is defined as the standard deviation of the residuals. It's the square root of the squared residuals divided by the number of observations – in this case, flights. It tells you, on average, how far apart the predictions are from the actuals, and as you can probably tell from the color-coding, less is better because you want your predictions to be as close as possible to the actuals in the *test* (**hold-out**) dataset. We have also included this metric for the **train** dataset to see how well it's generalizing. You expect the test error to be higher than the training error, but not by much. If it is, like it is for random_forest, you need to tune some of the parameters to reduce overfitting. In this case, reducing the trees' maximum depth, increasing the number of trees (also called **estimators**), and reducing the maximum number of features to use should do the trick. On the other hand, with knn, you can adjust the number of neighbors, but it is expected, because of its **lazy learner** nature, to overperform on the training data.

In any case, these numbers are pretty good because even our worst performing model is below a test RMSE of 10 minutes, and about half of them have a test RMSE of less than 7.5, quite possibly predicting a delay effectively, on average, since the threshold for a delay is 15 minutes.

Note that `linear_poly` is the second and `linear_interact` is the fourth most performant model, significantly ahead of `linear`, suggesting that non-linearity and interactivity are important factors to produce better predictive performance.

- **R^2: R-squared** is also known as the **coefficient of determination**. It's defined as the proportion of the variance in the y (outcome) target that can be explained by the X (predictors) features in the model. It answers the question of what proportion of the model variability is explainable? And as you can probably tell from the color-coding, more is better. And our models appear to include significant X features, as evidenced by our *Pearson's correlation coefficients*. So if this R^2 value was low, perhaps adding additional features would help, such as flight logs, terminal conditions, and even those things airline executives said they weren't interested in exploring right now, such as *knock-off* effects and international flights. These could fill in the gaps in the unexplained variance.

Let's see if we can get good metrics with classification.

Classifying flights as delayed or not delayed with various classification methods

Just as we did with regression, to compare and contrast classification methods, we will first create a dictionary for them named `class_models`. Each model is its own dictionary and the function that creates it is the `model` attribute. This structure will be used later to store the fitted model and its metrics. Model classes in this dictionary have been chosen to represent several model families and to illustrate important concepts that we will discuss later. Some of these will look familiar because they are the same methods used in regression but applied to classification:

```python
class_models = {
    #Generalized Linear Models (GLMs)
    'logistic':{'model': linear_model.LogisticRegression()},
    'ridge':{
        'model': linear_model.RidgeClassifierCV(
            cv=5, alphas=[1e-3, 1e-2, 1e-1, 1],
            class_weight='balanced'
        )
    },
    #Tree
    'decision_tree':{
        'model': tree. DecisionTreeClassifier(max_depth=7,
                                              random_state=rand)
    },
```

```
    #Nearest Neighbors
    'knn':{'model': neighbors.KNeighborsClassifier(n_neighbors=7)},
    #Naive Bayes
    'naive_bayes':{'model': naive_bayes.GaussianNB()},
    #Ensemble Methods
    'gradient_boosting':{
        'model':ensemble.
        GradientBoostingClassifier(n_estimators=210)
    },
    'random_forest':{
        'model':ensemble.RandomForestClassifier(
            max_depth=11,class_weight='balanced', random_state=rand
        )
    },
    #Neural Networks
    'mlp':{
        'model': make_pipeline(
            StandardScaler(),
            neural_network.MLPClassifier(
                hidden_layer_sizes=(7,),
                max_iter=500,
                early_stopping=True,
                random_state=rand
            )
        )
    }
}
```

Before we start fitting the data to these models, we will briefly explain them one by one:

- `logistic`: **logistic regression** was introduced in *Chapter 2, Key Concepts of Interpretability*. It has many of the same pros and cons as **linear regression**. For instance, feature interactions must be added manually. Like other classification models, it returns a probability between 0 and 1, which, when closer to 1, denotes a probable match to a **positive class** while, when closer to 0, it denotes an improbable match to the **positive class**, and therefore a probable match to the **negative class**. Naturally, 0.5 is the threshold used to decide between classes, but it doesn't have to be. As we will examine later in the book, there are interpretation and performance reasons to adjust the threshold. Note that this is a binary classification problem, so we are only choosing between delayed (positive) and not delayed (negative), but this method could be extended to multi-class classification. It would then be called **multinomial classification**.

- `ridge`: **Ridge classification** leverages the same regularization technique used in **ridge regression** but applied to classification. It does this by converting the target values to -1 (for a negative class) and keeping 1 for a positive class and then performing ridge regression. At its heart, its regression in disguise will predict values between -1 and 1, and then convert them back to a 0–1 scale. Like with `RidgeCV` for regression, `RidgeClassifierCV` uses leave-one-out cross-validation, which means it first splits the data into different equal-size sets – in this case, we are using five sets (`cv=5`) – and then removes features one at a time to see how well the model performs without them, on average in all the five sets. Those features that don't make much of a difference are penalized by testing several regularization strengths (`alphas`) to find the optimal strength. As with all *regularization* techniques, the point is to discourage learning from unnecessary complexity, minimizing the impact of less salient features.

- `decision_tree`: A standard **decision tree**, such as this one, is also known as a **CART (classification and regression tree)** because it can be used for regression or classification tasks. It has the same algorithm for both tasks but functions slightly differently, like the algorithm used to decide where to "split" a branch. In this case, we are only allowing our trees to have a depth of 7.

- `knn`: **k-NN** can also be applied to classification tasks, except instead of averaging what the nearest neighbors' target features (or labels) are, it chooses the most frequent one (also known as the **mode**). We are also using a k-value of 7 for classification (`n_neighbors`).

- `naive_bayes`: **Gaussian Naïve Bayes** is part of the family of *Naïve Bayes* classifiers, which are called naïve because they make the assumption that the features are independent of each other, which is usually not the case. This dramatically impedes its capacity to predict unless the assumption is correct. It's called *Bayes* because it's based on **Bayes' theorem of conditional probabilities**, which is that the conditional probability of a class is the class probability times the feature probability given the class. *Gaussian Naïve Bayes* makes an additional assumption, which is that continuous values have a normal distribution, also known as a **Gaussian distribution**.

- `gradient_boosting`: Like **random forest**, **gradient-boosted trees** are also an ensemble method, but that leverages **boosting** instead of **bagging**. **Boosting** doesn't work in parallel but in sequence, iteratively training weak learners and incorporating their strengths into a stronger learner, while adapting another weak learner to tackle their weaknesses. Although ensembles and boosting, in particular, can be done with a model class, this method uses decision trees. We have limited the number of trees to 210 (`n_estimators=210`).

- `random_forest`: The same **random forest** as with regression except it generates classification decision trees and not regression trees.

- `mlp`: The same **multi-layer perceptron** as with regression, but the output layer, by default, uses a **logistic** function in the output layer to yield probabilities, which it then converts to 1 or 0, based on the 0.5 threshold. Another difference is that we are using seven neurons in the first and only hidden layer (`hidden_layer_sizes=(7,)`) because binary classification tends to require fewer of them to achieve an optimal result.

Please note that some of these models use balanced weights for the classes (`class_weight='balanced'`), which is very important because this happens to be an **imbalanced classification** task. By that, we mean that negative classes vastly outnumber positive classes. We can find out what this looks like for our training data:

```
print(y_train_class[y_train_class==1].shape[0] y_train_class.shape[0])
```

As you can see, the output in our training data's positive classes represents only 6% of the total. Models that account for this will achieve *more balanced* results. There are different ways of accounting for *class imbalance,* which we will discuss in further detail in *Chapter 11, Bias Mitigation and Causal Inference Methods*, but `class_weight='balanced'` applies a weight inversely proportional to class frequencies, giving the outnumbered *positive* class a leg up.

Training and evaluating the classification models

Now, let's iterate over our dictionary of models (`class_models`), fit them to the training data, and predict both probabilities and the class except for `ridge`, which doesn't output probabilities. We'll then compute five metrics based on the quality of these predictions. Lastly, we'll save the fitted model, test predictions, and metrics in the dictionary for later use. You can go get a coffee while you run the next snippet of code because `gradient_boosting` of `sklearn` takes longer than the rest to train, so this can take a few minutes to run:

```python
for model_name in class_models.keys():
    fitted_model = class_models[model_name]
    ['model'].fit(X_train,y_train_class)
    y_train_pred = fitted_model.predict(X_train.values)
    if model_name == 'ridge':
        y_test_pred = fitted_model.predict(X_test.values)
    else:
        y_test_prob = fitted_model.predict_proba(X_test.values)[:,1]
        y_test_pred = np.where(y_test_prob > 0.5, 1, 0)
     class_models[model_name]['fitted'] = fitted_model
    class_models[model_name]['probs'] = y_test_prob
    class_models[model_name]['preds'] = y_test_pred
    class_models[model_name]['Accuracy_train'] =\
    metrics.accuracy_score(y_train_class, y_train_pred
    )
    class_models[model_name]['Accuracy_test'] =\
    metrics.accuracy_score(y_test_class, y_test_pred
    )
    class_models[model_name]['Recall_train'] =\
    metrics.recall_score(y_train_class, y_train_pred
    )
    class_models[model_name]['Recall_test'] =\
    metrics.recall_score(y_test_class, y_test_pred
    )

    if model_name != 'ridge':
        class_models[model_name]['ROC_AUC_test'] =\
```

```
            metrics.roc_auc_score(y_test_class, y_test_prob)
    else:
            class_models[model_name]['ROC_AUC_test'] = np.nan


    class_models[model_name]['F1_test'] =\
    metrics.f1_score(y_test_class, y_test_pred
    )
    class_models[model_name]['MCC_test'] =\
    metrics.matthews_corrcoef(y_test_class, y_test_pred
    )
```

We can now convert the dictionary to a `DataFrame` and display the metrics in a sorted and color-coded fashion:

```
class_metrics = pd.DataFrame.from_dict(
    class_models,'index')[['Accuracy_train', 'Accuracy_test',
                            'Recall_train', 'Recall_test',
                            'ROC_AUC_test', 'F1_test', 'MCC_test']
]
class_metrics.sort_values(
    by='ROC_AUC_test', ascending=False).
    style.format(dict(zip(class_metrics.columns, ['{:.3f}']*7))
).background_gradient(
cmap='plasma', low=1, high=0.1, subset=['Accuracy_train',
                                        'Accuracy_test']
).background_gradient(
    cmap='viridis',
    low=1,
    high=0.1,
    subset=['Recall_train', 'Recall_test',
            'ROC_AUC_test', 'F1_test', 'MCC_test']
)
```

The preceding code outputs *Figure 3.3*:

	Accuracy_train	Accuracy_test	Recall_train	Recall_test	ROC_AUC_test	F1_test	MCC_test
mlp	0.998	0.999	0.987	0.989	1.000	0.988	0.987
gradient_boosting	0.992	0.992	0.893	0.894	0.999	0.929	0.926
random_forest	0.943	0.942	1.000	0.993	0.995	0.677	0.691
decision_tree	0.983	0.983	0.857	0.852	0.995	0.859	0.850
logistic	0.975	0.975	0.687	0.686	0.962	0.769	0.762
knn	0.973	0.965	0.681	0.608	0.948	0.681	0.668
naive_bayes	0.925	0.926	0.279	0.274	0.812	0.311	0.275
ridge	0.890	0.891	0.777	0.778	nan	0.467	0.464

Figure 3.3: Classification metrics for our models

To interpret the metrics in *Figure 3.3*, we ought to first understand what they mean, both in general and in the context of this classification exercise:

- **Accuracy**: Accuracy is the simplest way to measure the effectiveness of a classification task, and it's the percentage of correct predictions over all predictions. In other words, in a binary classification task, you can calculate this by adding the number of **True Positives** (**TPs**) and **True Negatives** (**TNs**) and dividing them by a tally of all predictions made. As with regression metrics, you can measure accuracy for both train and test to gauge overfitting.

- **Recall**: Even though accuracy sounds like a great metric, recall is much better in this case and the reason is you could have an accuracy of 94%, which sounds pretty good, but it turns out you are always predicting no delay! In other words, even if you get high accuracy, it is meaningless unless you are predicting accurately for the least represented class, delays. We can find this number with recall (also known as **sensitivity** or **true positive rate**), which is $\frac{TP}{TP+FN}$, and it can be interpreted as how much of the relevant results were returned – in other words, in this case, what percentage of the actual delays were predicted.

  Another good measure involving true positives is **precision**, which is how much our predicted samples are relevant, which is $\frac{TP}{TP+FP}$. In this case, that would be what percentage of predicted delays were actual delays. For imbalanced classes, it is recommended to use both, but depending on your preference for *FN* over *FP*, you will prefer recall over precision or vice versa.

- **ROC-AUC: ROC** is an acronym for **Receiver Operating Characteristic** and was designed to separate signal from noise. What it does is plot the proportion of **true positive rate** (**recall**) on the x axis and the false positive rate on the y axis. **AUC** stands for **area under the curve**, which is a number between 0 and 1 that assesses the prediction ability of the classifier 1 being perfect, 0.5 being as good as a random coin toss, and anything lower meaning that if we inverted the results of our prediction, we would have a better prediction. To illustrate this, let's generate a ROC curve for our worst-performing model, Naïve Bayes, according to the AUC metric:

```python
plt.tick_params(axis = 'both', which = 'major')
fpr, tpr, _ = metrics.roc_curve(
  y_test_class, class_models['naive_bayes']['probs'])
plt.plot(
    fpr,
    tpr,
    label='ROC curve (area = %0.2f)'
    % class_models['naive_bayes']['ROC_AUC_test']
)
plt.plot([0, 1], [0, 1], 'k-') #random coin toss line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.legend(loc="lower right")
```

The preceding code outputs *Figure 3.4*. Note that the diagonal line signifies half the area. In other words, the point where it has a coin-toss-like prediction quality:
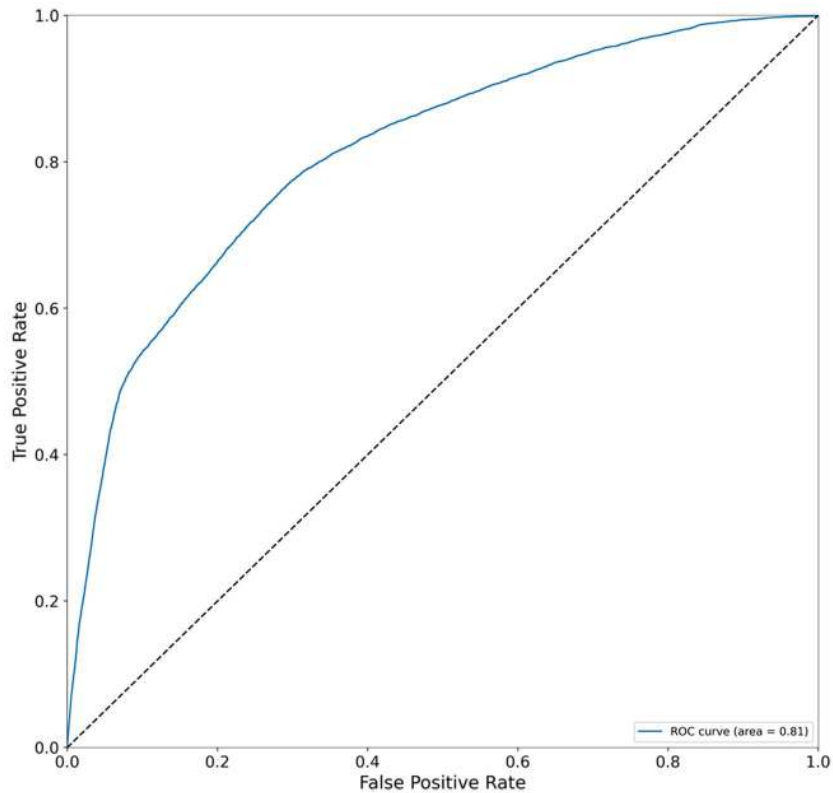
Figure 3.4: ROC curve for Naïve Bayes

- **F1:** The **F1-score** is also called the harmonic average of precision and recall because it's calculated like this: $\frac{2TP}{2TP+FP+FN}$. Since it includes both precision and recall metrics, which pertain to the proportion of true positives, it's a good metric choice to use when your dataset is imbalanced, and you don't prefer either precision or recall.

- **MCC:** The **Matthews correlation coefficient** is a metric drawn from biostatistics. It's gaining popularity in the broader data science community because it has the ability to produce high scores considering *TP*, *FN*, *TN*, and *FP* fairly, because it takes into account the proportions of classes. This makes it optimal for imbalanced classification tasks. Unlike all other metrics used so far, it doesn't range from 0 to 1 but from -1, complete disagreement, to 1, a total agreement between predictions and actuals. The mid-point, 0, is equivalent to a random prediction:

$$MCC = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Our classification metrics are mostly very good, exceeding 96% accuracy and 75% recall. However, even recall isn't everything. For instance, `RandomForest`, due to its class balancing with weights, got the highest recall but did poorly in F1 and MCC, which suggests that precision is not very good.

Ridge classification also had the same setting and had such a poor F1 score that the precision must have been dismal. This doesn't mean this weighting technique is inherently wrong, but it often requires more control. This book will cover techniques to achieve the right balance between fairness and accuracy, accuracy and reliability, reliability and validity, and so on. This is a balancing act that requires many metrics and visualizations. A key takeaway from this exercise should be that a **single metric will not tell you the whole story**, and interpretation is about **telling the most relevant and sufficiently complete story**.

Understanding limitations of traditional model interpretation methods

In a nutshell, traditional interpretation methods *only cover high-level questions about your models* such as the following:

- In aggregate, do they perform well?
- *What* changes in hyperparameters may impact predictive performance?
- *What* latent patterns can you find between the features and their predictive performance?

These questions are very limiting if you are trying to understand not only whether your model works but *why* and *how*?

This gap in understanding can lead to unexpected issues with your model that won't necessarily be immediately apparent. Let's consider that models, once deployed, are not static but dynamic. They face different challenges than they did in the "lab" when you were training them. They may face not only performance issues but issues with bias, such as imbalance with underrepresented classes, or security vulnerabilities with adversarial attacks. Realizing that the features have changed in the real-world environment, we might have to add new features instead of merely retraining with the same feature set. And if there are some troubling assumptions made by your model, you might have to re-examine the whole pipeline. But how do you recognize that these problems exist in the first place? That's when you will need a whole new set of interpretation tools that can help you dig deeper and answer more specific questions about your model. These tools provide interpretations that can truly account for **Fairness, Accountability, and Transparency** (**FAT**), which we discussed in *Chapter 1, Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*

Studying intrinsically interpretable (white-box) models

So far, in this chapter, we have already fitted our training data to model classes representing each of these "white-box" model families. The purpose of this section is to show you exactly why they are *intrinsically interpretable*. We'll do so by employing the models that were previously fitted.

Generalized Linear Models (GLMs)

GLMs are a large family of model classes that have a model for every statistical distribution. Just like **linear regression** assumes your target feature and residuals have a normal distribution, **logistic regression** assumes the Bernoulli distribution. There are GLMs for every distribution, such as **Poisson regression** for Poisson distribution and **multinomial response** for multinomial distribution. You choose which GLM to use based on the distribution of your target variable and whether your data meets the other assumptions of the GLM (they vary). In addition to an underlying distribution, what ties GLMs together into a single family is the fact that they all have a linear predictor. In other words, the $\hat{y}$ target variable (or predictor) can be expressed mathematically as a weighted sum of X features, where weights are called b coefficients. This is the simple formula, the linear predictor function, that all GLMs share:

$$\hat{y} = \beta X$$

However, although they share this same formula, they each have a different link function, which provides a link between the linear predictor function and the mean of the statistical distribution of the GLM. This can add some non-linearity to the resulting model formula while retaining the linear combination between the b coefficients and the X input data, which can be a source of confusion. Still, it's linear because of the linear combination.

There are also many variations for specific GLMs. For instance, **Polynomial regression** is *linear regression* with polynomials of its features, and **ridge regression** is *linear regression* with L2 regularization. We won't cover all GLMs in this section because they aren't needed for the example in this chapter, but all have plausible use cases.

Incidentally, there's also a similar concept called **Generalized Additive Models (GAMs)**, which are GLMs that don't require linear combinations of features and coefficients and instead retain the addition part, but of arbitrary functions applied to the features. GAMs are also interpretable, but they are not as common, and are usually tailored to specific use cases *ad hoc*.

Linear regression

In *Chapter 1*, *Interpretation, Interpretability, and Explainability, and Why Does It All Matter?*, we covered the formula of simple linear regression, which only has a single X feature. Multiple linear regression extends this to have any number of features, so instead of being:

$$\hat{y} = \beta_0$$

it can be:

$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \ldots + \beta_n X_n$$

with n features, and where β_0 is the intercept, and thanks to linear algebra this can be a simple matrix multiplication:

$$\hat{y} = \beta X$$

The method used to arrive at the optimal *b* coefficients, **OLS**, is well-studied and understood. Also, in addition to the coefficients, you can extract confidence intervals for each. The model's correctness depends on whether the input data meets the assumptions: **linearity**, normality, independence, a lack of multicollinearity, and homoscedasticity. We've discussed linearity, so far, quite a bit so we will briefly explain the rest:

- **Normality** is the property that each feature is normally distributed. This can be tested with a **Q-Q plot**, histogram, or **Kolmogorov-Smirnov** test, and non-normality can be corrected with non-linear transformations. If a feature isn't normally distributed, it will make its coefficient confidence intervals invalid.

- **Independence** is when your *observations* (the rows in your dataset) are independent of each other, like different and unrelated events. If your *observations* aren't independent, it could affect your interpretation of the results. In this chapter's example, if you had multiple rows about the same flight, that could violate this assumption and make results hard to understand. This can be tested by looking for duplicate flight numbers.

- Multicollinearity occurs when the features are highly correlated with each other. **Lack of multicollinearity** is desirable because otherwise, you'd have inaccurate coefficients. This can be tested with a **correlation matrix**, **tolerance measure**, or **Variance Inflation Factor** (**VIF**), and it can be fixed by removing one of each highly correlated feature.

- **Homoscedasticity** was briefly discussed in *Chapter 1, Interpretation, Interpretability, and Explainability; and Why Does It All Matter?* and it's when the residuals (the errors) are more or less equal across the regression line. This can be tested with the **Goldfeld–Quandt test**, and heteroscedasticity (the lack of homoscedasticity) can be corrected with non-linear transformations. This assumption is often violated in practice.

Even though we haven't done it for this chapter's example, if you are going to rely on linear regression heavily, it's always good to test these assumptions before you even begin to fit your data to a linear regression model. This book won't detail how this is done because it's more about model-agnostic and deep-learning interpretation methods than delving into how to meet the assumptions of a specific class of models, such as **normality** and **homoscedasticity**. However, we covered the characteristics that trump interpretation the most in *Chapter 2, Key Concepts of Interpretability*, and we will continue to look for these characteristics: **non-linearity**, **non-monotonicity**, and **interactivity**. We will do this mainly because the linearity and correlation of and between features are still relevant, regardless of the modeling class used to make predictions. And these are characteristics that can be easily tested in the methods used for linear regression.

Interpretation

So how do we interpret a linear regression model? Easy! Just get the coefficients and the intercept. Our scikit-learn models have these attributes embedded in the fitted model:

```python
coefs_lm = reg_models['linear']['fitted'].coef_
intercept_lm = reg_models['linear']['fitted'].intercept_
print('coefficients:%s' % coefs_lm)
print('intercept:%s' % intercept_lm)
```

The preceding code outputs the following:

```
coefficients:   [ 0.0045 -0.0053 0.8941 0.0152 ...]
intercept: -37.86
```

So now you know the formula, which looks something like this:

$$\hat{y} = -37.86 + 0.0045X1 + -0.0053X2 + 0.894X3 + ...$$

This formula should provide some intuition on how the model can be interpreted globally. Interpreting each coefficient in the model can be done for multiple linear regression, just as we did with the simple linear regression example in *Chapter 1, Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*. The coefficients act as weights, but they also tell a story that varies depending on the kind of feature. To make interpretation more manageable, let's put our coefficients in a DataFrame alongside the names of each feature:

```
pd.DataFrame({'feature': X_train.columns.tolist(),\
              'coef': coefs_lm.tolist()})
```

The preceding code produces the DataFrame in *Figure 3.5*:

	feature	coef
0	CRS_DEP_TIME	0.004550
1	DEP_TIME	-0.005251
2	DEP_DELAY	0.894126
3	DEP_AFPH	-0.015296
4	DEP_RFPH	-0.469623
5	TAXI_OUT	0.125278
6	WHEELS_OFF	-0.000647
7	CRS_ELAPSED_TIME	-0.012624
8	PCT_ELAPSED_TIME	45.011289
9	DISTANCE	0.000676
10	CRS_ARR_TIME	-0.000370
11	ARR_AFPH	0.000548
12	ARR_RFPH	0.373867
13	WEATHER_DELAY	-0.906364
14	NAS_DELAY	-0.674053
15	SECURITY_DELAY	-0.917411
16	LATE_AIRCRAFT_DELAY	-0.929844
17	DEP_MONTH	-0.039662
18	DEP_DOW	-0.017967
19	ORIGIN_HUB	-1.029129
20	DEST_HUB	-0.394935

Figure 3.5: Coefficients of linear regression features

Here's how to interpret a feature using the coefficients in *Figure 3.5*:

- **Continuous:** Like `ARR_RFPH`, you know that for every one-unit increase (relative flights per hour), it increases the predicted delay by 0.373844 minutes, if all other features stay the same.

- **Binary:** Like `ORIGIN_HUB`, you know the difference between the origin airport being a hub or not is expressed by the coefficient -1.029088. In other words, since it's a negative number, the origin airport is a hub. It reduces the delay by just over 1 minute if all other features stay the same.

- **Categorical:** We don't have categorical features, but we have ordinal features that could have been, and **actually should have been**, categorical features. For instance, `DEP_MONTH` and `DEP_DOW` are integers from 1–12 and 0–6, respectively. If they are treated as ordinals, we are assuming because of the linear nature of linear regression that an increase or decrease in months has an impact on the outcome. It's the same with the day of the week. But the impact is tiny. Had we treated them as dummy or one-hot encoded features, we could measure whether Fridays are more prone to carrier delays than Saturdays and Wednesdays, or Julys more than Octobers and Junes. This couldn't possibly be modeled with them in order, because they have no relation to this order (yep – it's non-linear!). So, say we had a feature called `DEP_FRIDAY` and another called `DEP_JULY`. They are treated like binary features and can tell you precisely what effect a departure being on a Friday or in July has on the model. Some features were kept as ordinal or continuous on purpose, despite being good candidates for being categorical, to demonstrate how not making the right adjustments to your features can impact the **expressive power** of model interpretation. It would have been good to tell airline executives more about how the day and time of a departure impacted delays. Also, in some cases – not in this one – an oversight like this can grossly affect a linear regression model's performance.

The intercept (-37.86) is not a feature, but it does have a meaning, which is, if all features were at 0, what would the prediction be? In practice, this doesn't happen unless your features happen to all have a plausible reason to be 0. Just as in *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?* you wouldn't have expected anyone to have a height of 0, in this example, you wouldn't expect a flight to have a distance of 0. However, if you standardized the features so that they had a mean of 0, then you would change the interpretation of the intercept to be the prediction you expect if all features are their mean value.

Feature importance

The coefficients can also be leveraged to calculate feature importance. Unfortunately, scikit-learn's linear regressor is ill-equipped to do this because it doesn't output the standard error of the coefficients. According to their importance, all it takes to rank features is to divide the β s by their corresponding standard errors. This result is something called the **t-statistic**:

$$t_{\beta_i} = \frac{\beta_i}{SE(\beta_i)}$$

And then you take an absolute value of this and sort them from high to low. It's easy enough to calculate, but you need the standard error. You could reverse-engineer the linear algebra involved to retrieve it using the intercept, and the coefficients returned by scikit-learn. However, it's probably a lot easier to fit the linear regression model again, but this time using the `statsmodels` library, which has a summary with all the statistics included! By the way, `statsmodels` names its linear regressor `OLS`, which makes sense because `OLS` is the name of the mathematical method that fits the data:

```python
linreg_mdl = sm.OLS(y_train_reg, sm.add_constant(X_train))
linreg_mdl = linreg_mdl.fit()
print(linreg_mdl.summary())
```

There's quite a bit to unpack in the regression summary. This book won't address everything except that the t-statistic can tell you how important features are in relation to each other. There's another more pertinent statistical interpretation, which is that if you were to hypothesize that the *b* coefficient is 0 – in other words, that the feature has no impact on the model – the distance of the t-statistic from 0 helps reject that null hypothesis. This is what the **p-value** to the right of the t-statistic does. It's no coincidence that the closest *t* to 0 (for `ARR_AFPH`) has the only p-value above 0.05. This puts this feature at a level of insignificance since everything below 0.05 is statistically significant according to this method of hypothesis testing.

So to rank our features, let's extract the DataFrame from the `statsmodels` summary. Then, we drop the `const` (the intercept) because this is not a feature. Then, we make a new column with the absolute value of the t-statistic and sort it accordingly. To demonstrate how the absolute value of the t-statistic and p-value are inversely related, we are also color-coding these columns:

```python
summary_df = linreg_mdl.summary2().tables[1]
summary_df = summary_df.drop(
    ['const']).reset_index().rename(columns={'index':'feature'}
)
summary_df['t_abs'] = abs(summary_df['t'])
summary_df.sort_values(by='t_abs', ascending=False).
    style.format(
        dict(zip(summary_df.columns[1:], ['{:.4f}']*7)
    )
).background_gradient(cmap='plasma_r', low=0, high=0.1,\
                      subset=['P>|t|', 't_abs'])
```

The preceding code outputs *Figure 3.6*:

	feature	Coef.	Std.Err.	t	P>|t|	[0.025	0.975]	t_abs
2	DEP_DELAY	0.8941	0.0003	2951.0560	0.0000	0.8935	0.8947	2951.0560
16	LATE_AIRCRAFT_DELAY	-0.9298	0.0005	-1827.0181	0.0000	-0.9308	-0.9288	1827.0181
13	WEATHER_DELAY	-0.9064	0.0009	-995.3664	0.0000	-0.9081	-0.9046	995.3664
14	NAS_DELAY	-0.6741	0.0008	-829.1287	0.0000	-0.6756	-0.6725	829.1287
8	PCT_ELAPSED_TIME	45.0113	0.1172	384.0726	0.0000	44.7816	45.2410	384.0726
15	SECURITY_DELAY	-0.9174	0.0055	-167.8571	0.0000	-0.9281	-0.9067	167.8571
5	TAXI_OUT	0.1253	0.0012	104.1196	0.0000	0.1229	0.1276	104.1196
0	CRS_DEP_TIME	0.0045	0.0001	62.8717	0.0000	0.0044	0.0047	62.8717
1	DEP_TIME	-0.0053	0.0001	-57.1159	0.0000	-0.0054	-0.0051	57.1159
3	DEP_AFPH	-0.0153	0.0003	-47.7245	0.0000	-0.0159	-0.0147	47.7245
19	ORIGIN_HUB	-1.0291	0.0267	-38.5894	0.0000	-1.0814	-0.9769	38.5894
12	ARR_RFPH	0.3739	0.0132	28.3860	0.0000	0.3481	0.3997	28.3860
4	DEP_RFPH	-0.4696	0.0172	-27.3532	0.0000	-0.5033	-0.4360	27.3532
7	CRS_ELAPSED_TIME	-0.0126	0.0007	-19.1315	0.0000	-0.0139	-0.0113	19.1315
10	CRS_ARR_TIME	-0.0004	0.0000	-16.9387	0.0000	-0.0004	-0.0003	16.9387
20	DEST_HUB	-0.3949	0.0263	-15.0415	0.0000	-0.4464	-0.3435	15.0415
17	DEP_MONTH	-0.0397	0.0026	-15.0188	0.0000	-0.0448	-0.0345	15.0188
6	WHEELS_OFF	-0.0006	0.0001	-9.6461	0.0000	-0.0008	-0.0005	9.6461
9	DISTANCE	0.0007	0.0001	8.4288	0.0000	0.0005	0.0008	8.4288
18	DEP_DOW	-0.0180	0.0045	-4.0046	0.0001	-0.0268	-0.0092	4.0046
11	ARR_AFPH	0.0005	0.0003	1.6508	0.0988	-0.0001	0.0012	1.6508

Figure 3.6: Linear regression summary table sorted by the absolute value of the t-statistic

Something particularly interesting about the feature importance in *Figure 3.6* is that different kinds of delays occupy five out of the top six positions. Of course, this could be because linear regression is confounding the different non-linear effects these have, or perhaps there's something here we should look further into – especially since the `statsmodels` summary in the "**Warnings**" section cautions:

```
[2] The condition number is large, 5.69e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

This is odd. Hold that thought. We will examine this further later.

Ridge regression

Ridge regression is part of a sub-family of **penalized** or **regularized** regression along with the likes of LASSO and ElasticNet because, as explained earlier in this chapter, it penalizes using the *L2 norm*. This sub-family is also called **sparse linear models** because, thanks to the regularization, it cuts out some of the noise by making irrelevant features less relevant. **Sparsity** in this context means less is more because reduced complexity will lead to lower variance and improved generalization.

To illustrate this concept, look at the feature importance table (*Figure 3.6*) we output for linear regression. Something that should be immediately apparent is how the t_abs column starts with every row a different color, and then a whole bunch of them are the same shade of yellow. Because of the variation in confidence intervals, the absolute t-value is not something you can take proportionally and say that your top feature is hundreds of times more relevant than every one of your bottom 10 features. However, it should indicate that there are significantly more important features than others to the point of irrelevance, and possibly confoundment, hence creating noise. There's ample research on how there's a tendency for a small subset of features to have the most substantial effects on the outcome of the model. This is called the **bet on sparsity principle**. Whether it's true or not for your data, it's always good to test the theory by applying regularization, especially in cases where data is very wide (many features) or exhibits multicollinearity. These regularized regression techniques can be incorporated into feature selection processes or to inform your understanding of what features are essential.

There is a technique to adapt ridge regression to classification problems. It was briefly discussed before. It converts the labels to a -1 to 1 scale for training to predict values between -1 and 1, and then turns them back to a 0–1 scale. However, it uses regularized linear regression to fit the data and can be interpreted in the same way.

Interpretation

Ridge regression can be interpreted in the same way as linear regression, both globally and locally, because once the model has been fitted, there's no difference. The formula is the same:

$$\hat{y} = \beta^{ridge} X$$

Except β^{ridge} coefficients are different because they were penalized with a λ parameter, which controls how much shrinkage to apply.

We can quickly compare coefficients by extracting the ridge coefficients from their fitted model and placing them side by side in a DataFrame with the coefficients of the linear regression:

```
coefs_ridge = reg_models['ridge']['fitted'].coef_
coef_ridge_df =pd.DataFrame(
    {
        'feature':X_train.columns.values.tolist(),
        'coef_linear': coefs_lm,\
        'coef_ridge': coefs_ridge
    }
```

```
)
coef_ridge_df['coef_regularization'] =\
    coef_ridge_df['coef_linear'] - coef_ridge_df['coef_ridge']
coef_ridge_df.style.background_gradient(
    cmap='plasma_r', low=0, high=0.1 , subset=['coef_regularization']
)
```

As you can tell in the *Figure 3.7* output of the preceding code, the coefficients are always slightly different, but sometimes they are lower and sometimes higher:

	feature	coef_linear	coef_ridge	coef_regularization
0	CRS_DEP_TIME	0.004550	0.004496	0.000054
1	DEP_TIME	-0.005251	-0.004820	-0.000431
2	DEP_DELAY	0.894126	0.892334	0.001792
3	DEP_AFPH	-0.015296	-0.015189	-0.000107
4	DEP_RFPH	-0.469623	-0.469629	0.000006
5	TAXI_OUT	0.125278	0.125164	0.000114
6	WHEELS_OFF	-0.000647	0.000013	-0.000660
7	CRS_ELAPSED_TIME	-0.012624	-0.012624	-0.000000
	:	:	:	:
15	SECURITY_DELAY	-0.917411	-0.917412	0.000001
16	LATE_AIRCRAFT_DELAY	-0.929844	-0.930708	0.000865
17	DEP_MONTH	-0.039662	-0.039664	0.000002
18	DEP_DOW	-0.017967	-0.017965	-0.000001
19	ORIGIN_HUB	-1.029129	-1.029129	-0.000000
20	DEST_HUB	-0.394935	-0.394935	0.000001

Figure 3.7: Linear regression coefficients compared to ridge regression coefficients

We didn't save the λ parameter (which scikit-learn calls *alpha*) that the ridge regression cross-validation deemed optimal. However, we can run a little experiment of our own to figure out which parameter was the best. We do this by iterating through 100 possible alphas values between 100 (1) and 1013 (100,000,000,000,000), fitting the data to the ridge model, and then appending the coefficients to an array. We exclude the eight coefficient in the array because it's so much larger than the rest, and it will make it harder to visualize the effects of shrinkage:

```
num_alphas = 100
alphas = np.logspace(0, 13, num_alphas)
alphas_coefs = []
for alpha in alphas:
    ridge = linear_model.Ridge(alpha=alpha).fit(
```

```
        X_train, y_train_reg)
    alphas_coefs.append(np.concatenate((ridge.coef_[:8],
                                        ridge.coef_[9:])))
```

Now that we have an array of coefficients, we can plot the progression of coefficients:

```
plt.gca().invert_xaxis()
plt.tick_params(axis = 'both', which = 'major')
plt.plot(alphas, alphas_coefs)
plt.xscale("log")
plt.xlabel('Alpha')
plt.ylabel('Ridge coefficients')
plt.grid()
plt.show()
```

The preceding code generates *Figure 3.8*:
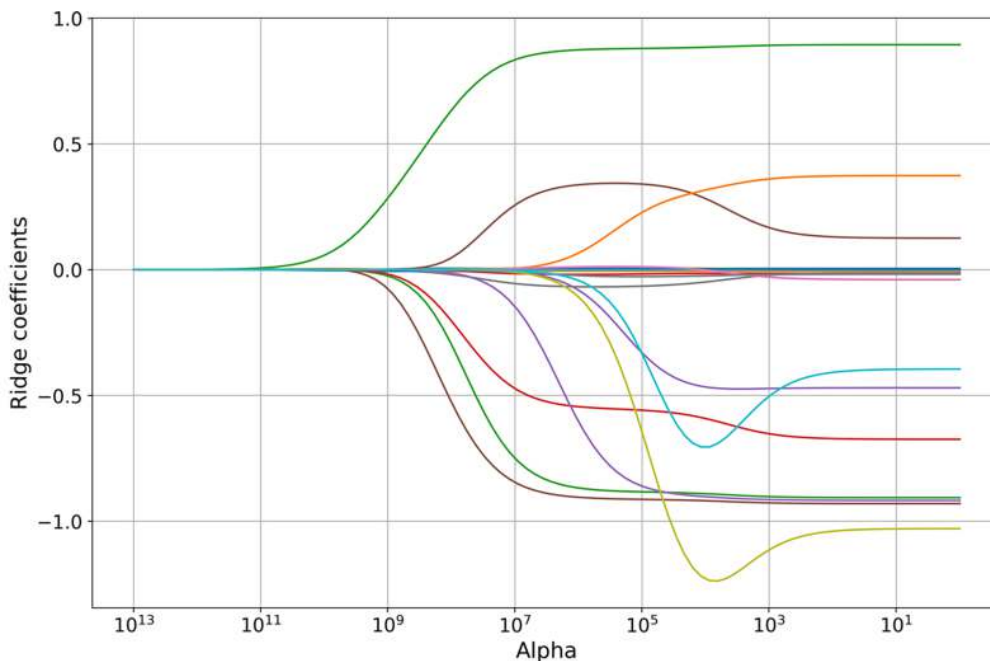
Figure 3.8: Value of alpha hyperparameters versus the value of ridge regression coefficients

Something to note in *Figure 3.8* is that the higher the alpha, the higher the regularization. This is why when alpha is 1012, all coefficients have converged to 0, and as the alpha becomes smaller, they get to a point where they have all diverged and more or less stabilized. In this case, this point is reached at about 102. Another way of seeing it is when all coefficients are around 0, it means that the regularization is so strong that all features are irrelevant. When they have sufficiently diverged and stabilized, the regularization makes them all relevant, which defeats the purpose.

Now on that note, if we go back to our code, we will find that this is what we chose for alphas in our `RidgeCV: alphas=[1e-3, 1e-2, 1e-1, 1]`. As you can tell from the preceding plot, by the time the alphas have reached `1` and below, the coefficients have already stabilized even though they are still fluctuating slightly. This can explain why our ridge was not better performing than linear regression. Usually, you would expect a regularized model to perform better than one that isn't – unless your hyperparameters are not right.

> Interpretation and hyperparameters
>
> Well-tuned regularization can help cut out the noise and thus increase interpretability, but the alphas chosen for `RidgeCV` were selected on purpose to be able to convey this point: *regularization can only work if you chose hyperparameters correctly*, or, when regularization hyperparameter tuning is automatic, the method must be optimal for your dataset.

Feature importance

This is precisely the same as with linear regression, but again we need the standard error of the coefficients, which is something that cannot be extracted from the scikit-learn model. You can use the `statsmodels fit_regularized` method to this effect.

Polynomial regression

Polynomial regression is a special case of linear or logistic regression where the features have been expanded to have higher degree terms. We have only performed polynomial linear regression in this chapter's exercise, so we will only discuss this variation. However, it is applied similarly.

A two-feature multiple linear regression would look like this:

$$\widehat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$

However, in polynomial regression, every feature is expanded to have higher degree terms and interactions between all the features. So, if this two-feature example was expanded to a second-degree polynomial, the linear regression formula would look like this:

$$\widehat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_1 X_2 + \beta_5 X_2^2$$

It's still linear regression in every way except it has extra features, higher-degree terms, and interactions. While you can limit polynomial expansion to only one or a few features, we used `PolynomialFeatures`, which does this to all features. Therefore, 21 features were likely multiplied many times over. We can extract the coefficients from our fitted model and, using the `shape` property of the `numpy` array, return how many coefficients were generated. This amount corresponds to the number of features generated:

```
reg_models['linear_poly']['fitted'].\
get_params()['linearregression'].coef_.shape[0]
```

It outputs 253. We can do the same with the version of polynomial regression, which was with interaction terms only:

```
reg_models['linear_interact']['fitted'].\
get_params()['linearregression'].coef_.shape[0]
```

The above code outputs 232. The reality is that most terms in a polynomial generated like this are interactions between all the features.

Interpretation and feature importance

Polynomial regression can be interpreted, both globally and locally, in precisely the same way as linear regression. In this case, it's not practical to understand a formula with 253 linearly combined terms, so it loses what we defined in *Chapter 2, Key Concepts of Interpretability*, as **global holistic interpretation**. However, it still can be interpreted in all other scopes and retains many of the properties of linear regression. For instance, since the model is additive, it is easy to separate the effects of the features. You can also use the same many peer-reviewed tried and tested statistical methods that are used for linear regression. For instance, you can use the t-statistic, p-value, confidence bounds, R-squared, as well as the many tests used to assess goodness of fit, residual analysis, linear correlation, and analysis of variance. This wealth of statistically proven methods to test and interpret models isn't something most model classes can count on. Unfortunately, many of them are model-specific to linear regression and its special cases.

Also, we won't do it here because there are so many terms. Still, you could undoubtedly rank features for polynomial regression in the same way we have for linear regression using the `statsmodels` library. The challenge is figuring out the order of the features generated by `PolynomialFeatures` to name them accordingly in the feature name column. Once this is done, you can tell if some second-degree terms or interactions are important. This could tell you if these features have a non-linear nature or highly depend on other features.

Logistic regression

We discussed logistic regression as well as its interpretation and feature importance in *Chapter 2, Key Concepts of Interpretability*. We will only expand on that a bit here in the context of this chapter's classification exercise and to underpin why exactly it is interpretable. The fitted logistic regression model has coefficients and intercepts just as the linear regression model does:

```
coefs_log = class_models['logistic']['fitted'].coef_
intercept_log = class_models['logistic']['fitted'].intercept_
print('coefficients:%s' % coefs_log)
print('intercept:%s' % intercept_log)
```

The preceding code outputs this:

```
coefficients: [[-6.31114061e-04 -1.48979793e-04  2.01484473e-01  1.32897749e-01
1.31740116e-05 -3.83761619e-04 -7.60281290e-02  ..]]
intercept: [-0.20139626]
```

However, the way these coefficients appear in the formula for a specific prediction $\hat{y}^{(i)}$ is entirely different:

$$P(\hat{y}^{(i)} = 1) = \frac{e^{\beta_0 + \beta_1 X_1^{(i)} + \beta_2 X_2^{(i)} \dots + \beta_n X_n^{(i)}}}{1 + e^{\beta_0 + \beta_1 X_1^{(i)} + \beta_2 X_2^{(i)} \dots + \beta_n X_n^{(i)}}}$$

In other words, the probability that $\hat{y}^{(i)} = 1$ (is a positive case) is expressed by a **logistic function** that involves exponentials of the linear combination of β coefficients and the x features. The presence of the exponentials explains why the coefficients extracted from the model are log odds because to isolate the coefficients, you should apply a logarithm to both sides of the equation.

Interpretation

To interpret each coefficient, you do it in precisely the same way as with linear regression, except with each unit increase in the features, you increase the odds of getting the positive case by a factor expressed by the exponential of the coefficient – all things being equal (remember the **ceteris paribus** assumption discussed in *Chapter 2*, *Key Concepts of Interpretability*). An exponential (e^{β}) has to be applied to each coefficient because they express an increase in log odds and not odds. Besides incorporating the log odds into the interpretation, the same that was said about continuous, binary, and categorical in linear regression interpretation applies to logistic regression.

Feature importance

Frustrating as it is, there isn't consensus yet from the statistical community on how to best get feature importance for logistic regression. There's a standardize-all-features-first method, a pseudo R^2 method, a *one feature at a time* ROC AUC method, a partial chi-squared statistic method, and then the simplest one, which is multiplying the standard deviations of each feature times the coefficients. We won't cover all these methods, but it has to be noted that computing feature importance consistently and reliably is a problem for most model classes, even white-box ones. We will dig deeper into this in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*. For logistic regression, perhaps the most popular method is achieved by standardizing all the features before training – that is, making sure they are centered at zero and divided by their standard deviation. But we didn't do this because although it has other benefits, it makes the interpretation of coefficients more difficult, so here we are using the rather crude method leveraged in *Chapter 2*, *Key Concepts of Interpretability*, which is to multiply the standard deviations of each feature times the coefficients:

```
stdv = np.std(X_train, 0)
abs(coefs_log.reshape(21,) * stdv).sort_values(ascending=False)
```

The preceding code yields the following output:

```
DEP_DELAY             8.92
CRS_ELAPSED_TIME      6.03
DISTANCE              5.31
LATE_AIRCRAFT_DELAY   4.99
NAS_DELAY             2.39
```

```
WEATHER_DELAY         2.16
TAXI_OUT              1.31
SECURITY_DELAY        0.38
ARR_AFPH              0.32
WHEELS_OFF        0.01
PCT_ELAPSED_TIME    0.003
```

It can still approximate the importance of features quite well. And just like with linear regression, you can tell that delay features are ranking quite high. All five of them are among the top eight features. Indeed, it's something we should look into. We will discuss more on that as we discuss some other white-box methods.

Decision trees

Decision trees have been used for the longest time, even before they were turned into algorithms. They hardly require any mathematical abilities to understand them, and this low barrier to comprehensibility makes them extremely interpretable in their simplest representations. However, in practice, there are many types of decision tree learning, and most of them are not very interpretable because they use **ensemble methods** (boosting, bagging, and stacking), or even leverage PCA or some other embedder. Even non-ensembled decision trees can get extremely complicated as they become deeper. Regardless of the complexity of a decision tree, they can always be mined for important insights about your data and expected predictions, and they can be fitted to both regression and classification tasks.

CART decision trees

The **Classification and Regression Trees** (**CART**) algorithm is the "vanilla" no-frills decision tree of choice in most use cases. And as noted, most decision trees aren't white-box models, but this one is because it is expressed as a mathematical formula, visualized, and printed as a set of rules that subdivides the tree into branches and eventually the leaves.

The mathematical formula:

$$\widehat{y} = \sum_{m=1}^{M} \mu_m \, I\{x \in R_m\}$$

And what this means is that if according to the identity function I, x is in the subset R_m, then it returns a 1, otherwise a 0. This binary term is multiplicated by the averages of all elements in the subset R_m denoted as μ_m. So if x_i is in the subset belonging to the leaf node R_k then the prediction $\widehat{y}_i = \mu_k$. In other words, the prediction is the average of all elements in the subset R_k. This is what happens to regression tasks, and in binary classification, there is simply no μ_m to multiply the I identify function.

At the heart of every decision tree algorithm, there's a method to generate the R_m subsets. For CART, this is achieved using something called the **Gini index**, recursively splitting on where the two branches are as different as possible. This concept will be explained in greater detail in *Chapter 4, Global Model-Agnostic Interpretation Methods*.

Interpretation

A decision tree can be globally and locally interpreted visually. Here, we have established a maximum depth of 2 (`max_depth=2`) because we could generate all 7 layers, but the text would be too small to appreciate. One of the limitations of this method is that it can get complicated to visualize with depths above 3 or 4. However, you can always programmatically traverse the branches of the tree and visualize only some branches at a time:

```
fig, axes = plt.subplots(
    nrows = 1, ncols = 1, figsize = (16,8), dpi=600)
tree.plot_tree(
    class_models['decision_tree']['fitted'],
    feature_names=X_train.columns.values.tolist(),
    filled = True, max_depth=2
)
fig.show()
```

The preceding code prints out the tree in *Figure 3.9*. From the tree, you can tell that the very first branch splits the decision tree based on the value of `DEP_DELAY` being equal to or smaller than 20.5. It tells you the Gini index that informed that decision and the number of `samples` (just another way of saying observations, data points, or rows) present. You can traverse these branches till they reach a leaf. There is one leaf node in this tree, and it is on the far left. This is a classification tree, so you can tell by the value =[629167, 0] that all 629,167 samples left in this node have been classified as a 0 (not delayed):
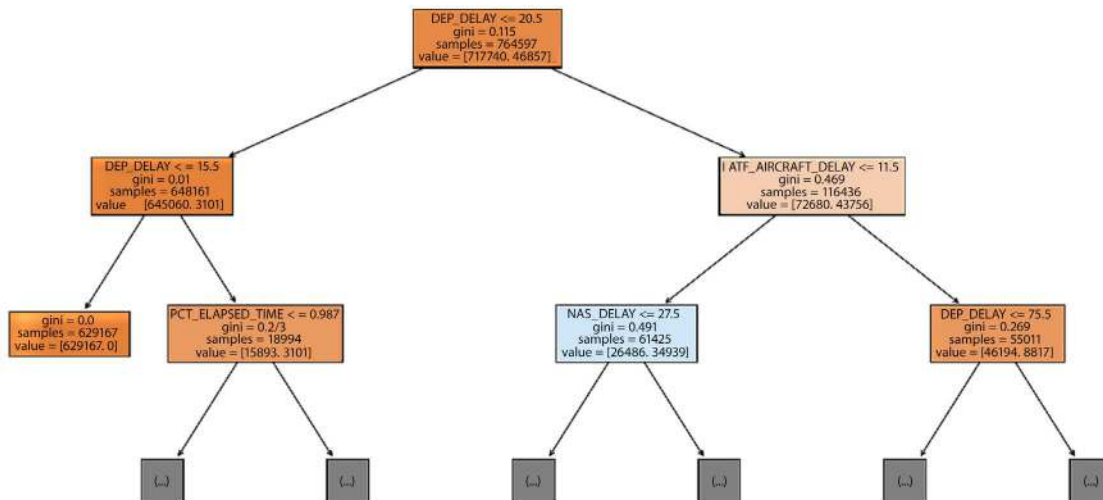
Figure 3.9: Our models' plotted decision tree

Another way the tree can be better visualized but with fewer details, such as the Gini index and sample size, is by printing out the decisions made in every branch and the class in every node:

```
text_tree = tree.export_text(
    class_models['decision_tree']['fitted'],
    feature_names=X_train.columns.values.tolist()
)
print(text_tree)
```

And the preceding code outputs the following:

```
|--- DEP_DELAY <= 20.50
|   |--- DEP_DELAY <= 15.50
|   |   |--- class: 0
|   |--- DEP_DELAY >  15.50
|   |   |--- PCT_ELAPSED_TIME <= 0.99
|   |   |   |--- PCT_ELAPSED_TIME <= 0.98
|   |   |   |   |--- PCT_ELAPSED_TIME <= 0.96
|   |   |   |   |   |--- CRS_ELAPSED_TIME <= 65.50
|   |   |   |   |   |   |--- PCT_ELAPSED_TIME <= 0.94
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |   |--- PCT_ELAPSED_TIME >  0.94
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- CRS_ELAPSED_TIME >  65.50
|   |   |   |   |   |   |--- PCT_ELAPSED_TIME <= 0.95
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |   |--- PCT_ELAPSED_TIME >  0.95
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |--- PCT_ELAPSED_TIME >  0.96
|   |   |   |   |   |--- CRS_ELAPSED_TIME <= 140.50
|   |   |   |   |   |   |--- DEP_DELAY <= 18.50
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |   |--- DEP_DELAY >  18.50
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |--- CRS_ELAPSED_TIME >  140.50
|   |   |   |   |   |   |--- DEP_DELAY <= 19.50
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |   |   |   |--- DEP_DELAY >  19.50
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |--- PCT_ELAPSED_TIME >  0.98
|   |   |   |   |--- DEP_DELAY <= 18.50
|   |   |   |   |   |--- DISTANCE <= 326.50
|   |   |   |   |   |   |--- LATE_AIRCRAFT_DELAY <= 0.50
|   |   |   |   |   |   |   |--- class: 1
|   |   |   |   |   |   |--- LATE_AIRCRAFT_DELAY >  0.50
|   |   |   |   |   |   |   |--- class: 0
|   |   |   |--- … (goes on for 6 more pages!)
```

Figure 3.10: Our decision tree's structure

There's a lot more that can be done with a decision tree, and scikit-learn provides an API to explore the tree.

Feature importance

Calculating feature importance in a CART decision tree is reasonably straightforward. As you can appreciate from the visualizations, some features appear more often in the decisions, but their appearances are weighted by how much they contributed to the overall reduction in the Gini index compared to the previous node. All the sum of the relative decrease in the Gini index throughout the tree is tallied, and the contribution of each feature is a percentage of this reduction:

```python
dt_imp_df = pd.DataFrame(
    {
        'feature':X_train.columns.values.tolist(),
        'importance': class_models['decision_tree']['fitted'].\
                    feature_importances_
    }
).sort_values(by='importance', ascending=False)
dt_imp_df
```

The `dt_imp_df` DataFrame output by the preceding code can be appreciated in *Figure 3.11*:

	feature	importance
2	DEP_DELAY	0.527482
16	LATE_AIRCRAFT_DELAY	0.199153
8	PCT_ELAPSED_TIME	0.105381
13	WEATHER_DELAY	0.101649
14	NAS_DELAY	0.062732
15	SECURITY_DELAY	0.001998
9	DISTANCE	0.001019
7	CRS_ELAPSED_TIME	0.000281
	:	:
4	DEP_RFPH	0.000000
20	DEST_HUB	0.000000

Figure 3.11: Our decision tree's feature importance

This last feature importance table, *Figure 3.11*, increases suspicions about the delay features. They occupy, yet again, five of the top six positions. Is it possible that all five of them have such an outsized effect on the model?

Interpretation and domain expertise

The target `CARRIER_DELAY` is also called a dependent variable because it's dependent on all the other features, the independent variables. Even though a statistical relationship doesn't imply causation, we want to inform our feature selection based on our understanding of what independent variables could plausibly affect a dependent one.

It makes sense that a departure delay (`DEPARTURE_DELAY`) affects the arrival delay (which we removed), and therefore, `CARRIER_DELAY`. Similarly, `LATE_AIRCRAFT_DELAY` makes sense as a predictor because it is known before the flight takes off if a previous aircraft was several minutes late, causing this flight to be at risk of arriving late, but not as a cause of the current flight (ruling this option out). However, even though the Bureau of Transportation Statistics website defines delays in such a way that they appear to be discrete categories, some may be determined well after a flight has departed. For instance, in predicting a delay mid-flight, could we use `WEATHER_DELAY` if the bad weather hasn't yet happened? And could we use `SECURITY_DELAY` if the security breach hasn't yet occurred? The answers to these questions are that we probably shouldn't because the rationale for including them is they could serve to rule out `CARRIER_DELAY`, but this only works if they are discrete categories that pre-date the dependent variable! If they don't they would be producing what is known as data leakage. Before coming to further conclusions, what you would need to do is talk to the airline executives to determine the timeline on which each delay category gets consistently set and (hypothetically) is accessible from the cockpit or the airline's command center. Even if you are forced to remove them from the models, maybe other data can fill the void in a meaningful way, such as the first 30 minutes of flight logs and/or historical weather patterns.

Interpretation is not always directly inferred from the data and the machine learning models, but by working closely with domain experts. But sometimes domain experts can mislead you too. In fact, another insight is with all the time-based metrics and categorical features we engineered at the beginning of the chapter (`DEP_DOW`, `DEST_HUB`, `ORIGIN_HUB`, and so on). It turns out they have consistently had little to no effect on the models. Despite the airline executives hinting at the importance of days of the week, hubs, and congestion, we should have explored the data further, looking for correlations before engineering the data. But even if we do engineer some useless features, it also helps to use a white-box model to assess their impact, as we have. In data science, practitioners often will learn the same way that the most performant machine learning models do – by trial and error!

RuleFit

RuleFit is a model-class family that is a hybrid between a LASSO linear regression to get regularized coefficients for every feature and decision rules, which also uses LASSO to regularize. These **decision rules** are extracted by traversing a decision tree, finding interaction effects between features, and assigning coefficients to them based on their impact on the model. The implementation used in this chapter uses gradient-boosted decision trees to perform this task.

We haven't covered decision rules explicitly in this chapter, but they are yet another family of **intrinsically interpretable models**. They weren't included because, at the time of writing, the only Python library that supports decision rules, called **Bayesian Rule List** (**BRL**) by Skater, is still at an experimental stage. In any case, the concept behind decision rules is very similar. They extract the feature interactions from a decision tree but don't discard the leaf node, and instead of assigning coefficients, they use the predictions in the leaf node to construct the rules. The last rule is a catch-all, like an *ELSE* statement. Unlike RuleFit, it can only be understood sequentially because it's so similar to any *IF-THEN-ELSE* statement, but that's its main advantage.

Interpretation and feature importance

You can put everything you need to know about RuleFit into a single DataFrame (`rulefit_df`). Then you remove the rules that have a coefficient of `0`. It has these because in LASSO, unlike ridge, coefficient estimates converge to zero. You can sort the DataFrame by importance in a descending manner to see what features or feature interactions (in the form of rules) are most important:

```
rulefit_df = reg_models['rulefit']['fitted'].get_rules()
rulefit_df = rulefit_df[rulefit_df.coef !=0].\
sort_values(
    by="importance", ascending=False
)
rulefit_df
```

The rules in the `rulefit_df` DataFrame can be seen in *Figure 3.12*:

	rule	type	coef	support	importance
101	LATE_AIRCRAFT_DELAY <= 222.5 & DEP_DELAY > 344.0 & WEATHER_DELAY <= 166.0	rule	222.024721	0.001684	9.102113
42	LATE_AIRCRAFT_DELAY <= 333.5 & DEP_DELAY > 477.5	rule	172.103034	0.001122	5.762432
16	LATE_AIRCRAFT_DELAY	linear	-0.386073	1.000000	4.523663
2	DEP_DELAY	linear	0.163704	1.000000	4.282909
64	DEP_DELAY > 1206.0	rule	278.817372	0.000187	3.812982
142	LATE_AIRCRAFT_DELAY <= 198.0 & DEP_DELAY > 341.5 & DEP_DELAY <= 788.0	rule	-92.790467	0.001496	3.586813
134	DEP_DELAY > 300.0 & LATE_AIRCRAFT_DELAY <= 158.5 & DEP_DELAY > 576.5	rule	115.440190	0.000748	3.156531
23	DEP_DELAY > 66.5 & NAS_DELAY > 43.5 & LATE_AIRCRAFT_DELAY <= 19.5 & DEP_DELAY <= 849.0	rule	-41.899504	0.004302	2.742345
	:	:	:	:	:
18	DEP_DOW	linear	0.009907	1.000000	0.019798
45	DEP_DELAY <= 66.5 & DEP_DELAY <= 20.5 & DEP_DELAY <= 849.0	rule	-0.042437	0.847924	0.015239
170	DEP_DELAY <= 880.5	rule	-0.269029	0.999252	0.007356

Figure 3.12: RuleFit's rules

There's a `type` for every RuleFit feature in *Figure 3.12*. Those that are `linear` are interpreted as you would any linear regression coefficient. Those that are `type=rule` are also to be treated like binary features in a linear regression model. For instance, if the rule `LATE_AIRCRAFT_DELAY <= 333.5 & DEP_DELAY > 477.5` is true, then the coefficient `172.103034` is applied to the prediction. The rules capture the interaction effects, so you don't have to add interaction terms to the model manually or use some non-linear method to find them. Furthermore, it does this in an easy-to-understand manner. You can use RuleFit to guide your understanding of feature interactions even if you choose to productionize other models.

Nearest neighbors

Nearest neighbors is a family of models that even includes unsupervised methods. All of them use the closeness between data points to inform their predictions. Of all these methods, only the supervised k-NN and its cousin Radius Nearest Neighbors are somewhat interpretable.

k-Nearest Neighbors

The idea behind **k-NN** is straightforward. It takes the *k* closest points to a data point in the training data and uses their labels (`y_train`) to inform the predictions. If it's a classification task, it's the **mode** of all the labels, and if it's a regression task, it's the **mean**. It's a **lazy learner** because the "fitted model" is not much more than the training data and the parameters, such as *k* and the list of classes (if it's a classification). It doesn't do much till inference. That's when it leverages the training data, tapping into it directly rather than extracting parameters, weights/biases, or coefficients learned by the model as **eager learners** do.

Interpretation

k-NN only has local interpretability because since there's no fitted model, you don't have global modular or global holistic interpretability. For classification tasks, you could attempt to get a sense of this using the decision boundaries and regions we studied in *Chapter 2, Key Concepts of Interpretability*. Still, it's always based on local instances.

To interpret a local point from our test dataset, we query the `pandas` DataFrame using its index. We will be using flight #721043:

```python
print(X_test.loc[721043,:])
```

The preceding code outputs the following `pandas` series:

```
CRS_DEP_TIME          655.00
DEP_TIME             1055.00
DEP_DELAY             240.00
TAXI_OUT               35.00
WHEELS_OFF            1130.00
CRS_ARR_TIME          914.00
CRS_ELAPSED_TIME      259.00
DISTANCE             1660.00WEATHER_DELAY        0.00
NAS_DELAY              22.00
SECURITY_DELAY          0.00
LATE_AIRCRAFT_DELAY   221.00
DEP_AFPH               90.80
ARR_AFPH               40.43
DEP_MONTH              10.00
DEP_DOW                 4.00
DEP_RFPH                0.89
ARR_RFPH                1.06
```

```
ORIGIN_HUB                1.0
DEST_HUB                0.00
PCT_ELAPSED_TIME            1.084942
Name: 721043, dtype: float64
```

In the y_test_class labels for flight #721043, we can tell that it was delayed because this code outputs 1:

```
print(y_test_class[721043])
```

However, our k-NN model predicted that it was not because this code outputs 0:

```
print(class_models['knn']['preds'][X_test.index.get_loc(721043)])
```

Please note that the predictions are output as a numpy array, so we can't access the prediction for flight #721043 using its pandas index (721043). We have to use the sequential location of this index in the test dataset using get_loc to retrieve it.

To find out why this was the case, we can use kneighbors on our model to find the seven nearest neighbors of this point. To this end, we have to reshape our data because kneighbors will only accept it in the same shape found in the training set, which is (n, 21) where n is the number of observations (rows). In this case, n=1 because we only want the nearest neighbors for a single data point. And as you can tell from what was output by X_test.loc[721043, :], the pandas series has a shape of (21,1), so we have to reverse this shape:

```
print(class_models['knn']['fitted'].\
      kneighbors(X_test.loc[721043,:].values.reshape(1,21), 7))
```

kneighbors outputs two arrays:

```
(array([[143.3160128 , 173.90740076, 192.66705727, 211.57109221,
         243.57211853, 259.61593993, 259.77507391]]),
 array([[105172, 571912,  73409,  89450,  77474, 705972, 706911]]))
```

The first is the distance of each of the seven closest training points to our test data point. And the second is the location of these data points in the training data:

```
print(y_train_class.iloc[[105172, 571912, 73409, 89450, 77474,\
                           705972, 706911]])
```

The preceding code outputs the following pandas series:

```
3813       0
229062     1
283316     0
385831     0
581905     1
726784     1
179364     0
Name: CARRIER_DELAY, dtype: int64
```

We can tell that the prediction reflects the **mode** because the most common class in the seven nearest points was 0 (not delayed). You can increase or decrease the *k* to see if this holds. Incidentally, when using binary classification, it's recommended to choose an odd-numbered *k* so that there are no ties. Another important aspect is the distance metric that was used to select the closest data points. You can easily find out which one it is using:

```
print(class_models['knn']['fitted'].effective_metric_)
```

The output is Euclidean, which makes sense for this example. After all, Euclidean is optimal for a **real-valued vector space** because most features are continuous. You could also test alternative distance metrics such as `minkowski`, `seuclidean`, or `mahalanobis`. When most of your features are binary and categorical, you have an **integer-valued vector space**. So your distances ought to be calculated with algorithms suited for this space such as `hamming` or `canberra`.

Feature importance

Feature importance is, after all, a global model interpretation method and k-NN has a hyper-local nature, so there's no way of deriving feature importance from a k-NN model.

Naïve Bayes

Like GLMs, Naïve Bayes is a family of model classes with a model tailored to different statistical distributions. However, unlike GLMs' assumption that the target *y* feature has the chosen distribution, all Naïve Bayes models assume that your *X* features have this distribution. More importantly, they were based on Bayes' theorem of conditional probability, so they output a probability and are, therefore, exclusively classifiers. But they treat the probability of each feature impacting the model independently, which is a strong assumption. This is why they are called naïve. There's one for Bernoulli called Bernoulli Naïve Bayes, one for multinomial called **Multinomial Naïve Bayes**, and, of course, one for Gaussian, which is the most common.

Gaussian Naïve Bayes

Bayes' theorem is defined by this formula: $P(A|B) = \dfrac{P(A|B) \cdot P(A)}{P(B)}$

In other words, to find the probability of *A* happening given that *B* is true, you take the conditional probability of *B* given *A* is true times the probability of *A* occurring, divided by the probability of *B*. In the context of a machine learning classifier, this formula can be rewritten as follows:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

This is because what we want is the probability of *y* given *X* is true. But our *X* has more than one feature, so this can be expanded like this:

$$P(y|x_1, x_2, \dots x_n) = \frac{P(x_1|y)P(x_2|y) \dots P(x_n|y) \cdot P(y)}{P(x_1)P(x_2) \dots P(x_n)}$$

To compute $\hat{y}$ predictions, we have to consider that we have to calculate and compare probabilities for each C_k class (the probability of a delay versus the probability of no delay) and choose the class with the highest probability:

$$\hat{y} = P(y|X) = \overset{\arg max}{\underset{C_k}{}} P(y = C_k) \prod_{i=1}^{n} P(x_i|y = C_k)$$

Calculating the probability of each class $P(y = C_k)$ (also known as the class prior) is relatively trivial. In fact, the fitted model has stored this in an attribute called `class_prior_`:

```
print(class_models['naive_bayes']['fitted'].class_prior_)
```

This outputs the following:

```
array([0.93871674, 0.06128326])
```

Naturally, since delays caused by the carrier only occur 6% of the time, there is a marginal probability of this occurring.

Then the formula has a product $\prod_{i=1}^{n}$ of conditional probabilities that each feature belongs to a class $P(x_i|y = C_k)$. Since this is binary there's no need to calculate the probabilities of multiple classes because they are inversely proportional. Therefore, we can drop C_k and replace it with a 1 like this:

$$\hat{y} = P(y = 1|X) = P(y = 1) \prod_{i=1}^{n} P(x_i|y = 1)$$

This is because what we are trying to predict is the probability of a delay. Also, $P(x_i|y = 1)$ is its own formula, which differs according to the assumed distribution of the model – in this case, Gaussian:

$$P(x_i|y = 1) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i-\theta_i)^2}{2\sigma_2^i}}$$

This formula is called the probability density of the Gaussian distribution.

Interpretation and feature importance

So what are these **sigmas** (σ_i) and **thetas** (θ_i) in the formula? They are, respectively, the variance and mean of the x_i feature when $y = 1$. The concept behind this is that features have a different variance and mean in one class versus another, which can inform the classification. This is a binary classification task, but you could calculate σ_i and θ_i for both classes. Fortunately, the fitted model has this stored:

```
print(class_models['naive_bayes']['fitted'].sigma_)
```

There are two arrays output, the first one corresponding to the negative class and the second to the positive. The arrays contain the sigma (variance) for each of the 21 features given the class:

```
array([[2.50123026e+05, 2.61324730e+05, ..., 1.13475535e-02],
       [2.60629652e+05, 2.96009867e+05, ..., 1.38936741e-02]])
```

You can also extract the thetas (means) from the model:

```
print(class_models['naive_bayes']['fitted'].theta_)
```

The preceding code also outputs two arrays, one for each class:

```
array([[1.30740577e+03, 1.31006271e+03, ..., 9.71131781e-01],
       [1.41305545e+03, 1.48087887e+03, ..., 9.83974416e-01]])
```

These two arrays are all you need to debug and interpret Naïve Bayes results because you can use them to compute the conditional probability that the x_i feature is given a positive class $P(x_i \mid y)$. You could use this probability to rank the features by importance on a global level, or interpret a specific prediction on a local level.

Naïve Bayes is a fast algorithm with some good use cases, such as spam filtering and recommendation systems, but the independence assumption hinders its performance in most situations. Speaking of performance, let's discuss this topic in the context of interpretability.

Recognizing the trade-off between performance and interpretability

We have briefly touched on this topic before, but high performance often requires complexity, and complexity inhibits interpretability. As studied in *Chapter 2*, *Key Concepts of Interpretability*, this complexity comes from primarily three sources: non-linearity, non-monotonicity, and interactivity. If the model adds any complexity, it is **compounded by the number and nature of features** in your dataset, which by itself is a source of complexity.

Special model properties

These special properties can help make a model more interpretable.

The key property: explainability

In *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, we discussed why being able to look under the hood of the model and intuitively understand how all its moving parts derive its predictions in a consistent manner is, mostly, what separates *explainability* from *interpretability*. This property is also called **transparency** or **translucency**. A model can be interpretable without this, but in the same way as interpreting a person's decisions because we can't understand what is going on "under the hood." This is often called **post-hoc interpretability** and this is the kind of interpretability this book primarily focuses on, with a few exceptions. That being said, we ought to recognize that if a model is understood by leveraging its mathematical formula (grounded in statistical and probability theory), as we've done with linear regression and Naïve Bayes, or by visualizing a human-interpretable structure, as with decision trees, or a set of rules as with RuleFit, it is much more interpretable than machine learning model classes where none of this is practically possible.

White-box models will always have the upper hand in this regard, and as listed in *Chapter 1, Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, there are many use cases in which a white-box model is a must-have. But even if you don't productionize white-box models, they can always serve a purpose in assisting with interpretation, if data dimensionality allows. Transparency is a key property because it wouldn't matter if it didn't comply with the other properties as long as it had explainability; it would still be more interpretable than those without it.

The remedial property: regularization

In this chapter, we've learned that *regularization* limits the complexity added by the introduction of too many features, and this can make the model more interpretable, not to mention more performant. Some models incorporate regularization into the training algorithm, such as RuleFit and gradient-boosted trees; others have the ability to integrate it, such as multi-layer perceptron, or linear regression, and some cannot include it, such as k-NN. Regularization comes in many forms. Decision trees have a method called pruning, which can help reduce complexity by removing non-significant branches. Neural networks have a technique called dropout, which randomly drops neural network nodes from layers during training. Regularization is a remedial property because it can help even the least interpretable models lessen complexity and thus improve interpretability.

Assessing performance

By now, in this chapter, you have already assessed performance on all of the white-box models reviewed in the last section as well as a few black-box models. Maybe you've already noticed that black-box models have topped most metrics, and for most use cases, this is generally the case.

Figuring out which model classes are more interpretable is not an exact science, but the following table (*Figure 3.17*) is sorted by those models with the most desirable properties – that is, they don't introduce non-linearity, non-monotonicity, and interactivity. Of course, explainability on its own is a property that is a game-changer, regardless, and regularization can help. There are also cases in which it's hard to assess properties. For instance, polynomial (linear) regression implements a linear model, but it fits non-linear relationships, which is why it is color-coded differently. As you will learn in *Chapter 12, Monotonic Constraints and Model Tuning for Interpretability*, some libraries support adding monotonic constraints to gradient-boosted trees and neural networks, which means it's possible to make these monotonic. However, the black-box methods we used in this chapter do not support monotonic constraints.

The task columns tell you whether they can be used for regression or classification. And the **Performance Rank** columns show you how well these models ranked in RMSE (for regression) and ROC AUC (for classification), where lower ranks are better. Please note that even though we have used only one metric to assess performance for this chart for simplicity's sake, the discussion about performance should be more nuanced than that. Another thing to note is that ridge regression did poorly, but this is because we used the wrong hyperparameters, as explained in the previous section:

White Box?	Model Class	Properties that Increase Interpretability					Task		Performance Rank	
		🔑 Expl.	Linear	Monotone	Non-Interactive	🖊 Regul.	Regr.	Classif.	Regr.	Classif.
✔	Linear Regression	✔	✔	✔	✔	✔	✔	✖	6	
✔	Regularized Regression	✔	✔	✔	✔	✔	✔	✔	7	8
✔	Logistic Regression	✔	⚠	✔	✔	✔	✖	✔		5
✔	Gaussian Naïve Bayes	✔	✖	✔	✔	✔	✖	✔		7
✔	Polynomial Regression	⚠	⚠	✔	⚠	✔	✔	✔	2	
✔	RuleFit	✔	✔	✖	✖	✔	✔	✔	8	
✔	Decision Tree	✔	✖	⚠	✖	✔	✔	✔	5	3
✔	k-Nearest Neighbors	⚠	✖	✖	✔	✖	✔	✔	9	6
✖	Random Forest	✖	✖	✖	✖	✔	✔	✔	3	4
✖	Gradient Boosted Trees	✖	✖	✖	✖	✔	✔	✔		2
✖	Multi-layer Perceptron	✖	✖	✖	✖	✔	✔	✔	1	1

Figure 3.13: A table assessing the interpretability and performance of several white-hat and black-box models we have explored in this chapter

Because it's compliant with all five properties, it's easy to tell why **linear regression is the gold standard for interpretability**. Also, while recognizing that this is anecdotal evidence, it should be immediately apparent that most of the best ranks are with black-box models. This is no accident! The math behind neural networks and gradient-boosted trees is brutally efficient in achieving the best metrics. Still, as the red dots suggest, they have all the properties that make a model less interpretable, making their biggest strength (complexity) a potential weakness.

This is precisely why black-box models are our primary interest in this book, although many of the methods you will learn to apply to white-box models. In *Part 2*, which comprises *Chapters 4* to *9*, we will learn model-agnostic and deep-learning-specific methods that assist with interpretation. And in *Part 3*, which includes *Chapters 10* to *14*, we will learn how to tune models and datasets to increase interpretability:

	White Box	Glass Box	Black Box
Inherent Interpretability	High	Mid-High	Low
Predictive Performance	Mid	High	High
Execution Speed Performance	High	Low	Mid

Figure 3.14: A table comparing white-box, black-box, and glass-box models, or at least what is known so far about them

Discovering newer interpretable (glass-box) models

In the last decade, there have been significant efforts in both industry and in academia to create new models that can have enough complexity to find the sweet spot between underfitting and overfitting, known as the **bias-variance trade-off**, but retain an adequate level of explainability.

Many models fit this description, but most of them are meant for specific use cases, haven't been properly tested yet, or have released a library or open-sourced code. However, two general-purpose ones are already gaining traction, which we will look at now.

Explainable Boosting Machine (EBM)

EBM is part of Microsoft's InterpretML framework, which includes many of the model-agnostic methods we will use later in the book.

EBM leverages the **GAMs** we mentioned earlier, which are like linear models but look like this:

$$\hat{y} = g(E[y]) = \beta_0 + f_1(x_1) + f_2(x_2) + \dots f_j(x_j)$$

Individual functions f_1 through f_p are fitted to each feature using spline functions. Then a link function g adapts the GAM to perform different tasks such as classification or regression, or adjust predictions to different statistical distributions. GAMs are white-box models, so what makes EBM a glass-box model? It incorporates bagging and gradient boosting, which tend to make models more performant. The boosting is done one feature at a time using a low learning rate so as not to confound them. It also finds practical interaction terms automatically, which improves performance while maintaining interpretability:

$$\hat{y} = g(E[y]) = \beta_0 + \sum f_j(x_j) + \sum f_{ji}(x_j, x_i)$$

Once fitted, this formula is made up of complicated non-linear formulas, so a global holistic interpretation isn't likely feasible. However, since the effects of each feature or pairwise interaction terms are additive, they are easily separable, and global modular interpretation is entirely possible. Local interpretation is equally easy, given that a mathematical formula can assist in debugging any prediction.

One drawback is that EBM can be much slower than gradient-boosted trees and neural networks because of the *one feature at a time* approach, a low learning rate not impacting the feature order, and spline fitting methods. However, it is parallelizable, so in environments with ample resources and multiple cores or machines, it will be much quicker. To avoid waiting for results for an hour or two, it is best to create abbreviated versions of `X_train` and `X_test` – that is, with fewer columns representing only the eight features white-box models found to be most important: `DEP_DELAY`, `LATE_AIRCRAFT_DELAY`, `PCT_ELAPSED_TIME`, `WEATHER_DELAY`, `NAS_DELAY`, `SECURITY_DELAY`, `DISTANCE`, `CRS_ELAPSED_TIME`, and `TAXI_OUT`. These are placed in a `feature_samp` array, and then the `X_train` and `X_test` DataFrames are subset to only include this feature. We are setting the `sample2_size` to 10%, but if you feel you have enough resources to handle it, adjust accordingly:

```
#Make new abbreviated versions of datasets
feature_samp = [
    'DEP_DELAY',
    'LATE_AIRCRAFT_DELAY',
    'PCT_ELAPSED_TIME',
    'DISTANCE',
    'WEATHER_DELAY',
    'NAS_DELAY',
    'SECURITY_DELAY',
    'CRS_ELAPSED_TIME'
]
X_train_abbrev2 = X_train[feature_samp]
X_test_abbrev2 = X_test[feature_samp]

#For sampling among observations
np.random.seed(rand)
sample2_size = 0.1
sample2_idx = np.random.choice(
    X_train.shape[0], math.ceil(
      X_train.shape[0]*sample2_size), replace=False
)
```

To train your EBM, all you have to do is instantiate an `ExplainableBoostingClassifier()` and then fit your model to your training data. Note that we are using `sample_idx` to sample a portion of the data so that it takes less time:

```
ebm_mdl = ExplainableBoostingClassifier()
ebm_mdl.fit(X_train_abbrev2.iloc[sample2_idx], y_train_class.iloc[sample2_idx])
```

Global interpretation

Global interpretation is dead simple. It comes with an `explain_global` dashboard you can explore. It loads with the feature importance plot first, and you can select individual features to graph what was learned from each one:

```
show(ebm_mdl.explain_global())
```

The preceding code generates a dashboard that looks like *Figure 3.15*:

Figure 3.15: EBM's global interpretation dashboard

Local interpretation

Local interpretation uses a dashboard like global does, except you choose specific predictions to interpret with `explain_local`. In this case, we are selecting #76, which, as you can tell, was incorrectly predicted. But the LIME-like plot we will study in *Chapter 5, Local Model-Agnostic Interpretation Methods*, helps us make sense of it:

```
ebm_lcl = ebm_mdl.explain_local(
    X_test_abbrev2.iloc[76:77], y_test_class[76:77], name='EBM'
)
show(ebm_lcl)
```

Similar to the global dashboard, the preceding code generates another one, depicted in *Figure 3.16*:
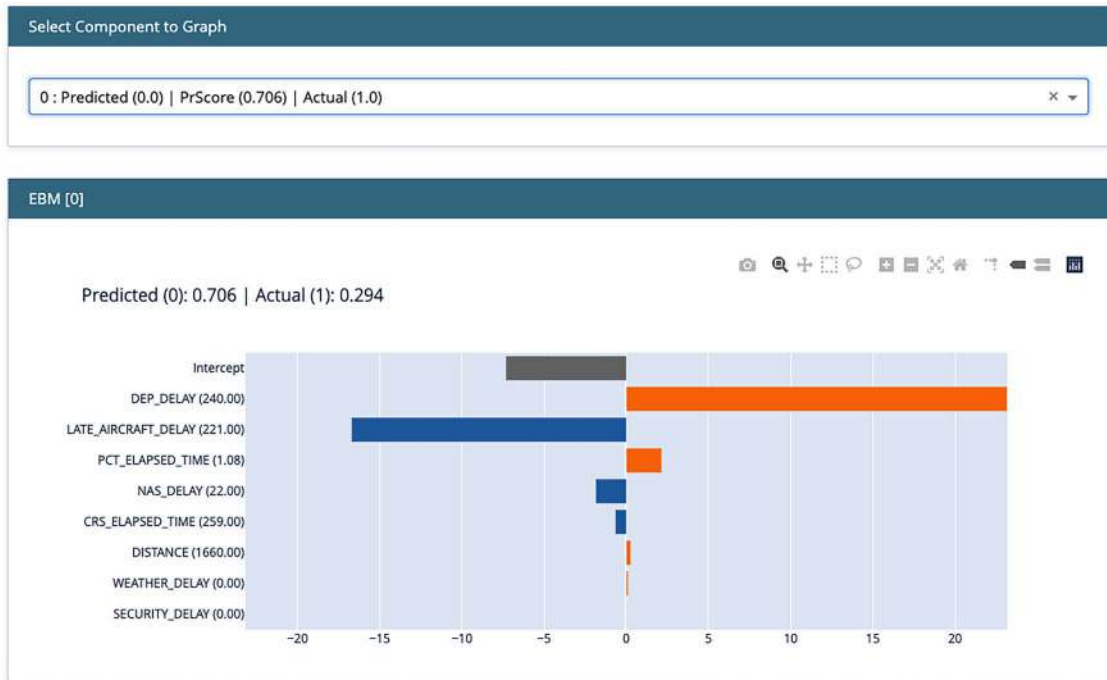
Figure 3.16: EBM's local interpretation dashboard

Performance

EBM performance, at least measured with the ROC AUC, is not far from what was achieved by the top two classification models, and we can only expect it to get better with 10 times more training and testing data!

```
ebm_perf = ROC(ebm_mdl.predict_proba).explain_perf(
    X_test_abbrev2.iloc[sample_idx],y_test_class.iloc[sample_idx],
    name='EBM'
)
show(ebm_perf)
```

You can appreciate the performance dashboard produced by the preceding code in *Figure 3.17*.

Figure 3.17: One of EBM's performance dashboards

The performance dashboard can also compare several models at a time since its explainers are model-agnostic. And there's even a fourth dashboard that can be used for data exploration. Next, we will cover another GAM-based model.

GAMI-Net

There's also a newer GAM-based method with similar properties to EBM but trained with neural networks. At the time of writing, this method has yet to get commercial traction but yields good interpretability and performance.

As we have previously discussed, interpretability is decreased by each additional feature, especially those that don't significantly impact model performance. In addition to too many features, it's also trumped by the added complexity of non-linearities, non-monotonicity, and interactions. GAMI-Net tackles all these problems by fitting non-linear subnetworks for each feature in the main effects network first. Then, fitting a pairwise interaction network with subnetworks for each combination of features. The user provides a maximum number of interactions to keep, which are then fitted to the residuals of the main effects network. See *Figure 3.18* for a diagram.
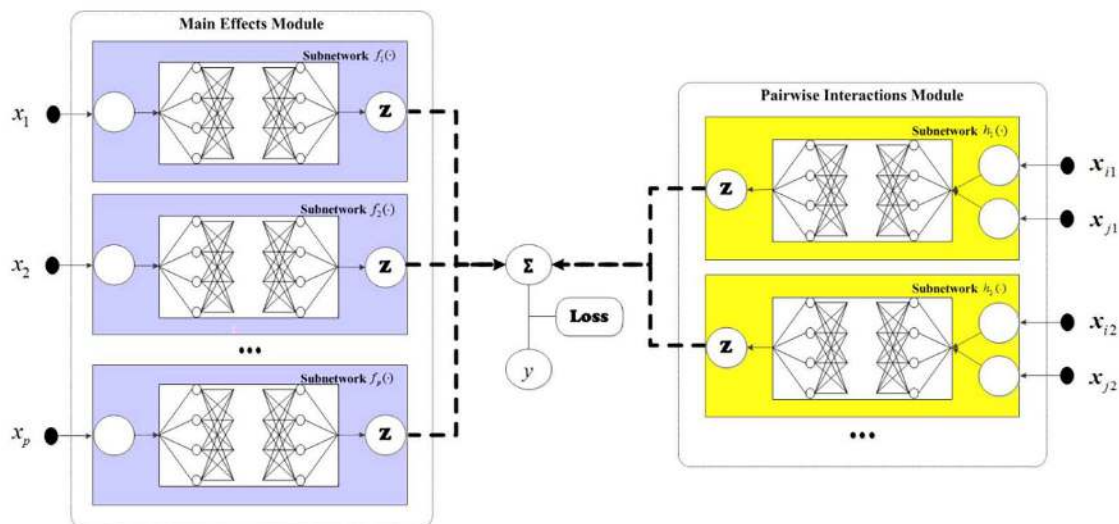
Figure 3.18: Diagram of the GAMI-Net model

GAMI-Net has three interpretability constraints built in:

- **Sparsity**: Only the top features and interactions are kept.
- **Heredity**: A pairwise interaction can be included if at least one of its parent features is included.
- **Marginal clarity**: Non-orthogonality in interactions is penalized to approximate better marginal clarity.

The GAMI-Net implementation can also enforce monotonic constraints, which we will cover in more detail in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*.

Before we start, we must create a dictionary called `meta_info` with details about each feature and target, such as the type (continuous, categorical, and target) and the scaler used to scale each feature — since the library expects each feature to be scaled independently. All the features in the abbreviated dataset are continuous so we can leverage dictionary comprehension to this easily:

```
meta_info = {col:{"type":"continuous"} for col in\
                                    X_train_abbrev.columns}
```

Next, we will create a copy of `X_train_abbrev` and `X_train_abbrev` and then scale them and store the scalers in the dictionary. Then, we will append information about the target variable to the dictionary. And lastly, we will convert all the data to `numpy` format:

```python
X_train_abbrev2 = X_train_abbrev.copy()
X_test_abbrev2 = X_test_abbrev.copy()

for key in meta_info.keys():
    scaler = MinMaxScaler()
    X_train_abbrev2[[key]] =scaler.fit_transform(
      X_train_abbrev2[[key]])
    X_test_abbrev2[[key]] = scaler.transform(X_test_abbrev2[[key]])
    meta_info[key]["scaler"] = scaler
meta_info["CARRIER_DELAY"] = {"type":"target", "values":["no", "yes"]}

X_train_abbrev2 = X_train_abbrev2.to_numpy().astype(np.float32)
X_test_abbrev2 = X_test_abbrev2.to_numpy().astype(np.float32)
y_train_class2 = y_train_class.to_numpy().reshape(-1,1)
y_test_class2 = y_test_class.to_numpy().reshape(-1,1)
```

Now that we have a `meta_info` dictionary and the dataset is ready, we can initialize and fit GAMI-Net to the training data. In addition to `meta_info`, it has a lot of parameters: `interact_num` defines how many top interactions it should consider, and `task_type` defines whether it's a classification or regression task. Note that GAMI-Net trains three neural networks, so there are three epoch parameters to fill in (`main_effect_epochs`, `interaction_epochs`, and `tuning_epochs`). The learning rate (`lr_bp`) and early stopping thresholds (`early_stop_thres`) are entered as a list for each of the epoch parameters. You will also find lists for the architecture of the networks, where each item corresponds to a number of nodes per layer (`interact_arch` and `subnet_arch`). Furthermore, there are additional parameters for batch size, activation function, whether to enforce heredity constraint, a loss threshold for early stopping, and what percentage of the training data to use for validation (`val_ratio`). Finally, there are two optional parameters for monotonic constraints (`mono_increasing_list`, `mono_decreasing_list`):

```python
gami_mdl = GAMINet(
    meta_info=meta_info,
    interact_num=8,
    task_type="Classification",
    main_effect_epochs=80,
    interaction_epochs=60,
    tuning_epochs=40,
    lr_bp=[0.0001] * 3,
    early_stop_thres=[10] * 3,
    interact_arch=[20] * 5,
    subnet_arch=[20] * 5,
```

```
        batch_size=200,
        activation_func=tf.nn.relu,
        heredity=True,
        loss_threshold=0.01,
        val_ratio=0.2,
        verbose=True,
        reg_clarity=1,
        random_state=rand
)

gami_mdl.fit(X_train_abbrev2, y_train_class2)
```

We can plot the training loss for each epoch across all three trainings with `plot_trajectory`. Then, with `plot_regularization`, we can plot the outcome for the regularization of both the main effects and interaction networks. Both plotting functions can save the image in a folder but will do so in a folder called `results` by default, unless you change the path with the `folder` parameter:

```
data_dict_logs = gami_mdl.summary_logs(save_dict=False)
plot_trajectory(data_dict_logs)
plot_regularization(data_dict_logs)
```

The preceding snippet generates the plots in *Figure 3.19*:
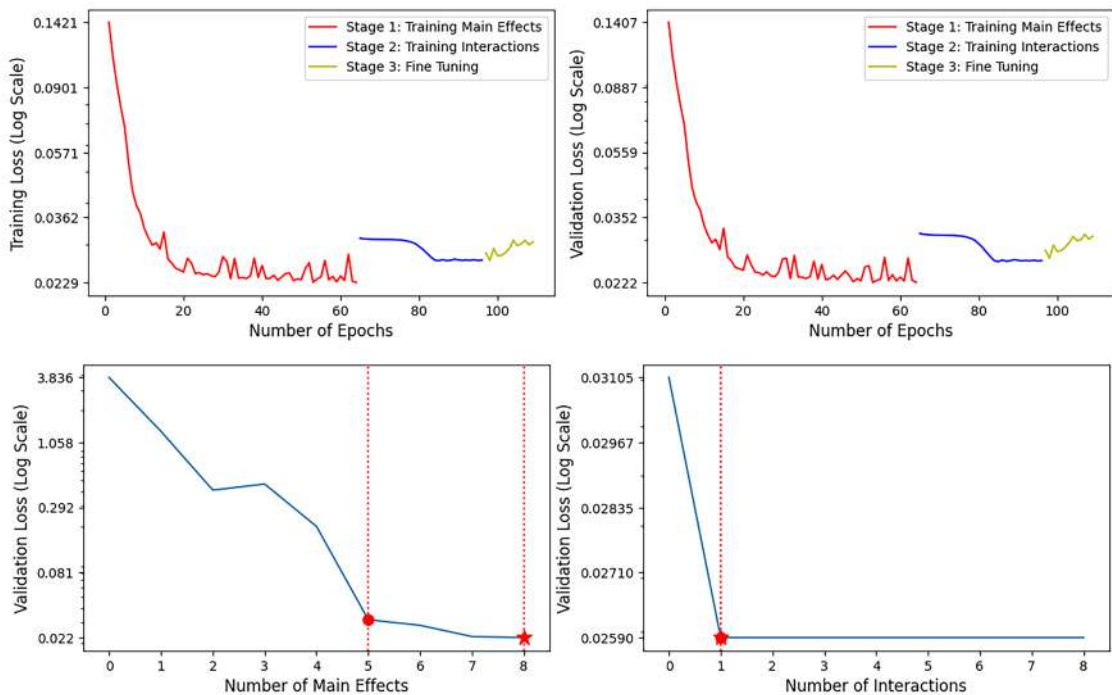
Figure 3.19: The trajectory and regularization plots for the GAMI-net training process

Figure 3.19 tells the story of how the three stages sequentially reduce loss while regularizing to only keep the fewest features and interactions as possible.

Global interpretation

Global explanations can be extracted in a dictionary with the `global_explain` function and then turned into a feature importance plot with `feature_importance_visualize`, like in the following snippet:

```
data_dict_global = gami_mdl.global_explain(save_dict=True)
feature_importance_visualize(data_dict_global)
plt.show()
```

The preceding snippet outputs the following plot:
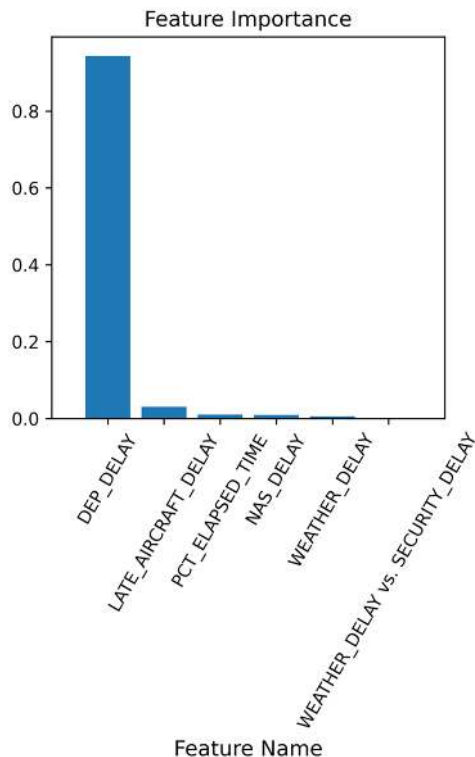
Figure 3.20: A global explanation plot

As you can tell by *Figure 3.20*, the most important feature is, by far, `DEP_DELAY` and one interaction is among the top six features in the plot. We can also use the `global_visualize_density` plot to output partial dependence plots, which we will cover in *Chapter 4, Global Model-Agnostic Interpretation Methods*.

Local interpretation

Let's examine an explanation for a single prediction using `local_explain`, followed by `local_visualize`. We are selecting test case #73:

```
    data_dict_local = gami_mdl.local_explain(
        X_test_abbrev2[[73]], y_test_class2[[73]], save_dict=False
    )
    local_visualize(data_dict_local[0])
    plt.tight_layout()
    plt.show()
```

The preceding code generates the plot in *Figure 3.21*:
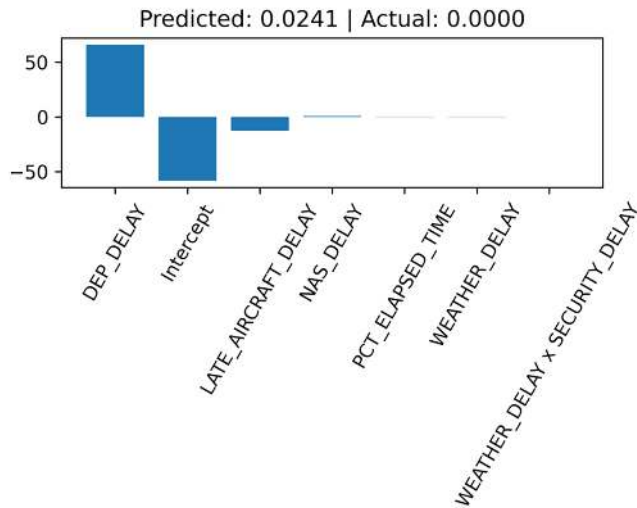
Figure 3.21: A local explanation plot for test case #73

Figure 3.21 tells the story of how each feature weighs in the outcome. Note that `DEP_DELAY` is over 50 but that there's an intercept that almost cancels it out. The intercept is a counterbalance – after all, the dataset is unbalanced toward it being less likely to be a `CARRIER_DELAY`. But all the subsequent features after the intercept are not enough to push the outcome positively.

Performance

To determine the predictive performance of the GAMI-Net model, all we need to do is get the scores (`y_test_prob`) and predictions (`y_test_pred`) for the test dataset and then use scikit-learn's `metrics` functions to compute them:

```
    y_test_prob = gami_mdl.predict(X_test_abbrev2)
    y_test_pred = np.where(y_test_prob > 0.5, 1, 0)
    print(
        'accuracy: %.3g, recall: %.3g, roc auc: %.3g, f1: %.3g,
          mcc: %.3g'
        % (
            metrics.accuracy_score(y_test_class2, y_test_pred),
            metrics.recall_score(y_test_class2, y_test_pred),
```

```
        metrics.roc_auc_score(y_test_class2, y_test_prob),
        metrics.f1_score(y_test_class2, y_test_pred),
        metrics.matthews_corrcoef(y_test_class2, y_test_pred)
    )
)
```

The preceding code yields the following metrics:

```
accuracy: 0.991, recall: 0.934, roc auc: 0.998,
f1: 0.924, mcc: 0.919
```

The performance was not bad considering it was trained on 10% of the training data and evaluated on only 10% of the test data – especially the recall score, which was among the top three places.

Mission accomplished

The mission was to train models that could predict preventable delays with enough accuracy to be useful, and then, to understand the factors that impacted these delays, according to these models, to improve OTP. The resulting regression models all predicted delays, on average, well below the 15-minute threshold according to the RMSE. And most of the classification models achieved an F1 score well above 50% – one of them reached 98.8%! We also managed to find factors that impacted delays for all white-box models, some of which performed reasonably well. So, it seems like it was a resounding success!

Don't celebrate just yet! Despite the high metrics, this mission was a failure. Through interpretation methods, we realized that the models were accurate mostly for the wrong reasons. This realization helps underpin the mission-critical lesson that a model can easily be right for the wrong reasons, so *the question "why?" is not a question to be asked only when it performs poorly but always*. And using interpretation methods is how we ask that question.

But if the mission failed, why is this section called *Mission accomplished?* Good question!

It turns out there was a secret mission. Hint: it's the title of this chapter. The point of it was to learn about common interpretation challenges through the failure of the overt mission. In case you missed them, here are the interpretation challenges we stumbled upon:

- Traditional model interpretation methods only cover surface-level questions about your models. Note that we had to resort to model-specific global interpretation methods to discover that the models were right for the wrong reasons.
- Assumptions can derail any machine learning project since this is information that you suppose without evidence. Note that it is crucial to work closely with domain experts to inform decisions throughout the machine learning workflow, but sometimes they can also mislead you. Ensure you check for inconsistencies between the data and what you assume to be the truth about that data. Finding and correcting these problems is at the heart of what interpretability is about.
- Many model classes, even white-box models, have issues with computing feature importance consistently and reliably.

- Incorrect model tuning can lead to a model that performs well enough but is less interpretable. Note that a regularized model overfits less but is also more interpretable. We will cover methods to address this challenge in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*. Feature selection and engineering can also have the same effect, which you can read about in *Chapter 10*, *Feature Selection and Engineering for Interpretability*.

- There's a trade-off between predictive performance and interpretability. And this trade-off extends to execution speed. For these reasons, this book primarily focuses on black-box models, which have the predictive performance we want and a reasonable execution speed but could use some help on the interpretability side.

If you learned about these challenges, then congratulations! Mission accomplished!

Summary

After reading this chapter, we covered some traditional methods for interpretability and what their limitations are. We learned about **intrinsically interpretable models** and how to both use them and interpret them, for both regression and classification. We also studied the **performance versus interpretability trade-off** and some models that attempt not to compromise in this trade-off. We also discovered many practical interpretation challenges involving the roles of feature selection and engineering, hyperparameters, domain experts, and execution speed.

In the next chapter, we will learn more about different interpretation methods to measure the effect of a feature on a model.

Dataset sources

United States Department of Transportation Bureau of Transportation Statistics. (2018). Airline On-Time Performance Data. Originally retrieved from `https://www.transtats.bts.gov`.
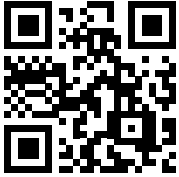
Further reading

- Friedman, J. and Popescu, B, 2008, *Predictive Learning via Rule Ensembles*. The Annals of Applied Statistics, 2(3), 916-954. `http://doi.org/10.1214/07-AOAS148`

- Hastie, T., Tibshirani, R., and Wainwright, M., 2015, *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/Crc Monographs on Statistics & Applied Probability, Taylor & Francis

- Thomas, D.R., Hughes, E., and Zumbo, B.D., 1998, *On Variable Importance in Linear Regression.* Social Indicators Research 45, 253–275: `https://doi.org/10.1023/A:1006954016433`

- Nori, H., Jenkins, S., Koch, P., and Caruana, R., 2019, *InterpretML: A Unified Framework for Machine Learning Interpretability*. arXiv preprint: `https://arxiv.org/pdf/1909.09223.pdf`

- Hastie, T., and Tibshirani, R., 1987, *Generalized Additive Models: Some Applications.* Journal of the American Statistical Association, 82(398):371–386: `http://doi.org/10.2307%2F2289439`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

4

Global Model-Agnostic Interpretation Methods

In the first part of this book, we introduced the concepts, challenges, and purpose of machine learning interpretation. This chapter kicks off the second part, which dives into a vast array of methods that are used to diagnose models and understand their underlying data. One of the biggest questions answered by interpretation methods is: *What matters most to the model and how does it matter?* Interpretation methods can shed light on the overall importance of features and how they—individually or combined—impact a model's outcome. This chapter will provide a theoretical and practical foundation to approach these questions.

Initially, we will explore the notion of feature importance by examining the model's inherent parameters. Following that, we will study how to employ **permutation feature importance** in a model-agnostic manner to effectively, reliably, and autonomously rank features. Finally, we will outline how **SHapley Additive exPlanations** (**SHAP**) can rectify some of the shortcomings of permutation feature importance.

This chapter will look at several ways to visualize global explanations, such as SHAP's bar and beeswarm plots, and then dive into feature-specific visualizations like **Partial Dependence Plots** (**PDP**) and **Accumulated Local Effect** (**ALE**) plots. Lastly, feature interactions can enrich explanations because features often team up, so we will discuss 2-dimensional PDP and ALE plots.

The following are the main topics we are going to cover in this chapter:

- What is feature importance?
- Gauging feature importance with model-agnostic methods
- Using SHAP, PDP, and ALE plots to visualize:
    - Global explanations
    - Feature summary explanations
    - Feature interactions

Technical requirements

This chapter's example uses the `pandas`, `numpy`, `sklearn`, `catboost`, `seaborn`, `matplotlib`, `shap`, `pdpbox`, and `pyale` libraries. Instructions on how to install all of these libraries are in the GitHub repository `README.md` file.

> The code for this chapter is located here: `https://packt.link/Ty0Ev`

The mission

The used car market in the United States is a thriving and substantial industry with significant economic impact. In recent years, approximately 40 million used light vehicles have been sold yearly, representing over two-thirds of the overall yearly sales in the automotive sector. In addition, the market has witnessed consistent growth, driven by the rising cost of new vehicles, longer-lasting cars, and an increasing consumer preference for pre-owned vehicles due to the perception of value for money. As a result, this market segment has become increasingly important for businesses and consumers.

Given the market opportunity, a tech startup is currently working on a machine-learning-driven, two-sided marketplace for used car sales. It plans to work much like the e-commerce site eBay, except it's focused on cars. For example, sellers can list their cars at a fixed price or auction them, and buyers can either pay the higher fixed price or participate in the auction, but how do you come up with a starting point for the price? Typically, sellers define the price, but the site can generate an optimal price that maximizes the overall value for all participants, ensuring that the platform remains attractive and sustainable in the long run.

Optimal pricing is not a simple solution since it has to maintain a healthy balance between the number of buyers and sellers on the platform while being perceived as fair by both buyers and sellers. It remains competitive with other platforms while being profitable. However, it all starts with a price prediction model that can estimate the fair value, and then, from there, it can incorporate other models and constraints to adjust it. To that end, one of the startup's data scientists has obtained a dataset of used car listings from Craigslist and merged it with other sources, such as the U.S. Census Bureau for demographic data and the **Environmental Protection Agency** (**EPA**) for emissions data. The idea is to train a model with this dataset, but we are unsure what features are helpful. And that's where you come in!

You have been hired to explain which features are useful to the machine learning model and why. This is critical because the startup only wants to ask sellers to provide the bare minimum of details about their car before they get a price estimate. Of course, there are details like the car's make and model and even the color that another machine learning model can automatically guess from the pictures. Still, some features like the transmission or cylinders may vary in the car model, and the seller may not know or be willing to disclose them. Limiting the questions asked produces the least friction and thus will lead to more sellers completing their listings successfully.

The approach

You have decided to take the following steps:

1. Train a couple of models.
2. Evaluate them.
3. Create feature importance values using several methods, both model-specific and model-agnostic.
4. Plot global summaries, feature summaries, and feature interaction plots to understand how these features relate to the outcome and each other.

The plots will help you communicate findings to the tech startup executives and your data science colleagues.

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/tree/main/04/UsedCars.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `sklearn` (scikit-learn) and `catboost` to load and configure the model
- `matplotlib`, `seaborn`, `shap`, `pdpbox`, and `pyale` to generate and visualize the model interpretations

You should load all of them first:

```
import math
import os, random
import numpy as np
import pandas as pd
import mldatasets
from sklearn import metrics, ensemble, tree, inspection,\
                    model_selection
import catboost as cb
import matplotlib.pyplot as plt
import seaborn as sns
import shap
from pdpbox import pdp, info_plots
from PyALE import ale
from lime.lime_tabular import LimeTabularExplainer
```

The following snippet of code will load the usedcars dataset:

```
usedcars_df = mldatasets.load("usedcars", prepare=True)
```

You can inspect it using the usedcars_df.info() function and verify that there are indeed over 256,000 rows and 29 columns with no nulls. Some of them are object data types because they are categorical.

The data dictionary for the dataset is as follows:

- Variables related to the listing:

    - price: target, continuous, the price posted for the vehicle
    - region: categorical, the region for the listing—usually this is a city, metropolitan area, or (for more rural areas) a portion of a state or a least populated state (out of 402)
    - posting_date: datetime, the date and time of the posting (all postings are for a single month period in 2021 so you cannot observe seasonal patterns with it)
    - lat: continuous, the latitude in decimal format
    - long: continuous, the longitude in decimal format
    - state: categorical, the two-letter state code (out of 51 including [DC])
    - city: categorical, the name of the city (out of over 6,700)
    - zip: nominal, the zip code (out of over 13,100)

- Variables related to the vehicle:

    - make: categorical, the brand or manufacturer of the vehicle (out of 37)
    - make_cat: categorical, the category for the make (out of 5). It's [obsolete] for makes no longer produced, such as "Saturn," [luxury sports] for brands like "Ferrari," and [luxury] for brands like "Mercedes-Benz." Everything else is [regular] or [premium]. The only difference is that [premium] include brands like "Cadillac" and "Acura," which are the high-end brands of car manufacturers in the [regular] category.
    - make_pop: continuous, the relative popularity of the make in percentiles (0–1)
    - model: categorical, the model (out of over 17,000)
    - model_premier: binary, whether the model is a luxury version/trim of a model (if the model itself is not already high-end such as those in the luxury, luxury sports, or premium categories)
    - year: ordinal, the year of the model (from 1984–2022)
    - make_yr_pop: continuous, the relative popularity of the make for the year it was made for in percentiles (0–1)
    - model_yr_pop: continuous, the relative popularity of the model for the year it was made in percentiles (0–1)
    - odometer: continuous, the reading in the vehicle's odometer
    - auto_trans: binary, whether the car has automatic transmission—otherwise, it is manual

- fuel: categorical, the type of fuel used (out of 5: [gas], [diesel], [hybrid], [electric] and [other])
- model_type: categorical (out of 13: [sedan], [SUV], and [pickup] are the three most popular, by far)
- drive: categorical, whether it's four-wheel, front-wheel, or rear-wheel drive (out of 3: [4wd], [fwd] and [rwd])
- cylinders: nominal, the number of cylinders of the engine (from 2–16). Generally, the more cylinders, the higher the horsepower
- title_status: categorical, what the title says about the status of the vehicle (out of 7 like [clean], [rebuilt], [unknown], [salvage], and [lien])
- condition: categorical, what the owner reported the condition of the vehicle to be (out of 7 like [good], [unknown], [excellent] and [like new])

- Variables related to the emissions of the vehicle:

    - epa_co2: continuous, tailpipe CO2 in grams/mile. For models after 2013, it is based on EPA tests. For previous years, CO2 is estimated using an EPA emission factor (-1 = not vvailable)
    - epa_displ: continuous, the engine displacement (in liters 0.6–8.4)

- Variables related to the ZIP code of the listing:

    - zip_population: continuous, the population
    - zip_density: continuous, the density (residents per sq. mile)
    - est_households_medianincome_usd: continuous, the median household income

Data preparation

We should apply categorical encoding to them so that there's one column per category, but only if the category has at least 500 records. We can do this with the make_dummies_with_limits utility function. But first, let's back up the original dataset:

```python
usedcars_orig_df = usedcars_df.copy()
usedcars_df = mldatasets.make_dummies_with_limits(
    usedcars_df,
    'fuel',
    min_recs=500,
    defcatname='other'
)
usedcars_df = mldatasets.make_dummies_with_limits(
    usedcars_df,
    'make_cat'
)
usedcars_df = mldatasets.make_dummies_with_limits(
    usedcars_df,
```

```
        'model_type',
        min_recs=500,
        defcatname='other'
    )
    usedcars_df = mldatasets.make_dummies_with_limits(
        usedcars_df,
        'condition',
        min_recs=200,
        defcatname='other'
    )
    usedcars_df = mldatasets.make_dummies_with_limits(
        usedcars_df,
        'drive'
    )
    usedcars_df = mldatasets.make_dummies_with_limits(
        usedcars_df,
        'title_status',
        min_recs=500,
        defcatname='other'
    )
```

The above snippet got rid of many `object` columns by turning them into a series of binary columns (of the `uint8` type). However, there are still a few `object` columns left. We can find out which ones like this:

```
    usedcars_df.dtypes[lambda x: x == object].index.tolist()
```

We don't need any of these columns since we have `latitude`, `longitude`, and some of the demographic features, which provide the model with some idea of where the car was being sold. As for the `make` and `model`, we have the `make` and `model` popularity and category features. We can remove the non-numerical features by simply only selecting the numerical ones like this:

```
    usedcars_df = usedcars_df.select_dtypes(
        include=(int,float,np.uint8)
    )
```

The final data preparation steps are to:

1. Define our random seed (`rand`) to ensure reproducibility.
2. Split data into X (features) and y (labels). The former has all columns except for the target variable (`target_col`). The latter is only the target.
3. Lastly, divide both X and y into train and test components randomly using scikit-learn's `train_test_split` function:

```
rand = 42
os.environ['PYTHONHASHSEED']=str(rand)
np.random.seed(rand)
random.seed(rand)
target_col = 'price'
X = usedcars_df.drop([target_col], axis=1)
y = usedcars_df[target_col]
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size=0.25, random_state=rand
)
```

Now we have everything to move forward so we will go ahead with some model training!

Model training and evaluation

The following code snippet will train two classifiers, **CatBoost** and **Random Forest**:

```
cb_mdl = cb.CatBoostRegressor(
    depth=7, learning_rate=0.2, random_state=rand, verbose=False
)
cb_mdl = cb_mdl.fit(X_train, y_train)
rf_mdl =ensemble.RandomForestRegressor(n_jobs=-1,random_state=rand)
rf_mdl = rf_mdl.fit(X_train.to_numpy(), y_train.to_numpy())
```

Next, we can evaluate the CatBoost model using a **regression plot**, and a few metrics. Run the following code, which will output *Figure 4.1*:

```
mdl = cb_mdl
y_train_pred, y_test_pred = mldatasets.evaluate_reg_mdl(
    mdl, X_train, X_test, y_train, y_test
)
```

The CatBoost model produced a high R-squared of 0.94 and a test RMSE of nearly 3,100. The regression plot in *Figure 4.1* tells us that although there are quite a few cases that have an extremely high error, the vast majority of the 64,000 test samples were predicted fairly well. You can confirm this by running the following code:

```
thresh = 4000
pct_under = np.where(
    np.abs(y_test - y_test_pred) < thresh, 1, 0
).sum() / len(y_test)
print(f"Percentage of test samples under ${thresh:,.0f} in absolute
      error {pct_under:.1%}")
```

It says that the percentage of test samples with an absolute error in the $4,000 range is nearly 90%. Granted it's a large margin of error for cars that cost a few thousand, but we just want to get a sense of how accurate the model is. We will likely need to improve it for production, but for now, it will do for the exercise requested by the tech startup:
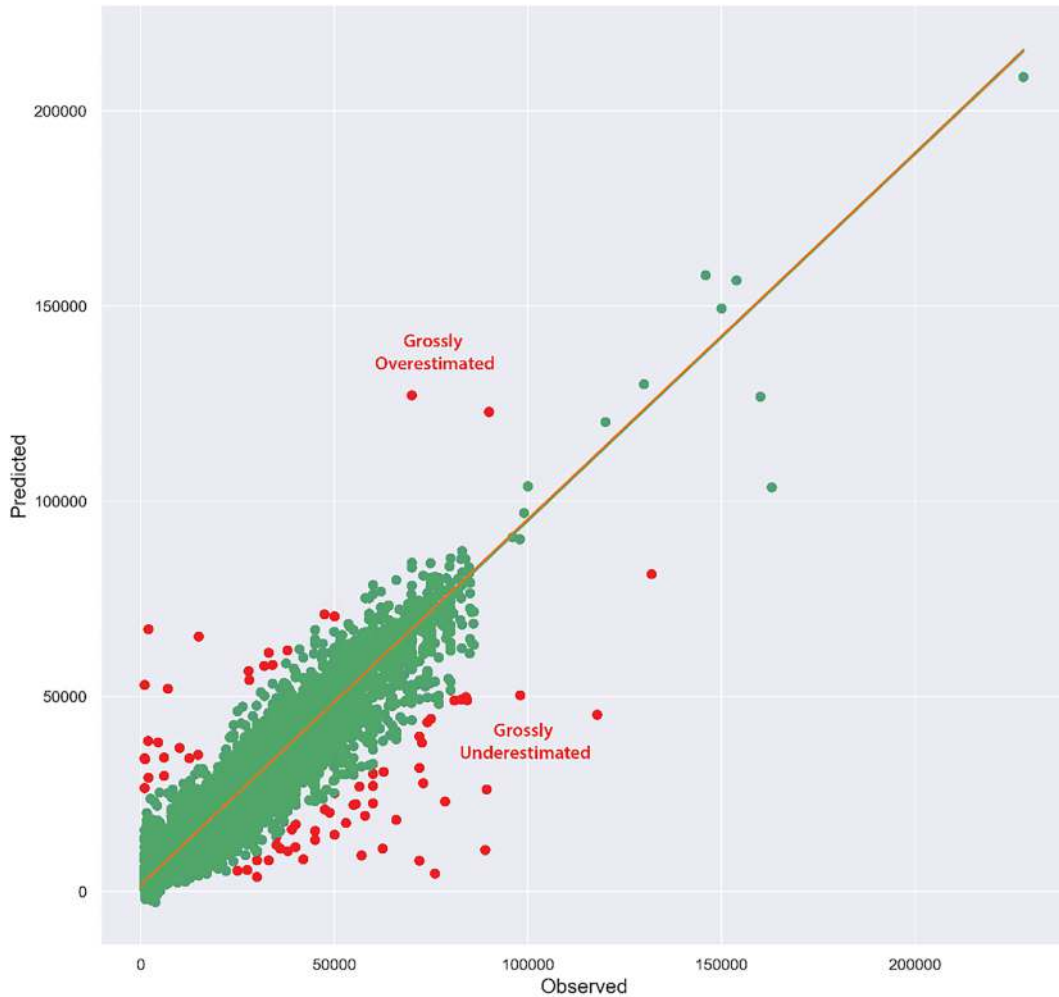

```
RMSE_train: 2543.4672        RMSE_test: 3099.8195        r2: 0.9444
```

Figure 4.1: CatBoost model predictive performance

So now let's repeat the same evaluation for the Random Forest model (`rf_mdl`). To do this, we must run the same snippet but simply replace `cb_mdl` with `rf_mdl` in the first line.

The Random Forest performs just as well for the test samples, and its metrics are remarkably similar to CatBoost, except it overfits much more in this case. This matters because we will now compare some feature importance methods on both and don't want differences in predictive performance to be a reason to doubt any findings.

What is feature importance?

Feature importance refers to the extent to which each feature contributes to the final output of a model. For linear models, it's easier to determine the importance since coefficients clearly indicate the contributions of each feature. However, this isn't always the case for non-linear models.

To simplify the concept, let's compare model classes to various team sports. In some sports, it's easy to identify the players who have the greatest impact on the outcome, while in others, it isn't. Let's consider two sports as examples:

- *Relay race*: In this sport, each runner covers equal distances, and the race's outcome largely depends on the speed at which they complete their part. Thus, it's easy to separate and quantify each racer's contributions. A relay race is similar to a linear model since the race's outcome is a linear combination of independent components.
- *Basketball*: In this game, players have distinct roles, and comparing them using the same metrics is not possible. Moreover, the varying game conditions and player interactions can significantly affect their contributions to the outcome. So, how can we measure and rank each player's contributions?

Models possess inherent parameters that can occasionally aid in unraveling the contributions of each feature. We've trained two models. How have their intrinsic parameters been used to calculate feature importance?

Let's begin with Random Forest. If you plot one of its estimators with the following code, it will generate *Figure 4.2*. Because each estimator is up to six levels deep, we will plot only up to the second level (`max_depth=2`) because, otherwise, the text would be too small. But feel free to increase the `max_depth`:

```
tree.plot_tree(rf_mdl.estimators_[0], filled=True, max_depth=2,\
               feature_names=X_train.columns)
```

Notice the `squared_error` and `samples` in each node of the estimator in *Figure 4.2*. By dividing these numbers, you can calculate the **Mean Squared Error** (**MSE**). Although for classifiers, it's the **Gini coefficient**, for regressors, MSE is the impurity measure. It's expected to decrease as you go deeper in the tree, so the sum of these weighted impurity decreases is calculated for each feature throughout the tree. How much a feature decreases node impurity indicates how much it contributes to the model's outcome. This would be a **model-specific** method since it can't be used with linear models or neural networks, but this is precisely how tree-based models compute feature importance. Random Forest is no different. However, it's an ensemble, so it has a collection of estimators, so the feature importance is the mean decrease in impurity across all estimators.
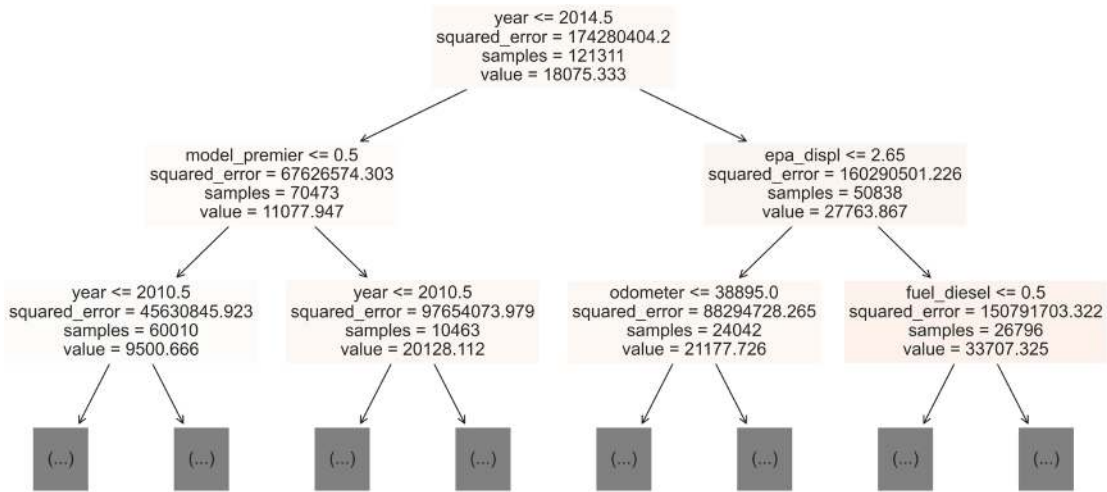
Figure 4.2: First level for the first estimator of the Random Forest model

We can obtain and print the feature importance values for the Random Forest model with the following snippet:

```
rf_feat_imp = rf_mdl.feature_importances_
print(rf_feat_imp)
```

Note that since they have been normalized, they add up to 1 (`sum(rf_feat_imp)`).

CatBoost uses a different method, by default, to compute feature importance, called `PredictionValuesChange`. It shows how much, on average, the model outcome changes if the feature value changes. It traverses the tree performing a weighted average of feature contributions according to the number of nodes in each branch (left or right). If it encounters a combination of features in a node, it evenly assigns contributions to each one. As a result, it can yield misleading feature importance values for features that usually interact with one another.

You can also retrieve the CatBoost feature importances with `feature_importances_` like this, and unlike Random Forest, they add up to 100 and not 1:

```
cb_feat_imp = cb_mdl.feature_importances_
print(cb_feat_imp)
```

Now, let's put them side by side in a pandas `DataFrame`. We will multiply the Random Forest feature importances (`rf_feat_imp`) times 100 to be on the same scale as CatBoost's. Then we will sort by `cb_feat_imp` and format so that there's a nice bar chart embedded in our DataFrame. The result of the following snippet is in *Figure 4.3*:

```
feat_imp_df = pd.DataFrame(
    {
        'feature':X_train.columns,
        'cb_feat_imp':cb_feat_imp,
```

```
        'rf_feat_imp':rf_feat_imp*100
    }
)
feat_imp_df = feat_imp_df.sort_values('cb_feat_imp', ascending=False)
feat_imp_df.style.format(
    '{:.2f}%', subset=['cb_feat_imp', 'rf_feat_imp'])
    .bar(subset=['cb_feat_imp', 'rf_feat_imp'], color='#4EF', width=60
)
```

Notice that both models have the same most important feature in *Figure 4.3*. The top ten features are mostly the same but not in the same rank. In particular, odometer appears to be much more important for CatBoost than for Random Forest. Also, all the other features don't match up in ranking except for the least important ones, for which there's a consensus that they are indeed last:

	feature	cb_feat_imp	rf_feat_imp
4	year	23.24%	45.99%
7	odometer	13.05%	9.41%
11	epa_displ	9.95%	11.79%
15	fuel_diesel	6.16%	3.74%
2	make_pop	5.11%	2.41%
3	model_premier	5.06%	5.91%
9	cylinders	5.06%	2.45%
10	epa_co2	4.03%	1.53%
31	model_type_pickup	3.41%	1.86%
24	make_cat_regular	3.20%	1.86%
44	drive_fwd	2.53%	0.58%
5	make_yr_pop	2.49%	2.36%
6	model_yr_pop	2.34%	1.73%
38	condition_good	1.93%	0.80%
1	long	1.47%	1.05%
0	lat	1.23%	0.83%
	:	:	:
	title_status_other	0.00%	0.00%

Figure 4.3: Compare both models' feature importance values

How can we address these differences and employ a technique that consistently represents a feature's importance? We will explore this using model-agnostic approaches.

Assessing feature importance with model-agnostic methods

Model-agnostic methods imply that we will not depend on intrinsic model parameters to compute feature importance. Instead, we will consider the model as a black box, with only the inputs and output visible. So, how can we determine which inputs made a difference?

What if we altered the inputs randomly? Indeed, one of the most effective methods for evaluating feature importance is through simulations designed to measure a feature's impact or lack thereof. In other words, let's remove a random player from the game and observe the outcome! In this section, we will discuss two ways to achieve this: permutation feature importance and SHAP.

Permutation feature importance

Once we have a trained model, we cannot remove a feature to assess the impact of not using it. However, we can:

- Replace the feature with a static value, such as the mean or median, rendering it devoid of useful information.
- Shuffle (permute) the feature values to disrupt the relationship between the feature and the outcome.

Permutation feature importance (`permutation_importance`) uses the second strategy with a test dataset. Then it measures a change in the score (MSE, r2, f1, accuracy, etc.). In this case, a substantial decrease in the negative **Mean Absolute Error** (**MAE**) when the feature is shuffled would suggest that the feature has a high influence on the prediction. It would have to repeat the shuffling several times (`n_repeats`) to arrive at conclusive results by averaging out the decreases in accuracy. Please note the default scorer for Random Forest regressors is R-squared, while it's RMSE for CatBoost, so we are making sure they both use the same scorer by setting the `scoring` parameter. The following code does all of this for both models:

```
cb_perm_imp = inspection.permutation_importance(
    cb_mdl, X_test, y_test, n_repeats=10, random_state=rand,\
    scoring='neg_mean_absolute_error'
)
```

```
rf_perm_imp = inspection.permutation_importance(
    rf_mdl, X_test.to_numpy(), y_test.to_numpy(), n_repeats=10,\
    random_state=rand, scoring='neg_mean_absolute_error'
)
```

The method outputs a mean score (`importances_mean`) and a standard deviation (`importances_std`) across all repeats for each model, which we can place in a `pandas` DataFrame, sort, and format, as we did before with feature importance. The following code generates the table in *Figure 4.4*:

```
perm_imp_df = pd.DataFrame(
    {
        'feature':X_train.columns,
        'cb_perm_mean':cb_perm_imp.importances_mean,
        'cb_perm_std':cb_perm_imp.importances_std,
        'rf_perm_mean':rf_perm_imp.importances_mean,
        'rf_perm_std':rf_perm_imp.importances_std
    }
)
perm_imp_df = perm_imp_df.sort_values(
    'cb_perm_mean', ascending=False
)
perm_imp_df.style.format(
    '{:.4f}', subset=['cb_perm_mean', 'cb_perm_std', 'rf_perm_mean',
    'rf_perm_std']).bar(subset=['cb_perm_mean', 'rf_perm_mean'],\
    color='#4EF', width=60
)
```

The first four features for both models in *Figure 4.4* are much more consistent than *Figure 4.3*—not that they have to be because they are different models! However, it makes sense considering they were derived from the same method. There are considerable differences too. Random Forest seems to rely much more heavily on fewer features, but these features might not be as necessary if they achieve a very similar predictive performance as CatBoost:

	feature	cb_perm_mean	cb_perm_std	rf_perm_mean	rf_perm_std
4	year	5424.1815	15.1548	6330.7544	19.9509
7	odometer	3014.9332	11.8530	3288.1512	13.0719
11	epa_displ	1330.8259	7.5498	2178.9189	10.0180
9	cylinders	1062.5487	9.9618	1229.9463	10.7112
2	make_pop	997.8885	7.0639	522.7714	4.7237
3	model_premier	578.3021	2.6501	1129.9980	7.5607
10	epa_co2	555.8032	2.9135	719.1720	5.2183
31	model_type_pickup	544.0469	3.3891	533.8197	6.0525
5	make_yr_pop	532.4376	5.3494	484.2530	2.7652
24	make_cat_regular	486.2884	2.6164	709.5006	4.9734
15	fuel_diesel	349.7755	2.8628	519.0072	7.3040
6	model_yr_pop	320.0405	2.3302	531.8455	4.1025
38	condition_good	249.2664	2.2921	264.4274	3.9697
44	drive_fwd	245.5661	3.0388	304.5629	4.0389
1	long	207.2723	2.7572	285.6865	2.4887
32	model_type_sedan	165.2068	2.7577	208.8010	3.5156
43	drive_4wd	140.6390	1.6213	230.8113	3.9377
0	lat	139.9203	1.6919	147.5605	1.4378
20	make_cat_luxury	135.7519	1.9622	108.5608	1.5728
25	model_type_SUV	88.4956	1.4549	115.0064	2.1095
	⋮	⋮	⋮	⋮	⋮
40	condition_new	0.2814	0.0831	0.1344	0.0455

Figure 4.4: Compare both models' permutation feature importance values

Permutation feature importance can be understood as the average increase in model error when a feature is made irrelevant, along with its interactions with other features. It is relatively fast to compute since models don't need to be retrained, but its specific value should be taken with a grain of salt because there are some drawbacks to this shuffling technique:

- Shuffling a highly correlated feature with another unshuffled feature may not significantly affect predictive performance, as the unshuffled feature retains some information from the shuffled one. This means it cannot accurately assess the importance of multicollinear features.

- Shuffling can lead to unrealistic observations, like predicting vehicle traffic with weather and ending up with winter temperatures during the summer. This will result in a higher predictive error for a model that has never encountered such examples, overstating the importance scores beyond their actual significance.

Therefore, these importance values are only useful to rank the features and gauge their relative importance against other features in a particular model. We will now explore how Shapley values can help address these issues.

SHAP values

Before delving into SHAP values, we should discuss **Shapley values**. SHAP is an implementation of Shapley values that takes some liberties but maintains many of the same properties.

It's appropriate that we've been discussing feature importance within the context of games, as Shapley values are rooted in **cooperative game theory**. In this context, players form different sets called **coalitions**, and when they play, they get varying scores known as **marginal contributions**. The Shapley value is the average of these contributions across multiple simulations. In terms of feature importance, players represent features, sets of players symbolize sets of features, and marginal contribution is related to a decrease in predictive error.

This approach may seem similar to permutation feature importance, but its focus on feature combinations rather than individual features helps tackle the multicollinear issue. Moreover, the values obtained through this method satisfy several favorable mathematical properties, such as:

- **Additivity**: the sum of the parts adds to the total value
- **Symmetry**: consistent values for equal contributions
- **Efficiency**: equal to the difference between the prediction and expected value
- **Dummy**: a value of zero for features with no impact on the outcome

In practice, this method demands substantial computational resources. For instance, five features yield $2^5 = 32$ possible coalitions, while 15 features result in 32,768. Therefore, most Shapley implementations use shortcuts like Monte Carlo sampling or leveraging the model's intrinsic parameters (which makes them model-specific). The SHAP library employs various strategies to reduce computational load without sacrificing Shapley properties too much.

Comprehensive explanations with KernelExplainer

Within SHAP, the most prevalent model-agnostic approach is `KernelExplainer`, which is based on **Local Interpretable Model-agnostic Explanations** (**LIME**). Don't fret if you don't understand the specifics since we will cover it in detail in *Chapter 5*, *Local Model-Agnostic Interpretation Methods*. To cut down on computation, it employs sample coalitions. In addition, it follows the same procedures as LIME, such as fitting weighted linear models, but employs Shapley sample coalitions and a different kernel that returns SHAP values as coefficients.

KernelExplainer can be initialized with a kmeans background sample of the training dataset (X_train_summary), which helps it define the kernels. It can be still slow. Therefore, it's better not to use large datasets to compute the shap_values. Instead, in the following code, we are using only 2% of the test dataset (X_test_sample):

```
rf_fn = lambda x: rf_mdl.predict(x)
X_train_summary = shap.kmeans(X_train.to_numpy(), 50)
X_test_sample = X_test.sample(frac=0.02)
rf_kernel_explainer = shap.KernelExplainer(rf_fn, X_train_summary)
rf_shap_values = rf_kernel_explainer.shap_values(
    X_test_sample.to_numpy()
)
```

It might take a while to run the whole thing. Should it take too long, feel free to reduce the sample size from 0.02 to 0.005. SHAP values will be less reliable but it's just an example so you can get a taste of KernelExplainer.

Once it completes, please run print(rf_shap_values.shape) to get an idea of the dimensions we'll be dealing with. Note that it's two dimensions! There's one SHAP value per observation x feature. For this reason, SHAP values can be used for both global and local interpretation. Put a pin in that! We will cover the local interpretation in the next chapter. For now, we will look at another SHAP explainer.

Faster explanations with TreeExplainer

TreeExplainer was designed to efficiently estimate SHAP values for tree-based models such as XGBoost, Random Forest, and CART decision trees. It can allocate non-zero values to non-influential features because it employs the conditional expectation value function rather than marginal expectation, violating the Shapley dummy property. This can have consequences when features are collinear, potentially making explanations less reliable. However, it adheres to other properties.

You can obtain the SHAP values with TreeExplainer like this:

```
cb_tree_explainer = shap.TreeExplainer(cb_mdl)
cb_shap_values = cb_tree_explainer.shap_values(X_test)
```

As you can tell, it's easier and much quicker. It also outputs a two-dimensional array like `KernelExplainer`. You can check with `print(cb_shap_values.shape)`.

For feature importance values, we can collapse two dimensions into one. All we need to do is average the absolute value per feature like this:

```python
cb_shap_imp = np.mean(np.abs(cb_shap_values),0)
```

For Random Forest, just replace the `cb_` for `rf_` with the same code.

We can now compare both SHAP feature importances side-by-side using a formatted and sorted `pandas` DataFrame. The following code will generate the table in *Figure 4.5*.

```python
shap_imp_df = pd.DataFrame(
    {
        'feature':X_train.columns,
        'cb_shap_imp':cb_shap_imp,
        'rf_shap_imp':rf_shap_imp
    }
)
shap_imp_df = shap_imp_df.sort_values('cb_shap_imp', ascending=False)
shap_imp_df.style.format(
    '{:.4f}', subset=['cb_shap_imp', 'rf_shap_imp']).bar(
    subset=['cb_shap_imp', 'rf_shap_imp'], color='#4EF', width=60
)
```

Figure 4.5 not only compares feature importance for two different models but also for two different SHAP explainers. They aren't necessarily perfect depictions, but they are both to be trusted more than permutation feature importance:

feature	cb_shap_imp	rf_shap_imp
year	5374.6635	5704.8741
odometer	3691.2966	2837.3699
epa_displ	1347.3157	1935.6400
cylinders	1125.1109	636.3444
model_premier	1123.7732	534.7291
make_pop	865.3036	376.7030
model_type_pickup	861.0232	532.9261
drive_fwd	837.2566	313.6196
make_cat_regular	767.6872	348.1303
fuel_diesel	676.5713	332.9367
epa_co2	506.1507	392.3129
long	471.7182	319.1998
model_type_sedan	382.8636	300.3917
model_yr_pop	324.9925	414.3527
make_yr_pop	311.5293	280.4556
condition_good	280.9659	132.0956
lat	224.7691	108.0771
make_cat_luxury	204.4589	52.7512
drive_4wd	186.8555	134.9422
zip_density	140.3340	80.8189
⋮	⋮	⋮
condition_new	1.0416	0.0869

Figure 4.5: Compare both models' SHAP feature importance values

SHAP for both models suggest that `loan_to_value_ratio` and `make_cat_va` importances were previously deflated. This makes sense because `loan_to_value_ratio` is highly correlated with several of the top features and `make_cat_va` with all the other product-type features.

Visualize global explanations

Previously, we covered the concept of global explanations and SHAP values. But we didn't demonstrate the many ways we can visualize them. As you will learn, SHAP values are very versatile and can be used to examine much more than feature importance!

But first, we must initialize a SHAP explainer. In the previous chapter, we generated the SHAP values using `shap.TreeExplainer` and `shap.KernelExplainer`. This time, we will use SHAP's newer interface, which simplifies the process by saving SHAP values and corresponding data in a single object and much more! Instead of explicitly defining the type of explainer, you initialize it with `shap.Explainer(model)`, which returns the callable object. Then, you load your test dataset (`X_test`) into the callable `Explainer`, and it returns an `Explanation` object:

```
cb_explainer = shap.Explainer(cb_mdl)
cb_shap = cb_explainer(X_test)
```

In case you are wondering, how did it know what kind of explainer to create? Glad you asked! There's an optional parameter called `algorithm` in the initialization function, and you can explicitly define `tree`, `linear`, `additive`, `kernel`, and others. But by default, it is set to `auto`, which means it will guess which kind of explainer is needed for the model. In this case, CatBoost is a tree ensemble, so `tree` is what makes sense. We can easily check that SHAP chose the right explainer with `cb_explainer.[dict]{custom-style="P - Italics"}[ or ]print(type(cb_explainer))`. It will return `<class 'shap.explainers._tree.Tree'>`, which is correct! As for the explanation stored in `cb_shap`, what is it exactly? It's an object that contains pretty much everything that is needed to plot explanations, such as the SHAP values (`cb_shap.values`) and corresponding dataset (`cb_shap.data`). They should be exactly the same dimensions because there's one SHAP value for every data point. It's easy to verify this by checking their dimensions with the `shape` property:

```
print("Values dimensions: %s" % (cb_shap.values.shape,))
print("Data dimensions: %s" % (cb_shap.data.shape,))
```

Now, let's put these values to some use!

SHAP bar plot

Let's start with the most straightforward global explanation visualizations we can do, which is feature importance. You can do this with a bar chart (`shap.plots.bar`). All it needs is the explanation object (`cb_shap`), but by default, it will only display 10 bars. Fortunately, we can change this with `max_display`:

```
shap.plots.bar(cb_shap, max_display=15)
```

The preceding snippet will generate *Figure 4.6*:
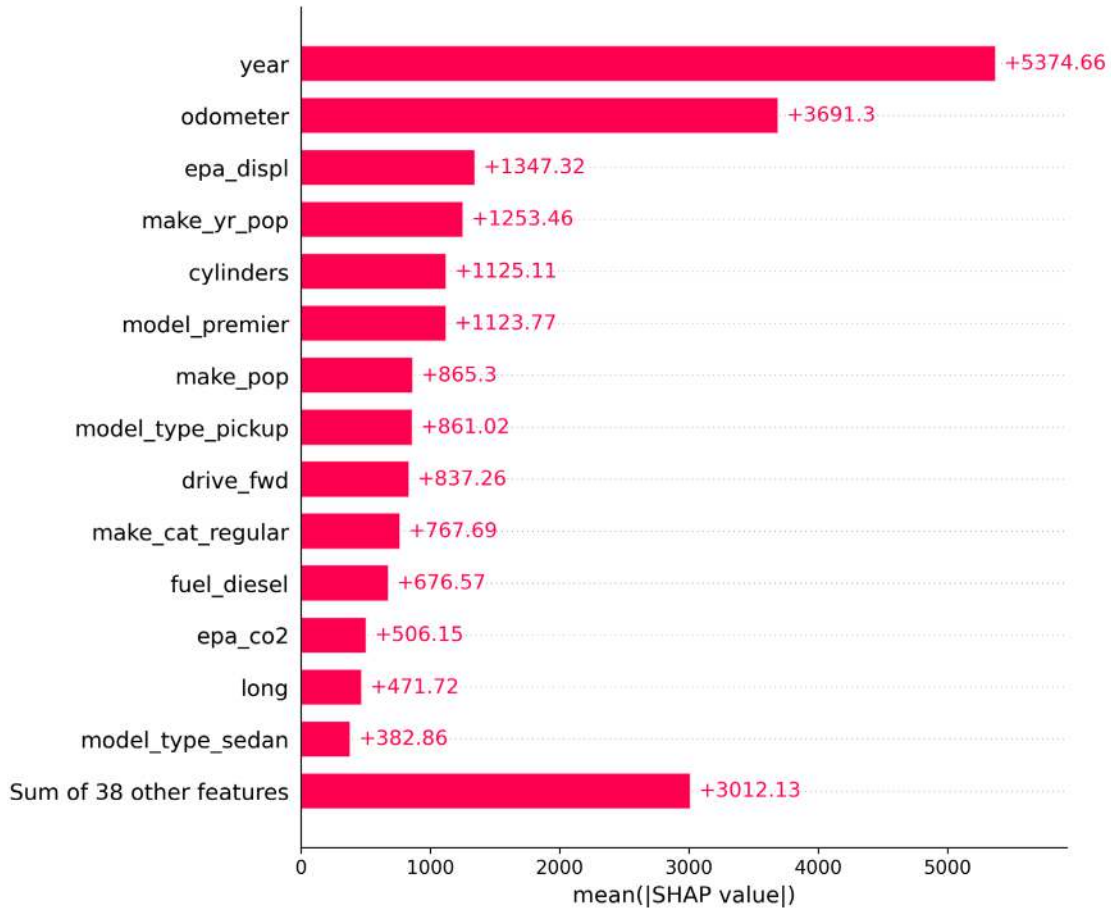
Figure 4.6: CatBoost model's SHAP feature importance

Figure 4.6 should look very familiar if you read the previous chapter. In fact, it should match the `cb_shap_imp` column in *Figure 4.5*.

SHAP feature importance offers considerable flexibility since it is simply the average of the absolute value of SHAP values for each feature. With the granularity of SHAP values, you can dissect them in the same way as the test dataset to obtain insights across various dimensions. This reveals more about feature importance than a single average for each feature.

For example, you can compare feature importance across different groups. Suppose we want to explore how feature importance differs between `year` cohorts. First, we need a threshold to define. Let's use 2014 because it's the median `year` in our dataset. Values above that can be set in the "Newer Car" cohort and pre-2014 values set to "Older Car." Then, use `np.where` to create an array assigning the cohorts to each observation. To create the bar chart, repeat the previous process but use the cohorts function to split the explanation, applying the absolute value (`abs`) and `mean` operations to each cohort.

```
yr_cohort = np.where(
    X_test.year > 2014, "Newer Car", "Older Car"
)
shap.plots.bar(cb_shap.cohorts(yr_cohort).abs.mean(0))
```

This code snippet produces *Figure 4.7*:
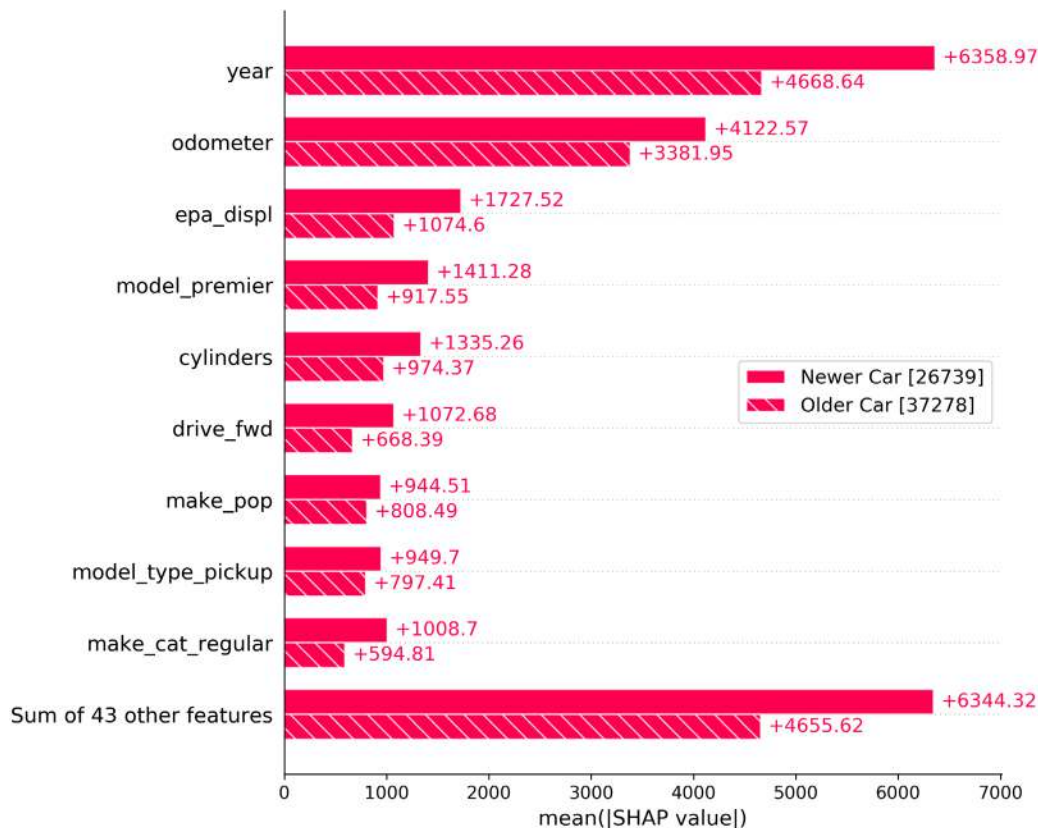
Figure 4.7: CatBoost model's SHAP feature importance split by property value cohorts

As you can see in *Figure 4.7*, all the top features matter less for "Older cars." One of the biggest differences is with year itself. When a car is older, how much older doesn't matter as much as when it's in the "Newer Car" cohort.

SHAP beeswarm plot

Bar charts can obscure some aspects of how features affect model outcomes. Not only do different feature values have varying impacts, but their distribution across all observations can also exhibit considerable variation. The beeswarm plot aims to provide more insight by using dots to represent each observation for every individual feature, even though features are ordered by global feature importance:

- Dots are color-coded based on their position in the range of low to high values for each feature.
- Dots are arranged horizontally according to their impact on the outcome, centered on the line where SHAP value = 0, with negative impacts on the left and positive impacts on the right.
- Dots accumulate vertically, creating a histogram-like visualization to display the number of observations influencing the outcome at every impact level for each feature.

To better understand this, we'll create a beeswarm plot. Generating the plot is easy with the `shap.plots.beeswarm` function. It only requires the explanation object (`cb_shap`), and, as with the bar plot, we'll override the default `max_display` to show only 15 features:

```
shap.plots.beeswarm(cb_shap, max_display=15)
```

The result of the preceding code is in *Figure 4.8*:
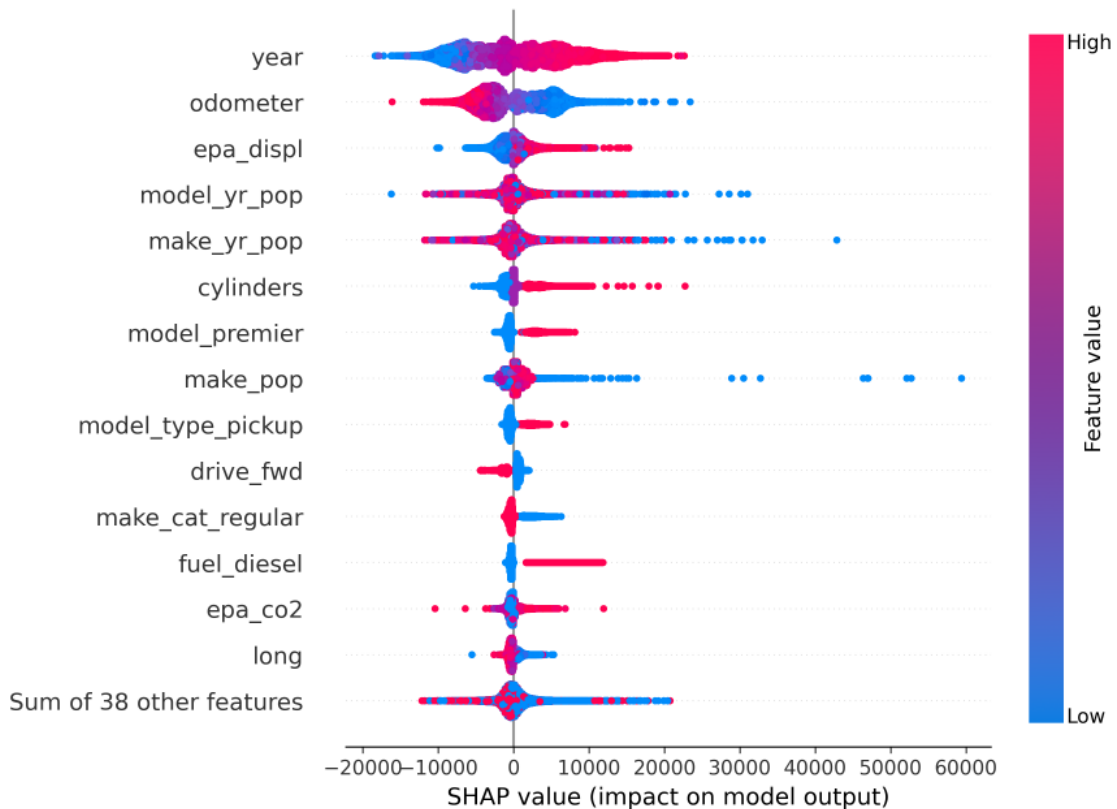
Figure 4.8: The CatBoost model's SHAP beeswarm plot

Figure 4.8 can be read as follows:

- The most important feature (of the 15) from top to bottom is `year` and the least is longitude (`long`). It should match the same order in the bar chart or if you take the average absolute values of the SHAP values across every feature.

- Low values for `year` negatively impact the model outcome, while high values impact it positively. There's a nice clean gradient in between, suggesting an increasing monotonic relationship between `year` and the predicted `price`. That is, the higher the `year`, the higher the `price` according to the CatBoost model.

- `odometer` has a negative or negligible impact on a sizable portion of the observations. However, it has a long tail of observations for which it has a sizable impact. You can tell this by looking at the density depicted vertically.

- If you scan the rest of the plot looking for other continuous features, you won't find such a clean gradient anywhere else as you did for `year` and `odometer` but you'll find some trends like higher values of `make_yr_pop` and `model_yr_pop` having mostly a negative impact.

- For binary features, it is easy to tell because you only have two colors, and they are sometimes neatly split, as with `model_premier`, `model_type_pickup`, `drive_fwd`, `make_cat_regular`, and `fuel_diesel`, demonstrating how a certain kind of vehicle might be a tell-tale sign for the model whether it is high priced or not. In this case, a pick-up model increases the price, whereas a vehicle with a regular make—that is, a brand that is not luxury—decreases the price.

While the beeswarm plot offers an excellent summary of many findings, it can sometimes be challenging to interpret, and it doesn't capture everything. The color-coding is useful for illustrating the relationship between feature values and model output, but what if you want more detail? That's where the partial dependence plot comes in. It's among the many feature summary explanations that provide a global interpretation method specific to features.

Feature summary explanations

This section will cover a number of methods used to visualize how an individual feature impacts the outcome.

Partial dependence plots

Partial Dependence Plots (**PDPs**) display a feature's relationship with the outcome according to the model. In essence, the PDP illustrates the marginal effect of a feature on the model's predicted output across all possible values of that feature.

The calculation involves two steps:

1. Initially, conduct a simulation where the feature value for each observation is altered to a range of different values, and predict the model using those values. For example, if the `year` varies between 1984 and 2022, create copies of each observation with `year` values ranging between these two numbers. Then, run the model using these values. This first step can be plotted as the **Individual Conditional Expectation** (**ICE**) plot, with simulated values for `year` on the X-axis and the model output on the Y-axis, and one line per simulated observation.

2. In the second step, simply average the ICE lines to obtain a general trend line. This line represents the PDP!

PDPs can be created using scikit-learn but these only work well with scikit-learn models. They also can be generated with the SHAP library, as well as another library called PDPBox. Each has its advantages and disadvantages, which we will cover in this section.

SHAP's `partial_dependence` plot function takes the name of a feature (`year`), a `predict` function (`cb_mdl.predict`), and a dataset (`X_test`). There are some optional parameters such as whether to show the ICE lines (`ice`), a horizontal model expected value line (`model_expected_value`), and a vertical feature expected value line (`feature_expected_value`). It shows the `ice` lines by default, but there are so many observations in the test dataset it would take a long time to generate the plot, and would be "too busy" to appreciate the trends. The SHAP PDP can also incorporate SHAP values (`shap_values=True`) but it would take a very long time considering the size of the dataset. It's best to sample your dataset to make it more plot-friendly:

```
shap.plots.partial_dependence(
    "year", cb_mdl.predict, X_test, ice=False, model_expected_value=True,\
    feature_expected_value=True
)
```

The preceding code will produce the plot in *Figure 4.9*:
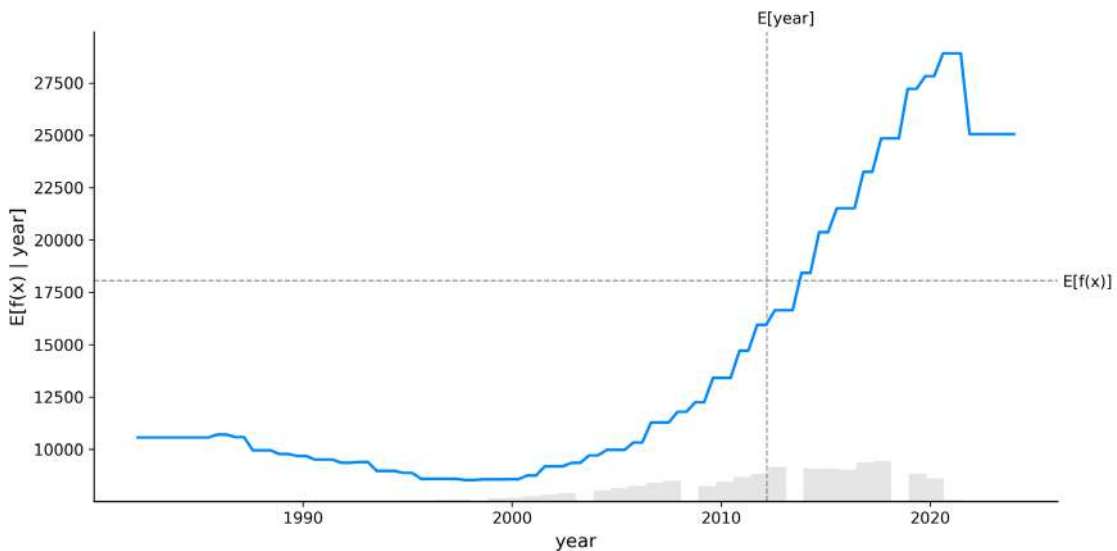
Figure 4.9: SHAP's PDP for year

As you can appreciate in *Figure 4.9*, there's an upward trend for year. This finding shouldn't be surprising considering the neat gradient in *Figure 4.8* for *year*. With the histogram, you can tell the bulk of observations have values for year over 2011, which is where it starts to have an above-verage impact on the model. This makes sense once you compare the location of the histogram to the location of the bulge in the beeswarm plot (*Figure 4.8*).

With PDPBox, we will make several variations of PDP plots. This library separates the potentially time-consuming process of making the simulations with the PDPIsolate function from the plotting with the plot function. We will only have to run PDPIsolate once but plot three times.

For the first plot, we use plot_pts_dist=True to display a rug. The rug is a more concise way of conveying the distribution than the histogram.

For the second one, we use plot_lines=True to plot the ICE lines, but we can only plot a fraction of them, so frac_to_plot=0.01 randomly selects 1% of them.

For the third one, instead of showing a rug, we can construct the X-axis with quantiles (to_bins=True):

```python
pdp_single_feature = pdp.PDPIsolate(
    model=cb_mdl, df=X_test, model_features=X_test.columns,\
    feature='year', feature_name='year', n_classes=0,\
    n_jobs=-1)
fig, axes = pdp_single_feature.plot(plot_pts_dist=True)
fig.show()
    fig, axes = pdp_single_feature.plot(
    plot_pts_dist=True, plot_lines=True, frac_to_plot=0.01
)
fig.show()
    fig, axes = pdp_single_feature.plot(
    to_bins=True, plot_lines=True, frac_to_plot=0.01
)
fig.show()
```

The preceding snippet will output the three plots in *Figure 4.10*:
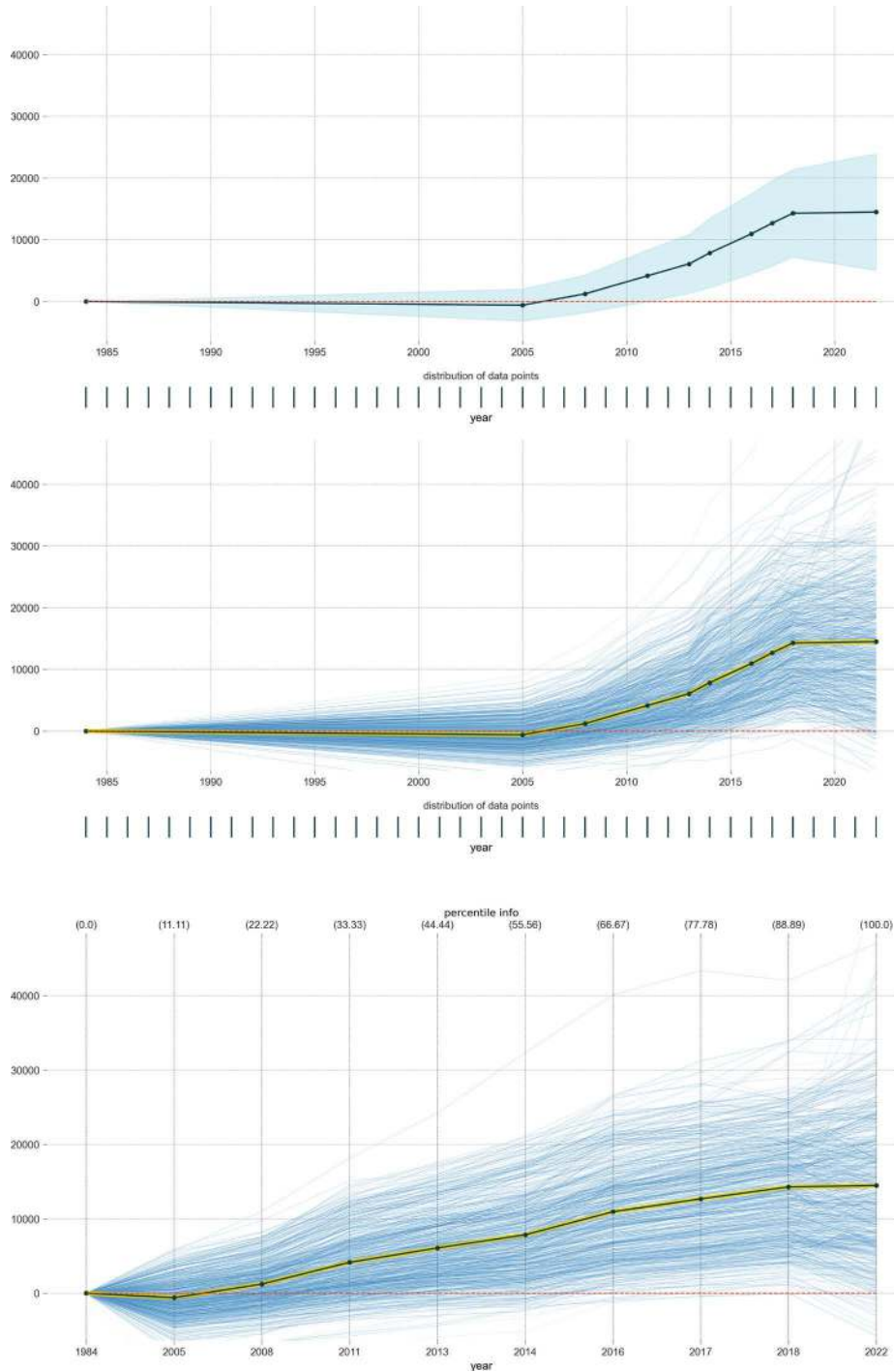
Figure 4.10: Three variations of PDPBox's PDPs for year

The ICE lines enrich the PDP plots by showing the potential for variance. The last plot in *Figure 4.10* also shows how, even though rugs or histograms are useful guides, organizing the axis in quantiles better helps visualize distribution. In this case, two-thirds of year is distributed before 2017. And the two decades between 1984 and 2005 only account for 11%. It's only fair they get a corresponding portion of the plot.

We can now create a few lists we will use to iterate across different kinds of features, whether continuous (cont_feature_l), binary (bin_feature_l), or categorical (cat_feature_l):

```
cont_feature_l = [
    'year', 'odometer', 'cylinders', 'epa_displ', 'make_pop',\
    'make_yr_pop', 'model_yr_pop'
]
make_cat_feature_l = [
    'make_cat_obsolete', 'make_cat_regular', 'make_cat_premium',\
    'make_cat_luxury', 'make_cat_luxury_sports'
]
bin_feature_l = ['model_premier', 'auto_trans']
cat_feature_l = make_cat_feature_l + bin_feature_l
```

To quickly get a sense of what PDPs look like for each feature, we can iterate across one of the lists of features producing PDP plots for each one. We will do continuous features (cont_feature_l) because it's easiest to visualize, but you can try one of the other lists as well:

```
for feature in cont_feature_l:
    pdp_single_feature = pdp.PDPIsolate(
        model=cb_mdl, df=X_test, model_features=X_test.columns,\
        feature=feature, feature_name=feature, n_classes=0,\
        n_jobs=-1
    )
    fig, axes = pdp_single_feature.plot(
        to_bins=True, plot_lines=True, frac_to_plot=0.01,\
        show_percentile=True, engine='matplotlib'
    )
```

The preceding code will output eight plots, including the one in *Figure 4.11*:
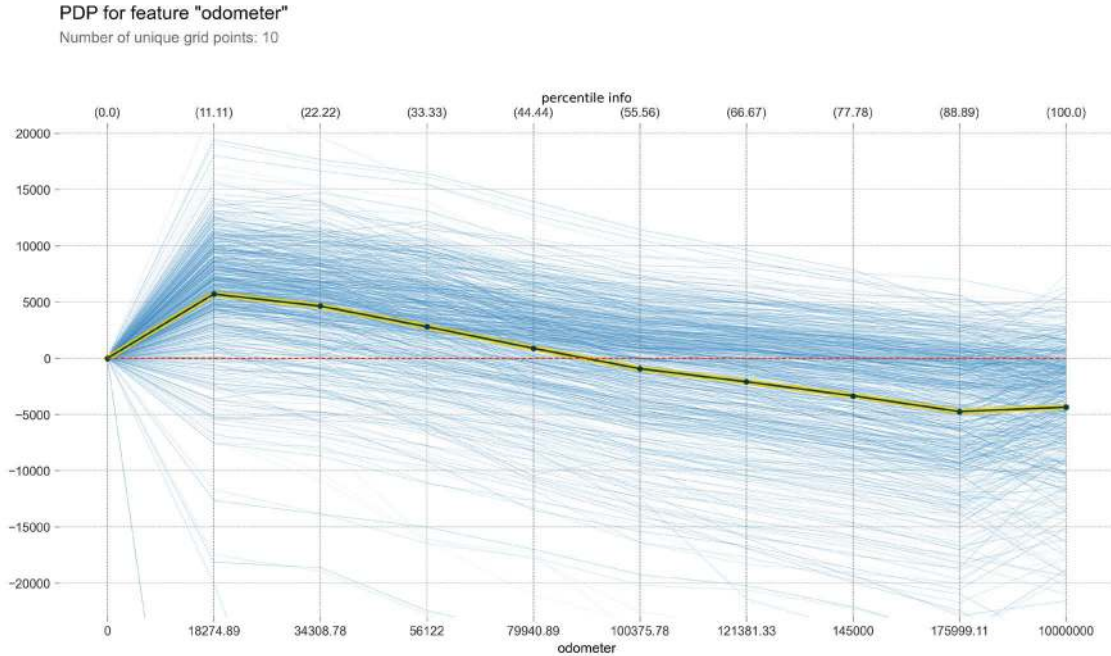
Figure 4.11: PDPBox's PDP for odometer

The beeswarm plot in *Figure 4.8* shows that a low value of odometer correlates with the model outputting a higher price. In *Figure 4.11*, it depicts how it's mostly monotonically decreasing except in the extremes. It's interesting that there are odometer values of zero and ten million in the extremes. While it makes sense that the model learned that odometer made no difference in prices when it's zero, a value of ten million is an anomaly, and for this reason, you can tell that the ICE lines are going in different directions because the model is not sure what to make of such a value. We must also keep in mind that ICE plots, and thus PDPs, are generated with simulations that may create examples that wouldn't exist in real life, such as a brand-new car with an extremely high odometer value.

If you replace `cont_feature_l` with `make_cat_feature_l` and re-run the previous snippet, you will realize that some make categories correlate positively while others negatively with the outcome. This finding shouldn't be surprising considering some make categories, such as "luxury" are indicative of a high price and others of a lower one. That being said, `make_cat` is a categorical feature that was one-hot encoded, so it's not natural to create PDP plots as if each category were an independent binary feature.

Fortunately, you can create a PDP with each one-hot encoded category side by side. All you need to do is plug in the list of product-type features in the `feature` attribute. PDPBox also has a "predict plot," which can help provide context by showing prediction distribution across feature values. `PredictPlot` is easy to plot, having many of the same attributes as the `plot`.

```
pdp_multi_feature = pdp.PDPIsolate(
    model=cb_mdl, df=X_test, model_features=X_test.columns,\
    feature=make_cat_feature_l, feature_name="make_cat", n_classes=0, n_jobs=-1
)
fig, axes = pdp_multi_feature.plot(
    plot_lines=True, frac_to_plot=0.01, show_percentile=True
)
fig.show()


predict_plot = info_plots.PredictPlot(
    model=cb_mdl, df=X_test, feature_name="make_cat", n_classes=0,
    model_features=X_test.columns, feature=make_cat_feature_l
)
fig, _, _ = predict_plot.plot()
fig.show()
```

The preceding snippet should create the two plots in *Figure 4.12*:

PDP for feature "make_cat"
Number of unique grid points: 5

Actual predictions plot for make_cat
Distribution of actual prediction through different feature values.
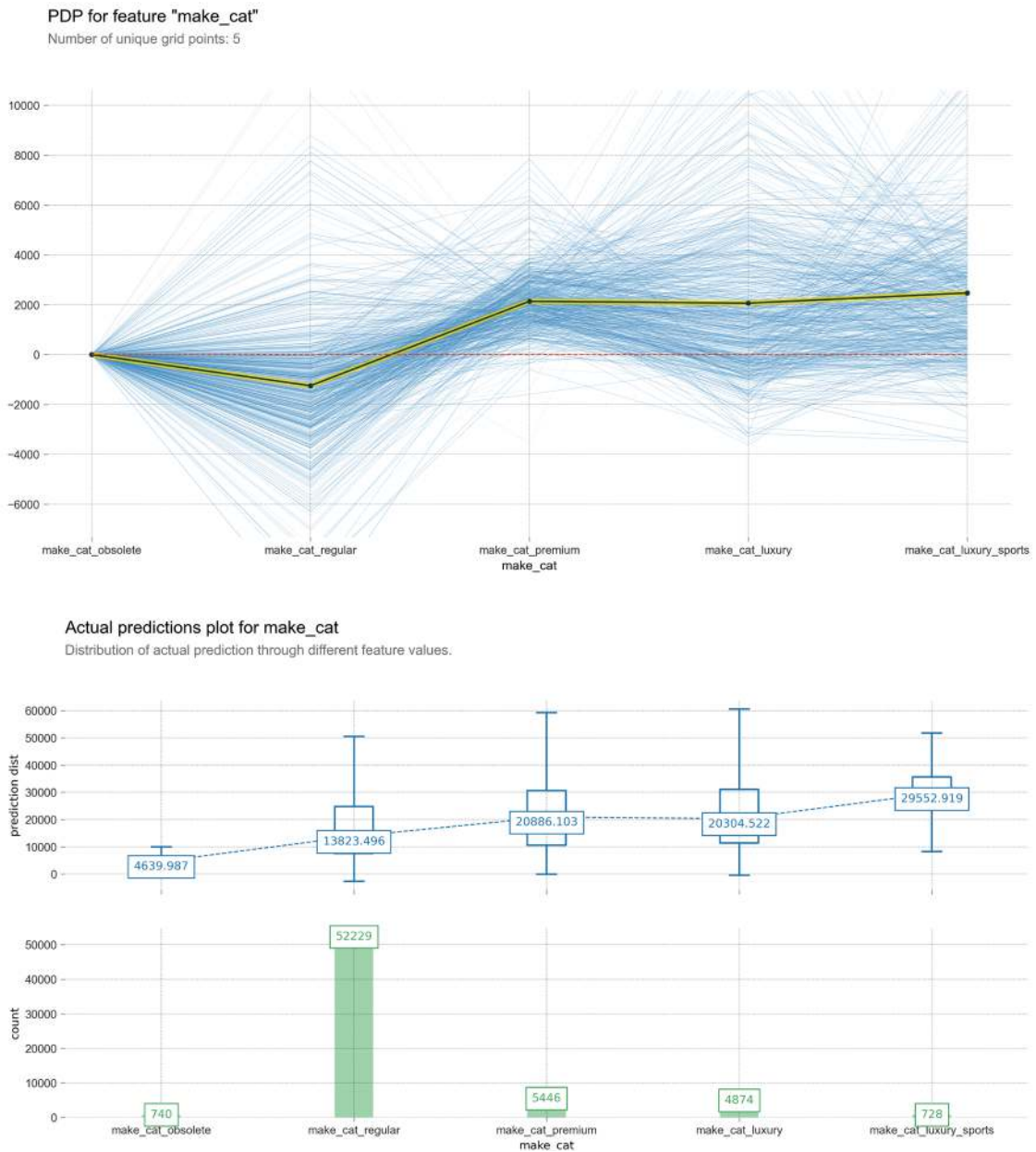
Figure 4.12: PDPBox's actual plot for make categories

The PDP for `make_cat` in *Figure 4.12* shows the tendency of "luxury" and "premium" categories to lead to higher prices but only by about 3,000 dollars on average with a lot of variance depicted in the ICE lines. However, remember what was said about simulations not being necessarily representative of real-life scenarios?

If we compare the PDP to the box and whiskers in the actual predictions in *Figure 4.12*, we can tell that the difference in average prediction between regular and any of the "luxury" and "premium" categories is at least seven thousand dollars. Of course, the average doesn't tell the whole story because even a premium car with too much mileage or that's very old might cost less than a regular vehicle. The price depends on many factors besides the reputation of the make. Of course, just judging by how the boxes and whiskers align in *Figure 4.12*, "luxury sports" and "obsolete" categories are stronger indications of higher and lower prices, respectively.

PDPs are often straightforward to interpret and relatively quick to generate. However, the simulation strategy it employs doesn't account for feature distribution and heavily relies on the assumption of feature independence, which can lead to counterintuitive examples. We will now explore two alternatives.

SHAP scatter plot

SHAP values are available for every data point, enabling you to plot them against feature values, resulting in a PDP-like visualization with model impact (SHAP values) on the *y*-axis and feature values on the *x*-axis. Being a similar concept, the SHAP library initially called this a `dependence_plot`, but now it's referred to as a scatter plot. Despite the similarities, PDP and SHAP values are calculated differently.

Creating a SHAP scatter plot is simple, requiring only the explanation object. Optionally, you can color-code the dots according to another feature to understand potential interactions. You can also clip outliers from the *x*-axis using percentiles with `xmin` and `xmax` attributes and make dots 20% opaque (`alpha`) to identify sparser areas more easily:

```
shap.plots.scatter(
    cb_shap[:,"odometer"], color=cb_shap[:,"year"], xmin="percentile(1)",\
    xmax="percentile(99)", alpha=0.2
)


shap.plots.scatter(
    cb_shap[:,"long"], color=cb_shap[:,"epa_displ"],\
    xmin="percentile(0.5)", xmax="percentile(99.5)", alpha=0.2, x_jitter=0.5
)
```

The above snippet creates *Figure 4.13*:
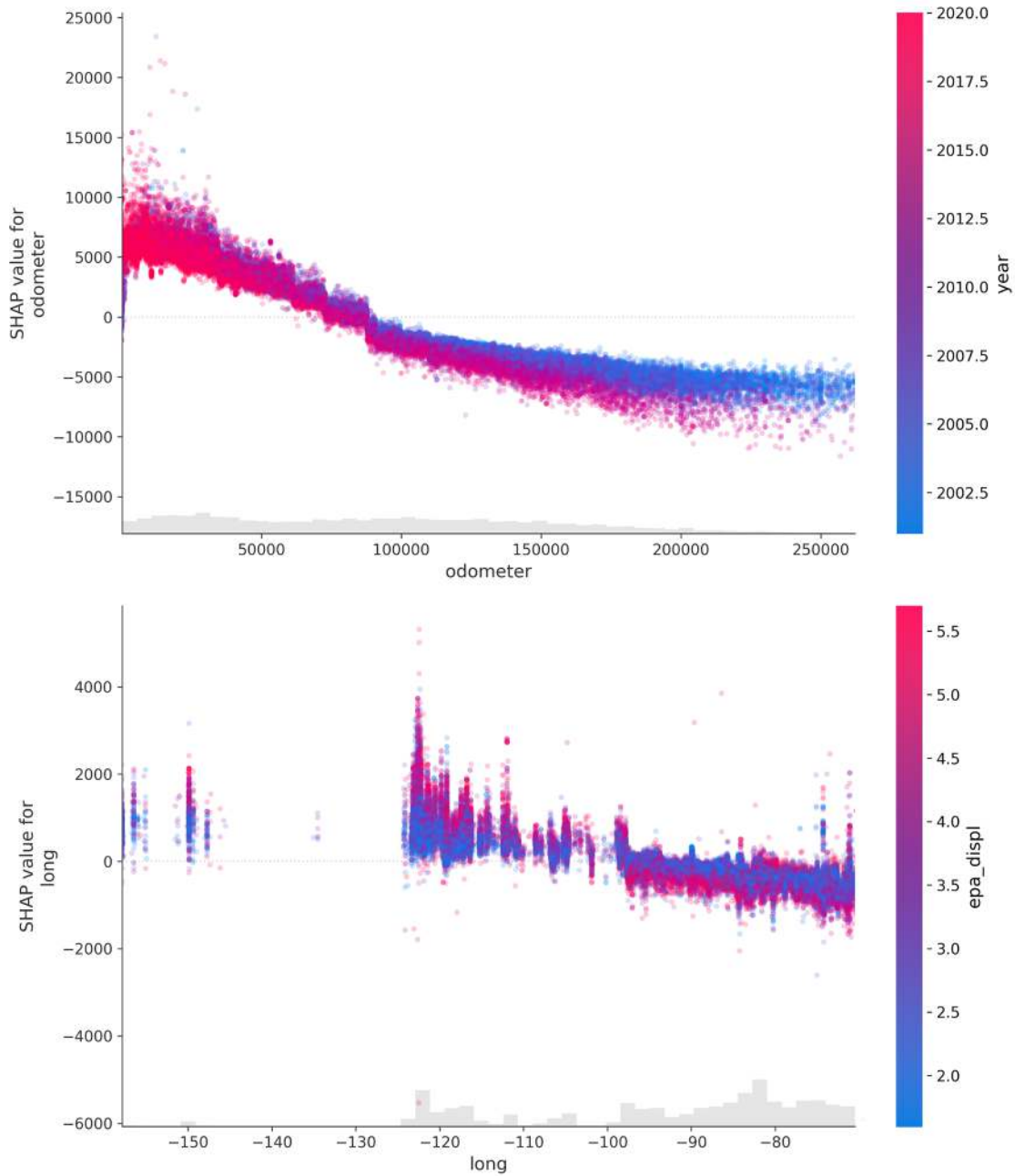
Figure 4.13: SHAP's scatter plots for odometer and long, color-coded for year and epa_displ respectively

The first plot in *Figure 4.13* depicts how a higher odometer reading negatively impacts the model outcome. Also, the color coding shows that the SHAP values for higher years are even lower when the odometer reading is over ninety thousand. In other words, an old car with a high odometer reading is expected, but with a new car, it's a red flag!

The second plot in *Figure 4.13* is very interesting; it shows how the west coast of the country (at about -120°) correlates with higher SHAP values and the further east it goes, the lower the SHAP values. Hawaii, Anchorage, and Alaska (at about -150°) are also higher than the east coast of the United States (at around -75°). The color coding shows how the more liters displaced of fuel mostly leads to higher SHAP values but it's not as stark of a difference the further east you go.

The scatter plot works well for continuous features, but can you use it for discrete ones? Yes! Let's create one for make_cat_luxury. Since there are only two possible values on the *x*-axis, 0 and 1, what makes sense is to jitter them so that all the dots don't get plotted on top of each other. For instance, x_jitter=0.4 means they will be jittered horizontally up to 0.4, or 0.2 on each side of the original value. We can combine this with alpha to ensure that we can appreciate the density:

```
shap.plots.scatter(
    cb_shap[:,"make_cat_luxury"], color=cb_shap[:,"year"], x_jitter=0.4,\
    alpha=0.2, hist=False
)
```

The snippet above produces *Figure 4.14*:
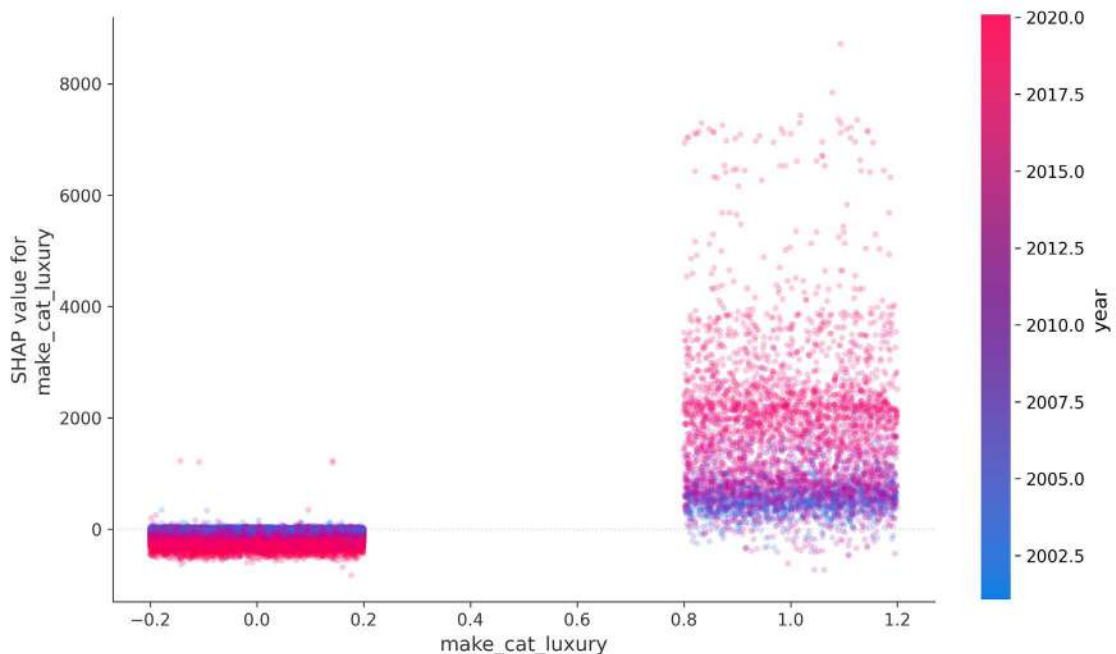
Figure 4.14: SHAP's scatter plot for make_cat_luxury color-coded for year

Figure 4.14 shows that according to the SHAP values, `make_cat_fha=1` positively impacts the model outcome, whereas `make_cat_fha=1` has a mildly negative effect. The color coding suggests that lower years tempers the impact, making it smaller. This makes sense because older luxury cars have depreciated.

While SHAP scatter plots may be an improvement from PDP plots, SHAP's tree explainers trade fidelity for speed, causing features that have no influence on the model to potentially have a SHAP value above zero. Fidelity refers to the accuracy of an explanation in representing the behavior of the model. To get higher fidelity, you need to use a method that makes fewer shortcuts in understanding what the model is doing, and even then, some adjustments to parameters like using a greater sample size will increase fidelity too because you are using more data to create the explanations. In this case, the solution is to use `KernelExplainer` instead because, as we previously discussed, it is more comprehensive, but it has issues with feature dependence. So there's no free lunch! Next, we will cover ALE plots as a partial solution to these problems.

ALE plots

ALE plots are advantageous over PDPs because they are unbiased and faster. ALE accounts for data distributions when calculating feature effects, resulting in an unbiased representation. The algorithm divides the feature space into equally sized windows and computes how predictions change within these windows, resulting in the *local effects*. Summing the effects across all windows makes them *accumulated*.

You can easily generate the ALE plot with the `ale` function. All you need is some data (`X_test_no_outliers`), the model (`cb_mdl`), and the feature(s) and feature type(s) to be plotted. Optionally, you can enter the `grid_size`, which will define the window size for local effects calculations. It is `20` by default, but we can increase it for higher fidelity, provided you have enough data. As mentioned previously, some adjustments to parameters can affect fidelity. For window size, it will break the data into smaller bins to compute values and thus make them more granular. Also, confidence intervals are shown by default. Incidentally, it's best to remove outliers in this case because it's hard to appreciate the plot when the maximum loan nearly reaches \$8 million. You can try using `X_test` instead of `X_test_no_outliers` to see what I mean.

Confidence intervals are shown by default. It's preferable to exclude outliers in this case, as it's difficult to appreciate the plot when only very few vehicles represent years before 1994 and after 2021 and have very low and very high odometer readings. You can use `X_test` instead of `X_test_no_outliers` in the `ale` function to see the difference:

```
X_test_no_outliers = X_test[
    (X_test.year.quantile(.01) <= X_test.year) &\
    (X_test.year <= X_test.year.quantile(.99)) &\
    (X_test.odometer.quantile(.01) <= X_test.odometer) &\
    (X_test.odometer <= X_test.odometer.quantile(.99))
]
ale_effect = ale(
    X=X_test_no_outliers, model=cb_mdl, feature=['odometer'],\
    feature_type='continuous', grid_size=80
)
```

```
plt.show()

ale_effect = ale(
    X=X_test_no_outliers, model=cb_mdl,\
    feature=['make_cat_luxury'],feature_type='discrete'
)
plt.show()
```

The above snippet produces the two plots in *Figure 4.15*:
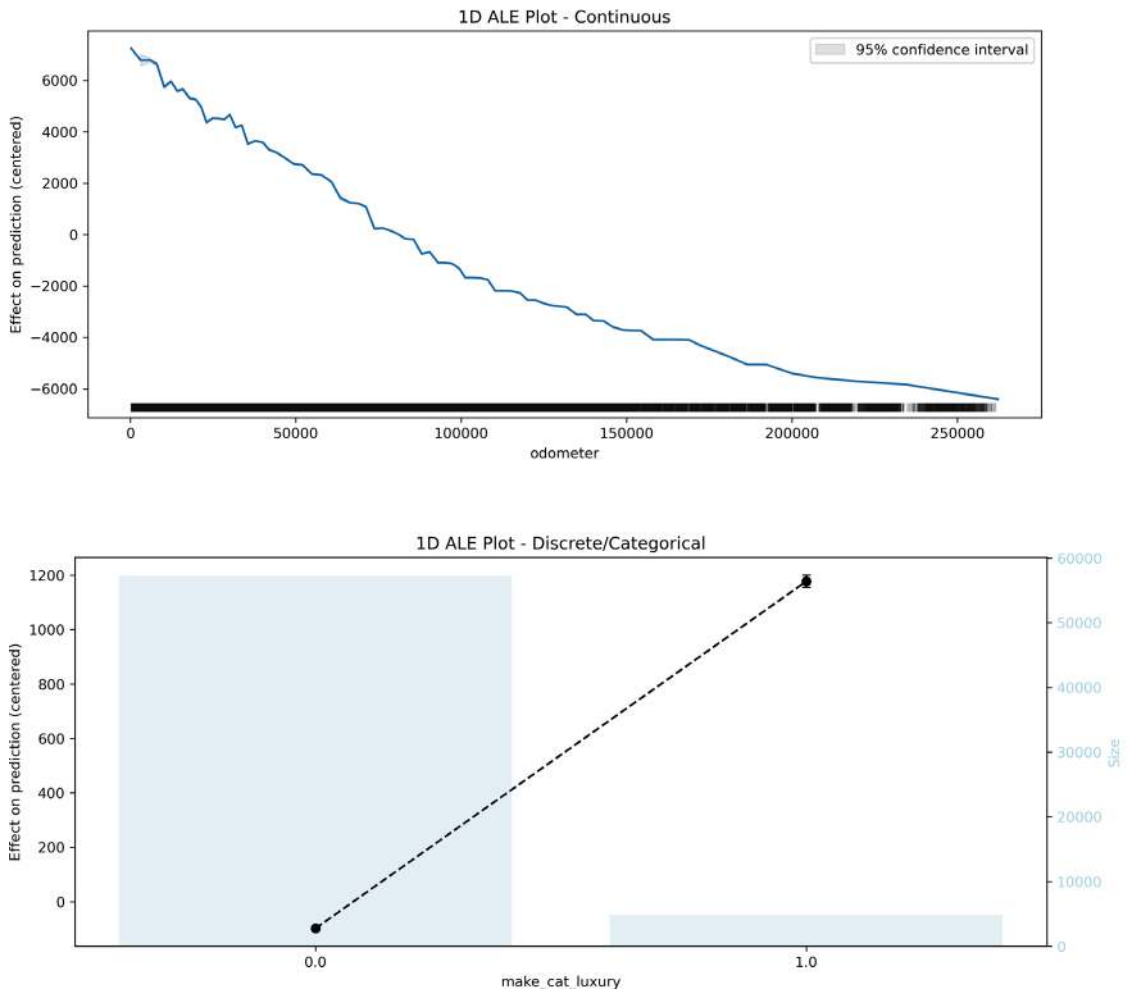
Figure 4.15: ALE plots for odometer and make_cat_luxury

The first plot in *Figure 4.15* is the ALE plot for `odometer`. As portrayed in *Figure 4.13*, the effect on the model goes from positive to negative as the odometer reading increases. However, unlike SHAP, in the ALE plot, it goes negative long before the odometer reaches 90,000. Please note the confidence interval is so thin it's only visible somewhere under 10,000. The second ALE plot shows a significant positive impact of the "luxury" category on the outcome and that luxury vehicles are not as represented as other ones.

So far, we've only discussed single-feature explanations. But we also can observe the interaction between features, which we will cover next.

Feature interactions

Features may not influence predictions independently. For example, as discussed in *Chapter 2*, *Key Concepts of Interpretability*, determining obesity based solely on weight isn't possible. A person's height or body fat, muscle, and other percentages are needed. Models understand data through correlations, and features are often correlated because they are naturally related, even if they are not linearly related. Interactions are what the model may do with correlated features. For instance, a decision tree may put them in the same branch, or a neural network may arrange its parameters in such a way that it creates interaction effects. This also occurs in our case. Let's explore this through several feature interaction visualizations.

SHAP bar plot with clustering

SHAP comes with a hierarchical clustering method (`shap.utils.hclust`) that allows for the grouping of training features based on the "redundancy" between any given pair of features. This refers to the degree to which they depend on each other, on a scale from complete redundancy (0) to total independence (1). We won't use the entire dataset for this task because it would take a very long time, so we will use a 10% `sample`:

```
X_samp = X.sample(frac=0.1)
y_samp = y.loc[X_samp.index]
clustering = shap.utils.hclust(X_samp, y_samp)
```

We can employ the same bar chart as in *Figure 4.6*, but this time, we input the `clustering` and clustering cutoff and voilá! We can visualize which features are most redundant. Our aim is to pinpoint relationships that are less than 0.7 independent (`clustering_cutoff`):

```
shap.plots.bar(cb_shap, clustering=clustering, clustering_cutoff=0.7)
```

The preceding snippet generates the bar plot in *Figure 4.16*:
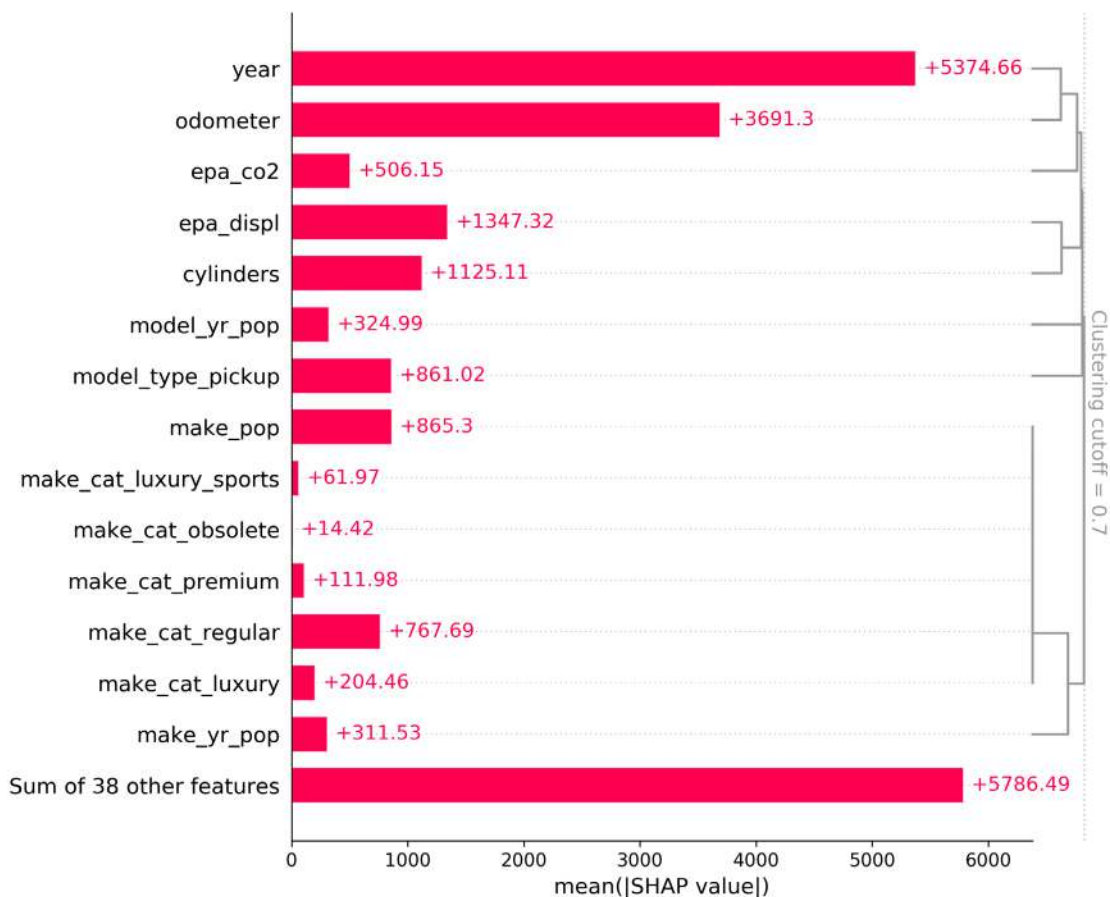
Figure 4.16: SHAP bar plot with clustering

The dendrogram on the right in *Figure 4.16* reveals which features rely on each other the most and at three tiers. For instance, year is dependent on odometer, and when combined, they both depend on epa_co2, which in turn depends indirectly on a number of features, including the fuel displacement (epa_displ) and cylinders features. Also, please note that all make category features depend on each other but also on the make's relative popularity (make_pop). This finding makes sense because some categories are overall more popular than others. These insights can serve as a guide for subsequent investigations.

2D ALE plots

The optimal way to visually inspect the impact of two variables on predictions is through 2D ALE plots, primarily because it's unbiased when dealing with correlated features.

Let's scrutinize the top features, year and odometer, which also have a clear dependency. We'll utilize the test dataset minus the outliers so that the plot focuses on the core of the data—that is, the part containing approximately 98% of the data points. This time, instead of a single feature, we'll insert a list of two features:

```
features_l = ["year", "odometer"]
ale_effect = ale(
    X=X_test_no_outliers, model=cb_mdl, feature=features_l,\
    feature_type='continuous', grid_size=50, include_CI=False
)
plt.show()
```

The above code produces the ALE plot in *Figure 4.17*:
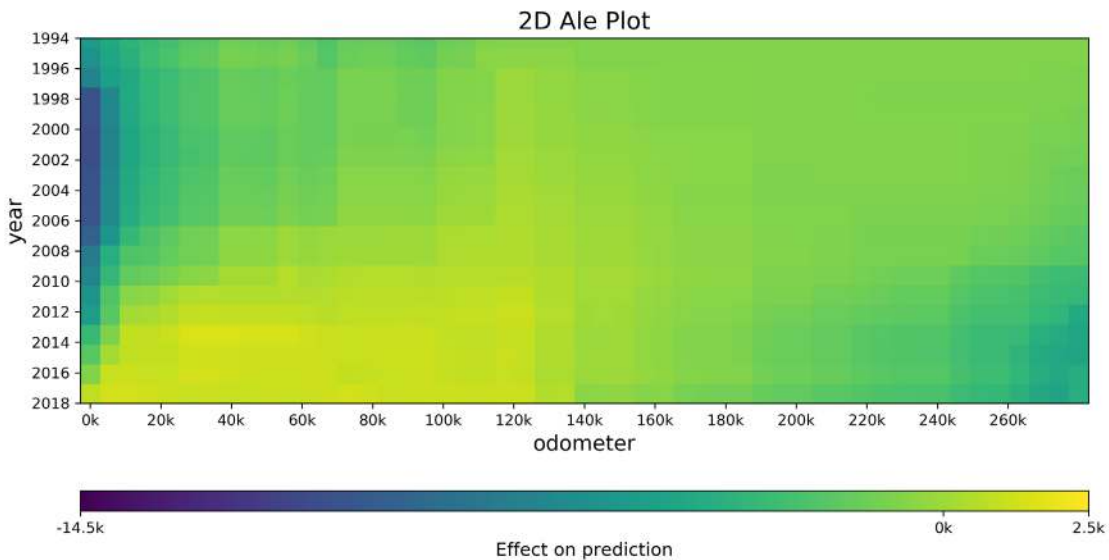
Figure 4.17: 2D ALE plot for odometer and year

As seen in *Figure 4.17*, except for a slim range of extremely high odometer readings and areas where older cars and low odometer readings overlap, there's a modest effect for most of the plot. Mostly, they seem to only have a larger negative effect in the corners.

An important point to remember is that SHAP's clustering distance ranges from redundancy to independence. The issue with highly correlated features is that, at a certain point, they cease depending on each other and become entirely redundant. So, let's examine how close to redundancy these two features are with the clustering array we created with shap.utils.hclust:

```
np.set_printoptions(suppress=True)
print(clustering)
```

The preceding snippet should output the below array:

```
[[   2.      21.       0.        2.     ]
 [  22.      52.       0.        3.     ]
 [  23.      53.       0.        4.     ]
 [  24.      54.       0.        5.     ]
 [  20.      55.       0.        6.     ]
 [  12.      14.       0.235     2.     ]
 [   0.      57.       0.253     3.     ]
 [   1.      58.       0.262     4.     ]
 [  13.      59.       0.289     5.     ]
 [   4.       7.       0.383     2.     ]
 [   9.      11.       0.394     2.     ]
 [  43.      44.       0.429     2.     ]
 [  15.      17.       0.462     2.     ]
 [   5.      56.       0.477     7.     ]
 [  10.      61.       0.605     3.     ]
 [  46.      49.       0.647     2.     ]
 [  62.      66.       0.654     5.     ]
 [  51.      67.       0.665     3.     ]
 [   6.      68.       0.67      6.     ]
 [  31.      70.       0.673     7.     ]
 [  65.      71.       0.694    14.     ]
 [  25.      32.       0.717     2.     ]
 [  41.     101.       0.997    52.     ]]
```

The array consists of rows of edges. The columns represent node number 1, node number 2, their distance, and parent node number. At the top, there are a few pairs that are completely redundant such as make_pop (feature 2) and make_cat_luxury_sports (feature 21). Not a strange outcome considering luxury sports cars, like a Ferrari, are the least popular vehicles in the dataset because they aren't sold as often as, say, a Ford:

```
features_l = ["cylinders", "epa_displ"]
ale_effect = ale(X=X_test, model=cb_mdl, feature=features_l,\
    feature_type='discrete', grid_size=50
)
```

The preceding snippet outputs *Figure 4.18*:
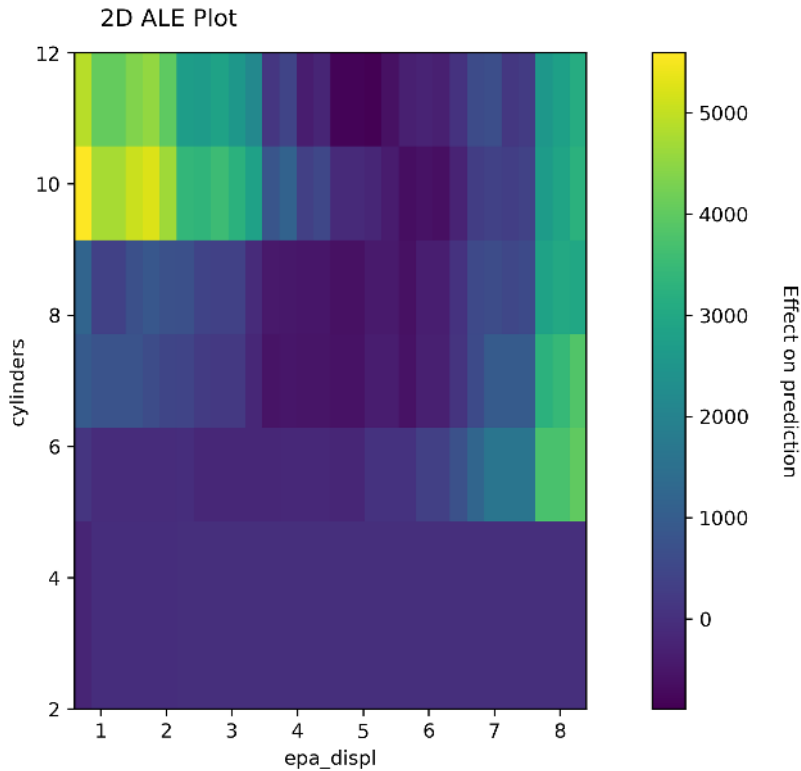
Figure 4.18: 2D ALE plot for epa_displ and cylinders

Figure 4.18 distinctly illustrates higher interaction effects where `cylinders` is larger than 8 and where `epa_displ` is under 4. This finding is counterintuitive considering vehicles with a lot of cylinders are unlikely to have low amounts of engine displacement. On the other hand, the higher interaction effects when there are over 4 cylinders and an engine displacement of at least 6 liters make more sense. Please note that there are other factors such as `year` and `model_type` that correlate with these two features, but ALE is great at separating the effects of `cylinders` and `epa_displ` from other highly correlated features.

PDP interactions plots

I'm sure you're wondering: Given the numerous limitations of PDP, when and where should we contemplate using 2D PDP?

Only when a proven relationship exists between two features, but they are not entirely redundant or independent, and ideally, when they perfectly complement each other. Even then, an ALE plot is advisable as it teases out higher-order effects.

Nonetheless, for illustrative purposes, we will generate a 2D PDP with `long` and `lat`, which are indirectly connected. The code for 2D is very similar to 1D, with the exception that we use `PDPInteract` instead of `PDPIsolate`, and then for the plot function, designate the `plot_type` as `contour`, but `grid` could also be used:

```
features_l = ["long", "lat"]
pdp_interaction_feature = pdp.PDPInteract(
    model=cb_mdl, df=X_test, model_features=X_test.columns,\
    features=features_l, feature_names=features_l, n_classes=0,\
    num_grid_points=15, n_jobs=-1
)
fig, _ = pdp_interaction_feature.plot(
    plot_type='contour', plot_pdp=False
)
fig.show()
```

The preceding snippet outputs *Figure 4.19*:
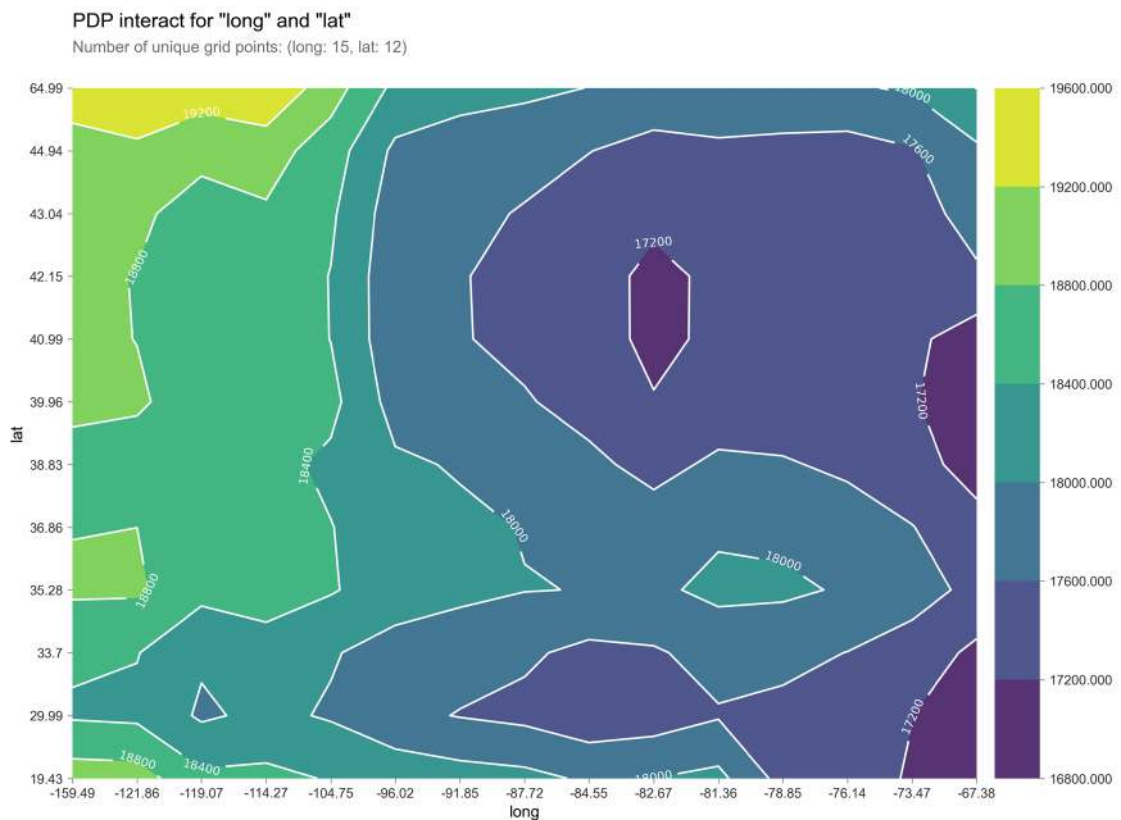
Figure 4.19: PDP interaction contour plot for long and lat

Figure 4.19 serves as proof of how the features are related to the outcome: `long` seems to be responsible for most of the effect, but in some areas, `lat` seems to make a difference, especially in Alaska in the top-left corner. We can check the reliability of this result with a 2D predict plot (`InteractPredictPlot`). As with *Figure 4.12*, the objective is to display the distribution of the data and the predicted scores for that data, but this time, it features a grid of bins that are color-coded for average scores and size-coded for the number of test observations. We can contrast this with the corresponding 2D target plot (`InteractTargetPlot`), which does the same but for the labels (`target`) and not the predicted score:

```
fig, axes, summary_df = info_plots.InteractPredictPlot(
    model=cb_mdl, X=X_test, features=features_l,\
    feature_names=features_l, num_grid_points=(15,12)
)
fig, axes, summary_df = info_plots.InteractTargetPlot(
    df=X_train.join(y_train), target='price', features=features_l,\
    feature_names=features_l, num_grid_points=(15,12)
)
```

The preceding snippet creates both plots in *Figure 4.20*:
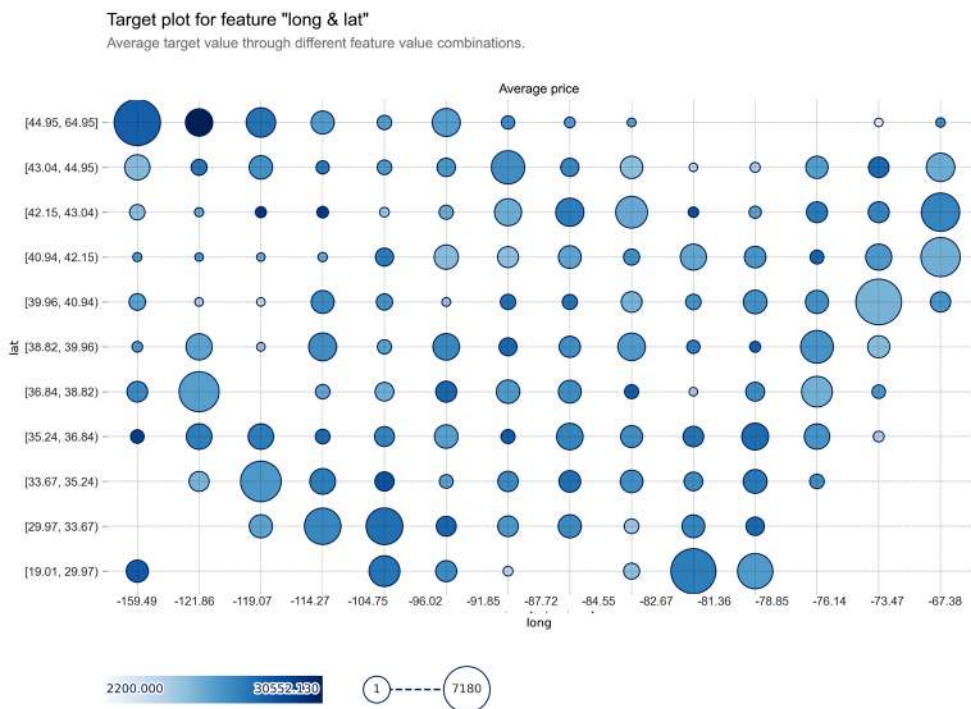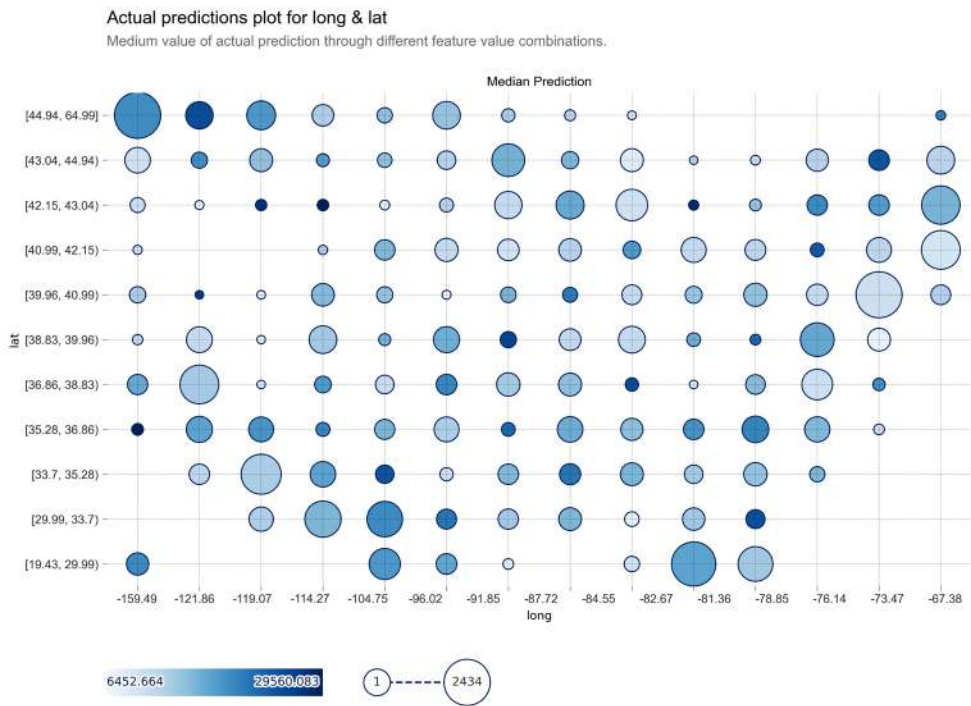
Figure 4.20: PDP actual plot for long and lat

From the initial plot, it's evident that the median predictions are greatest on the western half (on the right side) and especially toward the southwest (on the bottom right), but not by much. The second plot affirms that the labels are indeed more likely to be high-priced in these sections of the plot. Therefore, it's no surprise that the model learned this distribution to its degree of accuracy. These plots aid in affirming a relationship between both features.

In the following chapter, we'll delve deeper into local explanations!

Mission accomplished

We set out to uncover what features helped predict the used car price for the two-sided marketplace. Using the intrinsic parameters of the decision trees, permutation feature importance, and SHAP, we realized that at least 15 features have a negligible impact on either model. Also, about an equal amount of features holds the lion's share of impact. Some of the most critical features, like engine displacement (`epa_displ`) and cylinders, are technical and can vary for the same make and model, so the user would have to know and enter them.

We also found interesting and perfectly valid relationships between different features such as `year` and `odometer`, which help us understand how they interact in the model. We can share all of these findings with the tech startup.

Summary

After reading this chapter, you should understand what model-specific methods to compute feature importance are and their shortcomings. Then, you should have learned how model-agnostic methods' permutation feature importance and SHAP values are calculated and interpreted. You also learned the most common ways to visualize model explanations. You should know your way around global explanation methods like global summaries, feature summaries, and feature interaction plots and their advantages and disadvantages.

In the next chapter, we will delve into local explanations.

Further reading

- Shapley, Lloyd S., 1953, *A value for n-person Games*. In Kuhn, H. W.; Tucker, A. W. (eds.). *Contributions to the Theory of Games. Annals of Mathematical Studies*. 28. Princeton University Press. pp. 307–317: `https://doi.org/10.1515/9781400881970-018`
- Lundberg, S., and Lee, S., 2017, *A Unified Approach to Interpreting Model Predictions*. Advances in Neural Information Processing Systems: `https://arxiv.org/abs/1705.07874` (documentation for SHAP: `https://github.com/slundberg/shap`)
- Lundberg, S.M., Erion, G., and Lee, S., 2018, *Consistent Individualized Feature Attribution for Tree Ensembles*. ICML Workshop: `https://arxiv.org/abs/1802.03888`
- Molnar, C., 2019, *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*: `https://christophm.github.io/interpretable-ml-book/`
- Documentation for SHAP plots: `https://shap.readthedocs.io/en/latest/api.html#plots`

- Original paper for PDP method: Friedman, J.H., 2001, *Greedy function approximation: A gradient boosting machine*. Annals of Statistics, 29, 1189-1232: `https://doi.org/10.1214/aos/1013203451`
- Documentation for PDPBox: `https://pdpbox.readthedocs.io/en/latest/`
- Original paper for ALE method: Apley, D.W., & Zhu, J., 2020, *Visualizing the effects of predictor variables in black box supervised learning models*. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 82. `https://arxiv.org/abs/1612.08468`
- Repository for PyALE: `https://github.com/DanaJomar/PyALE`
- Original paper for LIME method: Ribeiro, M., Singh, S., and Guestrin, C., 2016, *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. `https://arxiv.org/abs/1602.04938`
- Documentation for LIME: `https://lime-ml.readthedocs.io/en/latest/`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

5

Local Model-Agnostic Interpretation Methods

In the previous two chapters, we dealt exclusively with global interpretation methods. This chapter will foray into local interpretation methods, which are there to explain why a single prediction or a group of predictions was made. It will cover how to leverage **SHapley Additive exPlanations** (**SHAP's**) `KernelExplainer` and also another method called **Local Interpretable Model-agnostic Explanations** (**LIME**) for local interpretations. We will also explore how to use these methods with both tabular and text data.

These are the main topics we are going to cover in this chapter:

- Leveraging SHAP's `KernelExplainer` for local interpretations with SHAP values
- Employing LIME
- Using LIME for **Natural Language Processing** (**NLP**)
- Trying SHAP for NLP
- Comparing SHAP with LIME

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `nltk`, `lightgbm`, `rulefit`, `matplotlib`, `seaborn`, `shap`, and `lime` libraries. Instructions on how to install all of these libraries are in the *Preface* of the book.

> The code for this chapter is located here: `https://packt.link/SRqJp`.

The mission

Who doesn't love chocolate?! It's a global favorite, with around nine out of ten people loving it and about a billion people eating it every day. One popular form in which it is consumed is a chocolate bar. However, even universally beloved ingredients can be used in ways that aren't universally appealing—so, chocolate bars can range from the sublime to the mediocre to the downright unpleasant.

Often, this is solely determined by the quality of the cocoa or additional ingredients, and sometimes it becomes an acquired taste once it's combined with exotic flavors.

A French chocolate manufacturer obsessed with excellence has reached out to you. They have a problem. All of their bars have been highly rated by critics, yet critics have very particular taste buds. And some bars they love have inexplicably mediocre sales, but non-critics seem to like them in focus groups and tastings, so they are puzzled why sales don't coincide with their market research. They have found a dataset of chocolate bars rated by knowledgeable lovers of chocolate, and these ratings happen to coincide with their sales. To get an unbiased opinion, they have sought your expertise.

As for the dataset, members of the *Manhattan Chocolate Society* have been meeting since 2007 for the sole purpose of tasting and judging fine chocolate to educate consumers and inspire chocolate makers to produce higher-quality chocolate. Since then, they have compiled a dataset of over 2,200 chocolate bars, rated by their members with the following scale:

- 4.0–5.00 = *Outstanding*
- 3.5–3.99 = *Highly Recommended*
- 3.0–3.49 = *Recommended*
- 2.0–2.99 = *Disappointing*
- 1.0–1.90 = *Unpleasant*

These ratings are derived from a rubric that factors in aroma, appearance, texture, flavor, aftertaste, and overall opinion, and the bars rated are mostly darker chocolate bars since the aim is to appreciate the flavors of cacao. In addition to the ratings, the *Manhattan Chocolate Society* dataset includes many characteristics, such as the country where the cocoa bean was farmed, how many ingredients the bar has, whether it includes salt, and the words used to describe it.

The goal is to understand why one of the chocolate manufacturers' bars is rated *Outstanding* yet sells poorly, while another one, whose sales are impressive, is rated as *Disappointing*.

The approach

You have decided to use local model interpretation to explain why each bar is rated as it is. To that end, you will prepare the dataset and then train classification models to predict if chocolate bar ratings are above or equal to *Highly Recommended*, because the client would like all their bars to fall above this threshold. You will need to train two models: one for tabular data, and another NLP one for the words used to describe the chocolate bars. We will employ **Support Vector Machines** (**SVMs**) and **Light Gradient Boosting Machine** (**LightGBM**), respectively, for these tasks. If you haven't used these black-box models, no worries—we will briefly explain them. Once you have trained the models, then comes the fun part: leveraging two local model-agnostic interpretation methods to understand what makes a specific chocolate bar *Highly Recommended* or not.

These explanation methods are SHAP and LIME, which when combined will provide a richer explanation to convey back to your client. Then, we will compare both methods to understand their strengths and limitations.

The preparations

- You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/05/ChocoRatings.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas`, `numpy`, and `nltk` to manipulate it
- `sklearn` (scikit-learn) and `lightgbm` to split the data and fit the models
- `matplotlib`, `seaborn`, `shap`, and `lime` to visualize the interpretations

You should load all of them first, as follows:

```python
import math
import mldatasets
import pandas as pd
import numpy as np
import re
import nltk
from nltk.probability import FreqDist
from nltk.tokenize import word_tokenize

from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn import metrics, svm
from sklearn.feature_extraction.text import TfidfVectorizer
import lightgbm as lgb
import matplotlib.pyplot as plt
import seaborn as sns
import shap
import lime
import lime.lime_tabular
from lime.lime_text import LimeTextExplainer
```

Understanding and preparing the data

We load the data into a DataFrame we call `chocolateratings_df`, like this:

```python
chocolateratings_df = mldatasets.load("chocolate-bar-ratings_v2")
```

There should be over 2,200 records and 18 columns. We can verify this was the case simply by inspecting the contents of the DataFrame, like this:

```
chocolateratings_df
```

The output shown here in *Figure 5.1* corresponds to what we were expecting:

	company	company_location	review_date	country_of_bean_origin	cocoa_percent	rating	counts_of_ingredients	cocoa_butter	vanilla
0	5150	U.S.A	2019	Madagascar	76.00	3.75	3	1	0
1	5150	U.S.A	2019	Dominican republic	76.00	3.50	3	1	0
2	5150	U.S.A	2019	Tanzania	76.00	3.25	3	1	0
3	A. Morin	France	2012	Peru	63.00	3.75	4	1	0
	:	:	:	:	:	:	:	:	:
2222	Zotter	Austria	2018	Congo	70.00	3.25	3	1	0
2223	Zotter	Austria	2018	Blend	75.00	3.00	3	1	0

Figure 5.1: Contents of the chocolate bar dataset

The data dictionary

The data dictionary comprises the following:

- company: Categorical; the manufacturer of the chocolate bar (out of over 500 different ones)
- company_location: Categorical; the country of the manufacturer (66 different countries)
- review_date: Continuous; the year in which the bar was reviewed (from 2006 to 2020)
- country_of_bean_origin: Categorical; the country where the cocoa beans were harvested (62 different countries)
- cocoa_percent: Categorical; what percentage of the bar is cocoa
- rating: Continuous; the rating given by the *Manhattan Chocolate Society* (possible values: 1–5)
- counts_of_ingredients: Continuous; the amount of ingredients in the bar
- cocoa_butter: Binary; was it made with cocoa butter?
- vanilla: Binary; was it made with vanilla?
- lecithin: Binary; was it made with lecithin?
- salt: Binary; was it made with salt?
- sugar: Binary; was it made with sugar?
- sweetener_without_sugar: Binary; was it made with sweetener without sugar?
- first_taste: Text; word(s) used to describe the first taste
- second_taste: Text; word(s) used to describe the second taste

- third_taste: Text; word(s) used to describe the third taste
- fourth_taste: Text; word(s) used to describe the fourth taste

Now that we have taken a peek at the data, we can quickly prepare it and then work on the modeling and interpretation!

Data preparation

The first thing we ought to do is set aside the text features so that we can process them separately. We can start by creating a DataFrame named `tastes_df` containing text features and then drop them from `chocolateratings_df`. We can then explore `tastes_df` using `head` and `tail`, as illustrated in the following code snippet:

```
tastes_df = chocolateratings_df[
    ['first_taste', 'second_taste', 'third_taste', 'fourth_taste']
]
chocolateratings_df = chocolateratings_df.drop(
    ['first_taste', 'second_taste',
     'third_taste', 'fourth_taste'],axis=1
)
tastes_df.head(90).tail(90)
```

The preceding code produces the DataFrame shown here in *Figure 5.2*:

	first_taste	second_taste	third_taste	fourth_taste
80	oily	vegetal	nutty	cocoa
81	oily	vanilla	melon	cocoa
82	rich	sour	mild smoke	nan
83	fruity	sour	nan	nan
84	roast	high astringent	nan	nan
85	smokey	savory	nan	nan
86	sandy	roasty	nutty	nan
87	roasty	brownie	nutty	nan
88	red wine	rich	long	nan
89	creamy	fruit	cocoa	nan

Figure 5.2: The taste columns have quite a few null values

Now, let's categorically encode the categorical features. There are too many countries in `company_location` and `country_of_bean_origin`, so let's establish a threshold. If, say, there are fewer than 3.333% (or 74 rows) for any country, let's bucket them into an `Other` category and then encode the categories. We can easily do this with the `make_dummies_with_limits` function and the process is shown again in the following code snippet:

```
chocolateratings_df = mldatasets.make_dummies_with_limits(
    chocolateratings_df, 'company_location', 0.03333
)
chocolateratings_df = mldatasets.make_dummies_with_limits(
    chocolateratings_df, 'country_of_bean_origin', 0.03333
)
```

Now, to process the content of `tastes_df`, the following code replaces all the null values with empty strings, then joins all the columns in `tastes_df` together, forming a single series. Then, it strips leading and trailing whitespace. The code is illustrated in the following snippet:

```
tastes_s = tastes_df.replace(
    np.nan, '', regex=True).agg(' '.join, axis=1).str.strip()
```

And voilà! You can verify that the result is a `pandas` series (`tastes_s`) with (mostly) taste-related adjectives by printing it. As expected, this series is the same length as the `chocolateratings_df` DataFrame, as illustrated in the following output:

```
0              cocoa blackberry robust
1                 cocoa vegetal savory
2                    rich fatty bready
3                 fruity melon roasty
4                        vegetal nutty
                      ...
2221         muted roasty accessible
2222     fatty mild nuts mild fruit
2223               fatty earthy cocoa
Length: 2224, dtype: object
```

But let's find out how many of its phrases are unique, with `print(np.unique(tastes_s).shape)`. Since the output is (2178,), that means fewer than 50 phrases are duplicated, so tokenizing by phrases would be a bad idea since so few of them are repeated. After all, when tokenizing, we want the elements to repeat enough times to make it worthwhile.

There are many approaches you could take here, such as tokenizing by bi-grams (sequences of two words) or even subwords (dividing words into logical parts). However, even though order matters slightly (because the first words are related to the first taste, and so on), our dataset is too small and had too many nulls (especially in `third taste` and `fourth taste`) to derive meaning from the order. This is why it was a good choice to concatenate all the "tastes" together, thus removing their discernible division.

Another thing to note is that our words are (mostly) adjectives such as "fruity" and "nutty". We made a small effort to remove adverbs such as "sweetly", but there are still some nouns present, such as "fruit" and "nuts", versus adjectives such as "fruity" and "nutty". We can't be sure if the chocolate connoisseurs who judged the bars meant something different by using "fruit" rather than "fruity". However, if we were sure of this, we could have performed **stemming** or **lemmatization** to turn all instances of "fruit", "fruity", and "fruitiness" into a consistent "fru" (*stem*) or "fruiti" (*lemma*). We won't concern ourselves with this because many of our adjectives' variations are not as common in the phrases anyway.

Let's find out the most common words by first tokenizing them with `word_tokenize` and using `FreqDist` to count their frequency. We can then place the resulting `tastewords_fdist` dictionary into a DataFrame (`tastewords_df`). We can save only those words with more than 74 instances as a list (`commontastes_l`). The code is illustrated in the following snippet:

```
tastewords_fdist = FreqDist(
    word for word in word_tokenize(tastes_s.str.cat(sep=' '))
)
tastewords_df = pd.DataFrame.from_dict(
    tastewords_fdist, orient='index').rename(columns={0:'freq'}
)
commontastes_l = tastewords_df[
    tastewords_df.freq > 74].index.to_list()
print(commontastes_l)
```

As you can tell from the following output for `commontastes_l`, the most common words are mostly different (except for `spice` and `spicy`):

```
['cocoa', 'rich', 'fatty', 'roasty', 'nutty', 'sweet', 'sandy', 'sour',
'intense', 'mild', 'fruit', 'sticky', 'earthy', 'spice', 'molasses', 'floral',
'spicy', 'woody', 'coffee', 'berry', 'vanilla', 'creamy']
```

Something we can do with this list to enhance our tabular dataset is turn these common words into binary features. In other words, there will be a column for each one of these "common tastes" (`commontastes_l`), and if the "tastes" for the chocolate bar include it, the column will contain a 1, otherwise a 0. Fortunately, we can easily do this with two lines of code. First, we create a new column with our text-tastes series (`tastes_s`). Then, we use the `make_dummies_from_dict` function we used in the last chapter to generate the dummy features by looking for each "common taste" in the contents of our new column, as follows:

```
chocolateratings_df['tastes'] = tastes_s
chocolateratings_df = mldatasets.make_dummies_from_dict(
    chocolateratings_df, 'tastes', commontastes_l)
```

Now that we are done with our feature engineering, we can use `info()` to examine our DataFrame. The output has all numeric non-null features except for `company`. There are over 500 companies, so **categorical encoding** of this feature would be complicated and, because it would be advisable to bucket most companies as `Other`, it would likely introduce bias toward the few companies that are most represented. Therefore, it's better to remove this column altogether. The output is shown here:

```
RangeIndex: 2224 entries, 0 to 2223
Data columns (total 46 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   company                 2224 non-null   object
 1   review_date             2224 non-null   int64
 2   cocoa_percent           2224 non-null   float64
 :      :                        :    :        :
 43  tastes_berry            2224 non-null   int64
 44  tastes_vanilla          2224 non-null   int64
 45  tastes_creamy           2224 non-null   int64
dtypes: float64(2), int64(30), object(1), uint8(13)
```

Our last step to prepare the data for modeling starts with initializing `rand`, a constant to serve as our "random state" throughout this exercise. Then, we define `y` as the `rating` column converted to 1 if greater than or equal to `3.5`, and `0` otherwise. `X` is everything else (excluding `company`). Then, we split `X` and `y` into train and test datasets with `train_test_split`, as illustrated in the following code snippet:

```
rand = 9
y = chocolateratings_df['rating'].\
apply(lambda x: 1 if x >= 3.5 else 0)
X = chocolateratings_df.drop(['rating','company'], axis=1).copy()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=rand)
```

In addition to the tabular test and train datasets, for our NLP models, we will need text-only feature datasets that are consistent with our `train_test_split` so that we can use the same y labels. To this end, we can do this by subsetting our tastes series (`tastes_s`) using the `index` of our `X_train` and `X_test` sets to yield NLP-specific versions of the series, as follows:

```
X_train_nlp = tastes_s[X_train.index]
X_test_nlp = tastes_s[X_test.index]
```

OK! We are all set now. Let's start modeling and interpreting our models!

Leveraging SHAP's KernelExplainer for local interpretations with SHAP values

For this section, and for subsequent use, we will train a **Support Vector Classifier** (**SVC**) model first.

Training a C-SVC model

SVM is a family of model classes that operate in high-dimensional space to find an optimal hyperplane, where they attempt to separate the classes with the maximum margin between them. Support vectors are the points closest to the decision boundary (the dividing hyperplane) that would change it if were removed. To find the best hyperplane, they use a cost function called **hinge loss** and a computationally cheap method to operate in high-dimensional space, called the **kernel trick**, and even though a hyperplane suggests linear separability, it's not always limited to a linear kernel.

The scikit-learn implementation we will use is called C-SVC. SVC uses an L2 regularization parameter called *C* and, by default, uses a kernel called the **Radial Basis Function** (**RBF**), which is decidedly non-linear. For an RBF, a **gamma** hyperparameter defines the radius of influence of each training example in the kernel, but in an inversely proportional fashion. Hence, a low value increases the radius, while a high value decreases it.

The SVM family includes several variations for classification and even regression classes through **Support Vector Regression** (**SVR**). The most significant advantage of SVM models is that they tend to work effectively and efficiently when there are many features compared to the observations, and even when the features exceed the observations! It also tends to find latent nonlinear relationships in the data, without overfitting or becoming unstable. However, SVMs are not as scalable to larger datasets, and it's hard to tune their hyperparameters.

Since we will use `seaborn` plot styling, which is activated with `set()`, for some of this chapter's plots, we will first save the original `matplotlib` settings (`rcParams`) so that we can restore them later. One thing to note about `SVC` is that it doesn't natively produce probabilities since it's linear algebra. However, if `probability=True`, the scikit-learn implementation uses cross-validation and then fits a logistic regression model to the SVC's scores to produce the probabilities. We are also using `gamma=auto`, which means it is set to 1/# features—so, 1/44. As always, it is recommended to set your `random_state` parameter for reproducibility. Once we fit the model to the training data, we can use `evaluate_class_mdl` to evaluate our model's predictive performance, as illustrated in the following code snippet:

```
svm_mdl = svm.SVC(probability=True, gamma='auto', random_state=rand)
fitted_svm_mdl = svm_mdl.fit(X_train, y_train)
y_train_svc_pred, y_test_svc_prob, y_test_svc_pred =\
    mldatasets.evaluate_class_mdl(
    fitted_svm_mdl, X_train, X_test, y_train, y_test
)
```

The preceding code produces the output shown here in *Figure 5.3*:
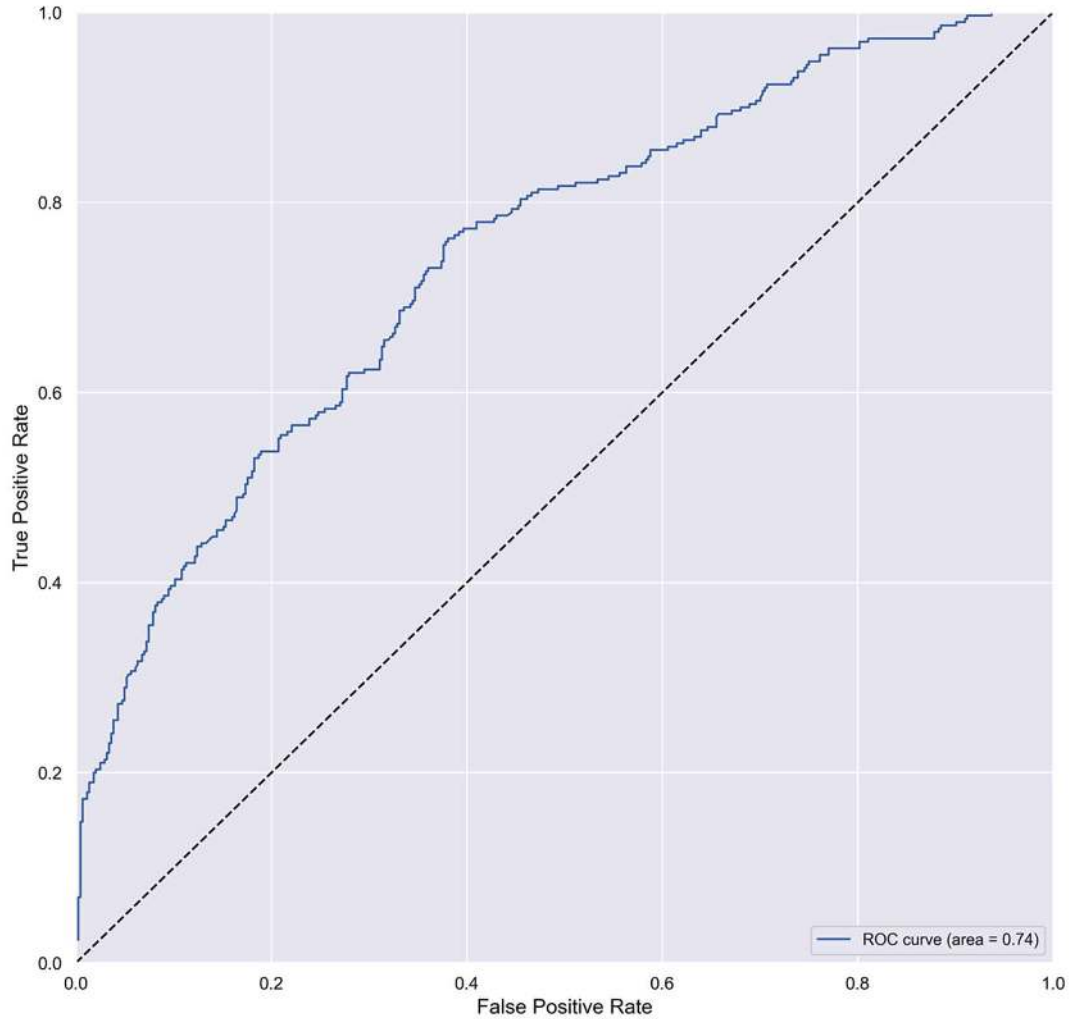

```
Accuracy_train:   0.7315          Accuracy_test:   0.6989
Precision_test:   0.6845          Recall_test:     0.4414
ROC-AUC_test:     0.7450          F1_test:         0.5367          MCC_test: 0.3461
```

Figure 5.3: Predictive performance of our SVC model

Figure 5.3 shows the performance achieved is not bad, considering this is a small imbalanced dataset in an already challenging domain for machine learning models' user ratings. In any case, the **Area Under the Curve** (**AUC**) curve is above the dotted coin toss line, and the **Matthews correlation coefficient** (**MCC**) is safely above 0. More importantly, precision is substantially higher than recall, and this is very good given the hypothetical cost of misclassifying a lousy chocolate bar as *Highly Recommended*. We favor precision over recall because we would prefer to have fewer false positives than false negatives.

Computing SHAP values using KernelExplainer

Given how computationally intensive calculating SHAP values by brute force can be, the SHAP library takes many statistically valid shortcuts. As we learned in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, these shortcuts range from leveraging a decision tree's structure (`TreeExplainer`) to the difference in a neural network's activations, a baseline (`DeepExplainer`) to a neural network's gradient (`GradientExplainer`). These shortcuts make the explainers model-specific since they are limited to a family of model classes. However, there is a model-agnostic explainer in SHAP, called the `KernelExplainer`.

`KernelExplainer` has two shortcuts; it samples a subset of all feature permutations for coalitions and uses a weighting scheme according to the size of the coalition to compute SHAP values. The first shortcut is a recommended technique to reduce computation time. The second one is drawn from LIME's weighting scheme, which we will cover next in this chapter, and the authors of SHAP did this so that it remains compliant with Shapley. However, for "missing" features in the coalition, it randomly samples from the features' values in a background training dataset, which violates the **dummy** property of Shapley values. More importantly, as with **permutation feature importance**, if there's multicollinearity, it puts too much weight on unlikely instances. Despite this near-fatal flaw, `KernelExplainer` has all the other benefits of Shapley values and at least one of LIME's main advantages.

Before we engage with the `KernelExplainer`, it's important to note that for classification models, it yields a list of multiple SHAP values. We can access the list of values for each class with an index. Confusion may arise if this index is not in the order we expect because it's in the order provided by the model. So, it is essential to make sure of the order of the classes in our model by running `print(svm_mdl.classes_)`.

The output `array([0, 1])` tells us that *Not Highly Recommended* has an index of 0, as we would expect, and *Highly Recommended* has an index of 1. We are interested in the SHAP values for the latter because this is what we are trying to predict.

`KernelExplainer` takes a `predict` function for a model (`fitted_svm_mdl.predict_proba`) and some background training data (`X_train_summary`). `KernelExplainer` leverages additional measures to minimize computation. One of these is using **k-means** to summarize the background training data instead of using it whole. Another method could be using a sample of the training data. In this case, we opted for k-means clustering into 10 centroids. Once we have initialized our explainer, we can use samples of our test dataset (`nsamples=200`) to come up with the SHAP values. It uses L1 regularization (`l1_reg`) during the fitting process. What we are telling it here is to regularize to a point where it only has 20 relevant features. Lastly, we can use a `summary_plot` to plot our SHAP values for class 1. The code is illustrated in the following snippet:

```
np.random.seed(rand)
X_train_summary = shap.kmeans(X_train, 10)
shap_svm_explainer = shap.KernelExplainer(
    fitted_svm_mdl.predict_proba, X_train_summary
)
```

```
shap_svm_values_test = shap_svm_explainer.shap_values(
    X_test, nsamples=200, l1_reg="num_features(20)"
)
shap.summary_plot(shap_svm_values_test[1], X_test, plot_type="dot")
```

The preceding code produces the output shown in *Figure 5.4*. Even though the point of this chapter is local model interpretation, it's important to start with the global form of this to make sure outcomes are intuitive. If they aren't, perhaps something is amiss:
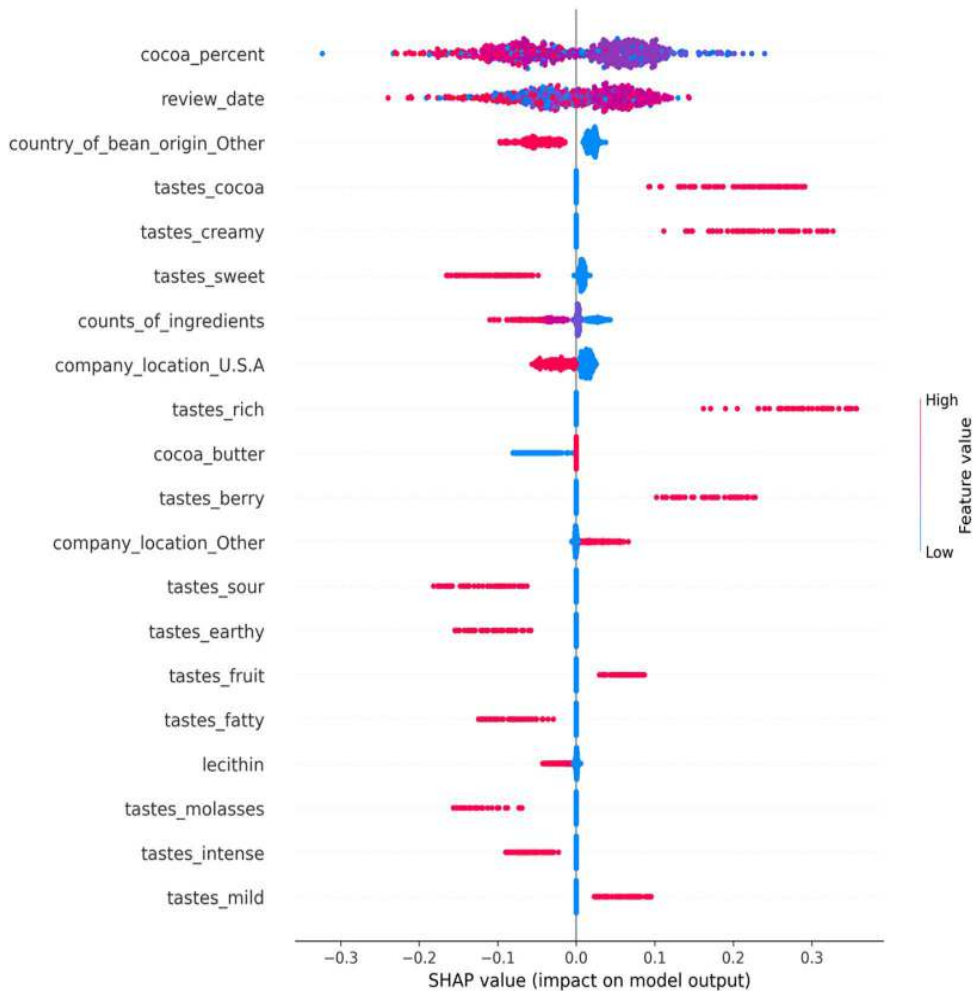
Figure 5.4: Global model interpretation with SHAP using a summary plot

In *Figure 5.4*, we can tell that the highest (red) cocoa percentages (`cocoa_percent`) tend to correlate with a decrease in the likelihood of *Highly Recommended*, but the middle values (purple) tend to increase it. This finding makes intuitive sense because the darkest chocolates are more of an acquired taste than less-dark chocolates. The low values (blue) are scattered throughout so they show no trend, but this could be because there aren't many.

On the other hand, `review_date` suggests that it was likely to be *Highly Recommended* in earlier years. There are significant shades of red and purple on both sides of 0, so it's hard to identify a trend here. A dependence plot, such as those used in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, would be better for this purpose. However, it's very easy for binary features to visualize how high and low values, ones and zeros, impact the model. For instance, we can tell that the presence of cocoa, creamy, rich, and berry tastes increases the likelihood of the chocolate being recommended, while sweet, earthy, sour, and fatty tastes do the opposite. Likewise, the odds for *Highly Recommended* decrease if the chocolate was manufactured in the US! Sorry, the US.

Local interpretation for a group of predictions using decision plots

For local interpretation, you don't have to visualize one point at a time—you can instead interpret several at a time. The key is providing some context to compare the points adequately, and there can't be so many that you can't distinguish them. Usually, you would find outliers or only those that meet specific criteria. For this exercise, we will select only those bars that were produced by your client, as follows:

```
sample_test_idx = X_test.index.get_indexer_for(
    [5,6,7,18,19,21,24,25,27]
)
```

One great thing about Shapley is its additivity property, which can be easily demonstrated. If you add all the SHAP values to the expected value used to compute them, you get a prediction. Of course, this is a classification problem, so the prediction is a probability; so, to get a Boolean array instead, we have to check if the probability is greater than 0.5. We can check if this Boolean array matches our model's test dataset predictions (`y_test_svc_pred`) by running the following code:

```
expected_value = shap_svm_explainer.expected_value[1]
y_test_shap_pred =\
    (shap_svm_values_test[1].sum(1) + expected_value) > 0.5
print(np.array_equal(y_test_shap_pred, y_test_svc_pred))
```

It should, and it does! You can see it confirmed with a `True` value.

SHAP's decision plot comes with a highlight feature that we can use to make false negatives (`FN`) stand out. Now, let's figure out which of our sample observations are `FN`, as follows:

```
FN = (~y_test_shap_pred[sample_test_idx]) &
    (y_test.iloc[sample_test_idx] == 1).to_numpy()
```

We can now quickly reset and plot a `decision_plot`. It takes the `expected_value`, the SHAP values, and the actual values of those items we wish to plot. Optionally, we can provide a Boolean array of the items we want to highlight, with dotted lines—in this case, the false negatives (`FN`), as illustrated in the following code snippet:

```
shap.decision_plot(
    expected_value, shap_svm_values_test[1][sample_test_idx],\
    X_test.iloc[sample_test_idx], highlight=FN)
```

The plot produced in *Figure 5.5* has a single color-coded line for each observation.
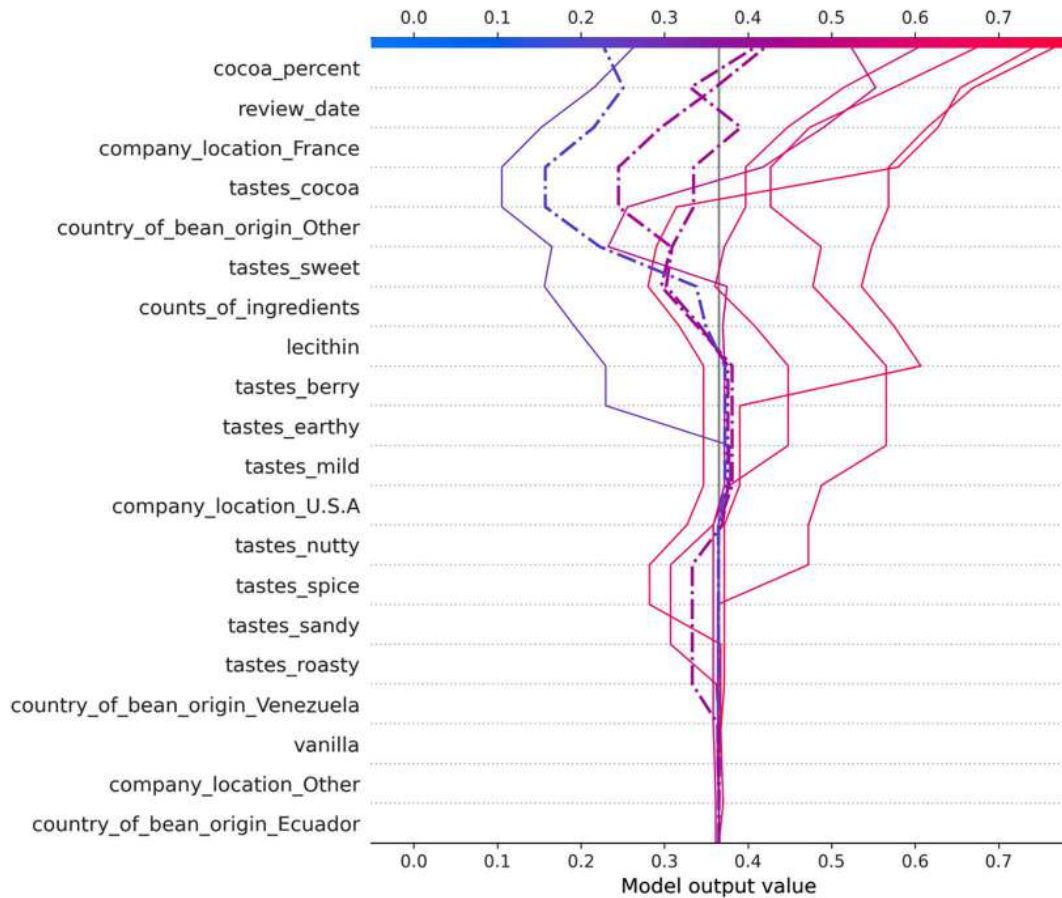
Figure 5.5: Local model interpretation with SHAP for a sample of predictions, highlighting false negatives

The color of each line represents not the value of any feature, but the model output. Since we used `predict_proba` in `KernelExplainer`, this is a probability, but otherwise, it would have displayed SHAP values, and the value they have when they strike the top x-axis is the predicted value. The features are sorted in terms of importance but only among the observations plotted, and you can tell that the lines increase and decrease horizontally depending on each feature. How much they vary and in which direction depends on the feature's contribution to the outcome. The gray line represents the class's expected value, which is like the intercept in a linear model. In fact, similarly, all lines start at this value, making it best to read the plot from bottom to top.

You can tell that there are three false negatives plotted in *Figure 5.5* because they have dotted lines. Using this plot, we can easily visualize which features made them veer toward the left the most because this is what made them negative predictions.

For instance, we know that the leftmost false negative was to the right of the expected value line until `lecithin` and then continued decreasing until `company_location_France`, and `review_date` increased its likelihood of *Highly Recommended*, but it wasn't enough. You can tell that `county_of_bean_origin_Other` decreased the likelihood of two of the misclassifications. This decision could be unfair because the country could be one of over 50 countries that didn't get their own feature. Quite possibly, there's a lot of variation between the beans of these countries grouped together.

Decision plots can also isolate a single observation. When it does this, it prints the value of each feature next to the dotted line. Let's plot one for a decision plot of the same company (true-positive observation #696), as follows:

```
shap.decision_plot(
    expected_value, shap_svm_values_test[1][696],
    X_test.iloc[696],highlight=0
)
```

Figure 5.6 here was outputted by the preceding code:
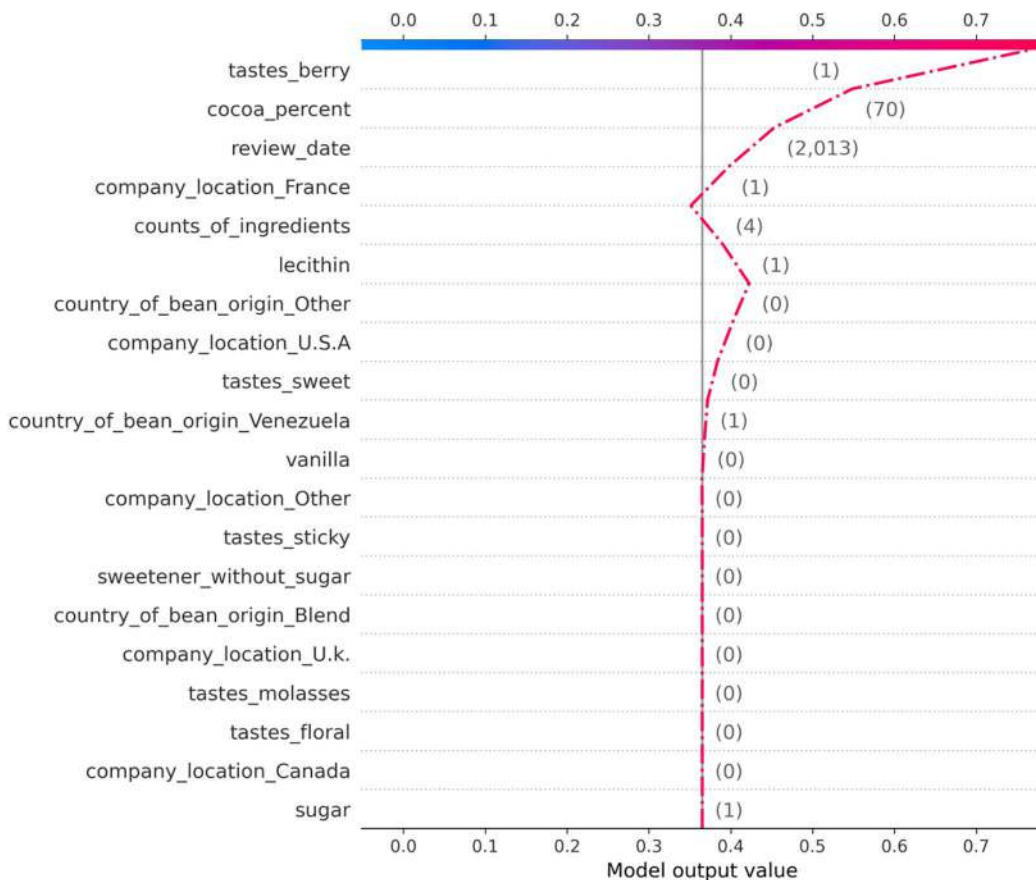
Figure 5.6: The SHAP decision plot for a single true positive in the sample of predictions

In *Figure 5.6*, you can see that `lecithin` and `counts_of_ingredients` decreased the *Highly Recommended* likelihood to a point where they could have jeopardized it. Fortunately, all features above those veered the line decidedly rightward because `company_location_France=1`, `cocoa_percent=70`, and `tastes_berry=1` are all favorable.

Local interpretation for a single prediction at a time using a force plot

Your client, the chocolate manufacturer, has two bars they want you to compare. Bar #5 is *Outstanding* and #24 is *Disappointing*. They are both in your test dataset. One way of comparing them is to place their values side by side in a DataFrame to understand how exactly they differ. We will concatenate the rating, the actual label y, and the y_pred predicted label to these observations' values, as follows:

```
eval_idxs = (X_test.index==5) | (X_test.index==24)
X_test_eval = X_test[eval_idxs]
eval_compare_df = pd.concat([
    chocolateratings_df.iloc[X_test[eval_idxs].index].rating,
    pd.DataFrame({'y':y_test[eval_idxs]}, index=[5,24]),
    pd.DataFrame({'y_pred':y_test_svc_pred[eval_idxs]},
    index=[24,5]), X_test_eval], axis=1).transpose()
eval_compare_df
```

The preceding code produces the DataFrame shown in *Figure 5.7*:

	5	24
rating	4	3
y	1	0
y_pred	1	0
review_date	**2013**	**2015**
cocoa_percent	70	70
counts_of_ingredients	4	4
cocoa_butter	1	1
vanilla	0	0
lecithin	1	1
salt	0	0
sugar	1	1
sweetener_without_sugar	0	0
company_location_Canada	0	0
:	:	:
country_of_bean_origin_Nicaragua	0	0
country_of_bean_origin_Other	**0**	**1**
country_of_bean_origin_Peru	0	0
country_of_bean_origin_Venezuela	**1**	**0**
tastes_cocoa	0	0

Figure 5.7: Observations #5 and #24 side by side, with feature differences highlighted in yellow

With this DataFrame, you can confirm that they aren't misclassifications because y=y_pred. A misclassification could make model interpretations unreliable for understanding why people tend to like one chocolate bar more than another. Then, you can examine the features to spot the differences—for instance, you can tell that the review_date is 2 years apart. Also, the beans for the *Outstanding* bar were from Venezuela, and the *Disappointing* beans came from another, lesser-represented country. The *Outstanding* one had a berry taste, and the *Disappointing* one was earthy.

The force plot can tell us a complete story of what weighed in the model's decisions (and, presumably, the reviewers'), and gives us clues as to what consumers might prefer. Plotting a `force_plot` requires the expected value for the class of your interest (`expected_value`), the SHAP values for the observation of your interest, and this observation's actual values. We will start with observation #5, as illustrated in the following code snippet:

```
shap.force_plot(
    expected_value,
    shap_svm_values_test[1][X_test.index==5],\
    X_test[X_test.index==5],
    matplotlib=True
)
```

The preceding code produces the plot shown in *Figure 5.8*. This force plot depicts how much `review_date`, `cocoa_percent`, and `tastes_berry` weigh in the prediction, while the only feature that seems to be weighing in the opposite direction is `counts_of_ingredients`:
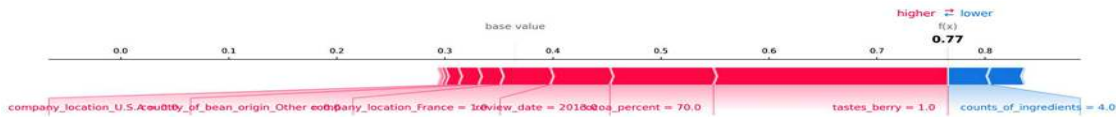
Figure 5.8: The force plot for observation #5 (Outstanding)

Let's compare it with a force plot of observation #24, as follows:

```
shap.force_plot(expected_value,\
                shap_svm_values_test[1][X_test.index==24],\
                X_test[X_test.index==24], matplotlib=True)
```

The preceding code produces the plot shown in *Figure 5.9*. We can easily tell that `tastes_earthy` and `country_of_bean_origin_Other` are considered highly negative attributes by our model. The outcome can be mostly explained by the difference in the chocolate tasting of "berry" versus "earthy." Despite our findings, the beans' origin country needs further investigation. After all, it is possible that the actual country of origin doesn't correlate with poor ratings.
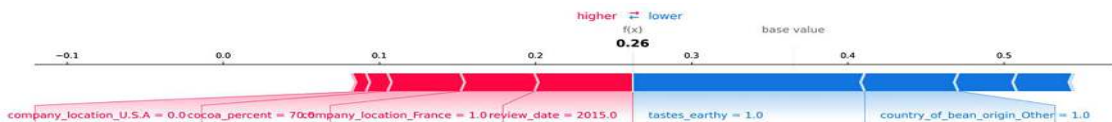
Figure 5.9: The force plot for observation #24 (Disappointing)

In this section, we covered the `KernelExplainer`, which uses some tricks it learned from LIME. But what is LIME? We will find that out next!

Employing LIME

Until now, the model-agnostic interpretation methods we've covered attempt to reconcile the totality of outputs of a model with its inputs. For these methods to get a good idea of how and why X becomes y_pred, we need some data first. Then, we perform simulations with this data, pushing variations of it into a model and evaluating what comes out of the model. Sometimes, they even leverage a global surrogate to connect the dots. By using what we learned in this process, we yield feature importance values that quantify a feature's impact, interactions, or decisions on a global level. For many methods such as SHAP, these can be observed locally too. However, even when they can be observed locally, what was quantified globally may not apply locally. For this reason, there should be another approach that quantifies the local effects of features solely for local interpretation—one such as LIME!

What is LIME?

**LIME** trains local surrogates to explain a single prediction. To this end, it starts by asking us which *data point* we want to interpret. You also provide it with your black-box model and a sample dataset. It then makes predictions on a *perturbed* version of the dataset with the model, creating a scheme whereby it samples and *weighs* points higher if they are *closer* to your chosen data point. This area around your point is called a neighborhood. Then, using the sampled points and black-box predictions in this neighborhood, it trains a weighted *intrinsically interpretable surrogate model*. Lastly, it interprets the surrogate model.

There are lots of keywords to unpack here so let's define them, as follows:

- **Chosen data point:** LIME calls the data point, row, or observation you want to interpret an *instance*. It's just another word for this concept.
- **Perturbation:** LIME simulates new samples by adding noise to each sample. In other words, it creates random samples close to each instance.
- **Weighting scheme:** LIME uses an exponential smoothing kernel to both define the local instance neighborhood radius and determine how to weigh the points farthest versus those closest to the instance.
- **Closer:** LIME uses Euclidean distance for tabular and image data, and cosine similarity for text. This is hard to imagine in high-dimensional feature spaces, but you can calculate the distance between points for any number of dimensions and find which points are closest to the one of interest.
- **Intrinsically interpretable surrogate model:** LIME uses a sparse linear model with weighted ridge regularization. However, it could use any intrinsically interpretable model as long as the data points can be weighted. The idea behind this is twofold. It needs a model that can yield reliable intrinsically interpretable parameters, such as coefficients that indicate the influence of the feature on the prediction. It also needs to consider data points closest to the chosen point more because these are more relevant.

Much like with **k-Nearest Neighbors** (**k-NN**), the intuition behind LIME is that points in a neighborhood have commonality because we expect points close to each other to have similar, if not the same, labels. There are decision boundaries for classifiers, so this could be a very naive assumption to make when close points are divided by one.

Similar to another model class in the Nearest Neighbors family, **Radius Nearest Neighbors**, LIME factors in the distance along a radius and weighs points accordingly, although it does this exponentially. However, LIME is not a model class but an interpretation method, so the similarities stop there. Instead of "voting" for predictions among neighbors, it fits a weighted surrogate sparse linear model because it assumes that every complex model is linear locally, and because it's not a model class, the predictions the surrogate model makes don't matter. In fact, the surrogate model doesn't even have to fit the data like a glove because all you need from it is the coefficients. Of course, that being said, it is best if it fits well so that there is higher fidelity in the interpretation.

LIME works for tabular, image, and text data and generally has high local fidelity, meaning that it can approximate the model predictions quite well on a local level. However, this is contingent on the neighborhood being defined correctly, which stems from choosing the right kernel width and the assumption of local linearity holding true.

Local interpretation for a single prediction at a time using LimeTabularExplainer

To explain a single prediction, you first instantiate a `LimeTabularExplainer` by providing it with your sample dataset in a `numpy` 2D array (`X_test.values`), a list with the names of the features (`X_test.columns`), a list with the indices of the categorical features (only the first three features aren't categorical), and the class names. Even though only the sample dataset is required, it is recommended that you provide names for your features and classes so that the interpretation makes sense. For tabular data, telling LIME which features are categorical (`categorical_features`) is important because it treats categorical features differently from continuous ones, and not specifying this could potentially make for a poor-fitting local surrogate. Another parameter that can greatly impact the local surrogate is `kernel_width`. This defines the diameter of the neighborhood, thus answering the question of what is considered local. It has a default value, which may or may not yield interpretations that make sense for your instance. You could tune this parameter on an instance-by-instance basis to ensure that each time you generate an explanation, it is consistent. Please note that the random nature of perturbation, when applied to a large neighborhood, might lead to inconsistent results. So as you make this neighborhood smaller, you will reduce the variability between runs. The code can be seen in the following snippet:

```python
lime_svm_explainer = lime.lime_tabular.LimeTabularExplainer(
    X_test.values,
    feature_names=X_test.columns,
    categorical_features=list(range(3,44)),
    class_names=['Not Highly Recomm.', 'Highly Recomm.']
)
```

With the instantiated explainer, you can now use `explain_instance` to fit a local surrogate model to observation #5. We will also use our model's classifier function (`predict_proba`) and limit our number of features to eight (`num_features=8`). We can take the "explanation" returned and immediately visualize it with `show_in_notebook`. At the same time, the `predict_proba` parameter makes sure it also includes a plot to show which class is the most probable, according to the local surrogate model. The code is illustrated in the following snippet:

```
lime_svm_explainer.explain_instance(
    X_test[X_test.index==5].values[0],
    fitted_svm_mdl.predict_proba,
    num_features=8
).show_in_notebook(predict_proba=True)
```

The preceding code provides the output shown in *Figure 5.10*. This figure can be read as other feature importance plots with the most influential features having the highest coefficients – and vice versa. However, it has features that weigh in each direction. According to the local surrogate, a `cocoa_percent` value smaller or equal to 70 is a favorable attribute, as is the berry taste. A lack of sour, sweet, and molasses tastes also weighs favorably in this model. However, a lack of rich, creamy, and cocoa tastes does the opposite, but not enough to push the scales toward *Not Highly Recommended*:
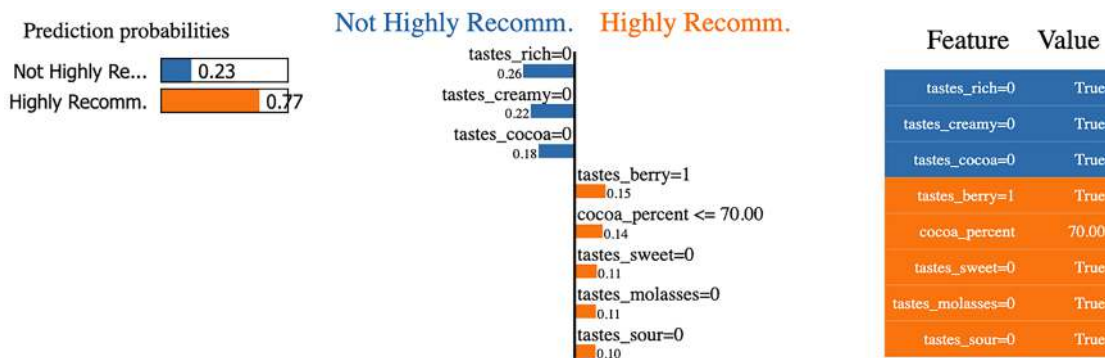

Figure 5.10: LIME's tabular explanation for observation #5 (Outstanding)

With a small adjustment to the code that produced *Figure 5.10*, we can produce the same plot but for observation #24, as follows:

```
lime_svm_explainer.explain_instance(
    X_test[X_test.index==24].values[0],
    fitted_svm_mdl.predict_proba,
    num_features=8
).show_in_notebook(predict_proba=True)
```

Here, in *Figure 5.11*, we can clearly see why the local surrogate believes that observation #24 is *Not Highly Recommended*:
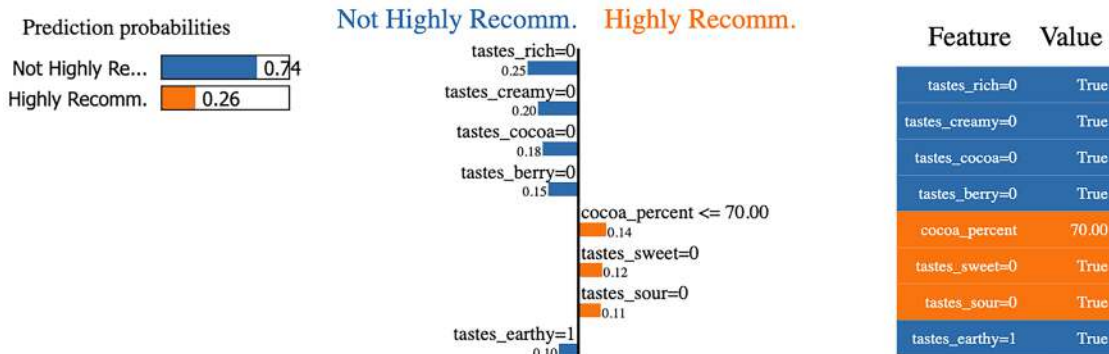
Figure 5.11: LIME's tabular explanation for observation #24 (Disappointing)

Once you compare the explanation of #24 (*Figure 5.11*) with that of #5 (*Figure 5.10*), the problems become evident. A single feature, `tastes_berry`, is what differentiates both explanations. Of course, we have limited it to the top eight features, so there's probably much more to it. However, you would expect the top eight features to include the ones that make the most difference.

According to SHAP, knowing that `tastes_earthy=1` is what globally explains the disappointing nature of the #24 chocolate bar, but this appears to be counterintuitive. So, what happened? It turns out that observations #5 and #24 are relatively similar and, thus, in the same neighborhood. This neighborhood also includes many chocolate bars with berry tastes and very few with earthy ones. However, there are not enough earthy ones to consider it a salient feature, so it attributes the difference between *Highly Recommended* and *Not Highly Recommended* to other features that seem to differentiate more often, at least locally. The reason for this is twofold: the local neighborhood could be too small, and linear models, given their simplicity, are on the bias end of a *bias-variance trade-off*. This bias is only exacerbated by the fact that some features such as `tastes_berry` can appear relatively more often than `tastes_earthy`. There's an approach we can use to fix this, and we'll cover this in the next section.

Using LIME for NLP

At the beginning of the chapter, we set aside training and test datasets with the cleaned-up contents of all the "tastes" columns for NLP. We can take a peek at the test dataset for NLP, as follows:

```
print(X_test_nlp)
```

This outputs the following:

```
1194                  roasty nutty rich
77      roasty oddly sweet marshmallow
121             balanced cherry choco
```

```
411                     sweet floral yogurt
1259             creamy burnt nuts woody
                        ...
327            sweet mild molasses bland
1832            intense fruity mild sour
464                 roasty sour milk note
2013             nutty fruit sour floral
1190             rich roasty nutty smoke
Length: 734, dtype: object
```

No machine learning model can ingest the data as text, so we need to turn it into a numerical format—in other words, vectorize it. There are many techniques we can use to do this. In our case, we are not interested in the position of words in each phrase, nor the semantics. However, we are interested in their relative occurrence—after all, that was an issue for us in the last section.

For these reasons, **Term Frequency-Inverse Document Frequency** (**TF-IDF**) is the ideal method because it's meant to evaluate how often a term (each word) appears in a document (each phrase). However, it's weighted according to its frequency in the entire corpus (all phrases). We can easily vectorize our datasets using the TF-IDF method with `TfidfVectorizer` from scikit-learn. However, when you have to make TF-IDF scores, these are fitted to the training dataset only because that way, the transformed train and test datasets have consistent scoring for each term. Have a look at the following code snippet:

```
vectorizer = TfidfVectorizer(lowercase=False)
X_train_nlp_fit = vectorizer.fit_transform(X_train_nlp)
X_test_nlp_fit = vectorizer.transform(X_test_nlp)
```

To get an idea of what the TF-IDF score looks like, we can place all the feature names in one column of a DataFrame, and their respective scores for a single observation in another. Note that since the vectorizer produces a `scipy` sparse matrix, we have to convert it into a `numpy` matrix with `todense()` and then a `numpy` array with `asarray()`. We can sort this DataFrame in descending order by TD-IDF scores. The code is shown in the following snippet:

```
pd.DataFrame(
    {
        'taste':vectorizer.get_feature_names_out(),
        'tf-idf': np.asarray(
            X_test_nlp_fit[X_test_nlp.index==5].todense())[0]
    }
).sort_values(by='tf-idf', ascending=False)
```

The preceding code produces the output shown here in *Figure 5.12*:

	taste	tf-idf
305	raspberry	0.585538
259	nut	0.491542
265	oily	0.463973
64	caramel	0.447504
274	papaya	0.000000

Figure 5.12: The TF-IDF scores for words present in observation #5

And as you can tell from *Figure 5.12*, the TD-IDF scores are normalized values between 0 and 1, and those most common in the corpus have a lower value. Interestingly enough, we realize that observation #5 in our tabular dataset had `berry=1` because of **raspberry**. The categorical encoding method we used searched occurrences of `berry` regardless of whether they matched an entire word or not. This isn't a problem because raspberry is a kind of berry, and raspberry wasn't one of our common tastes with its own binary column.

Now that we have vectorized our NLP datasets, we can proceed with the modeling.

Training a LightGBM model

LightGBM, like **XGBoost**, is another very popular and performant gradient-boosting framework that leverages boosted-tree ensembles and histogram-based split finding. The main differences lie in the split method's algorithms, which for LightGBM uses sampling with **Gradient-Based One-Side Sampling** (**GOSS**) and bundling sparse features with **Exclusive Feature Bundling** (**EFB**) versus XGBoost's more rigorous **Weighted Quantile Sketch** and **Sparsity-aware Split Finding**. Another difference lies in how the trees are built, which is **depth-first** (top-down) for XGBoost and **breadth-first** (across a tree's leaves) for LightGBM. We won't get into the details of how these algorithms work because that would derail the topic at hand. However, it's important to note that thanks to GOSS, LightGBM is usually even faster than XGBoost, and though it can lose predictive performance due to GOSS split approximations, it gains some of it back with its best-first approach. On the other hand, **Explainable Boosting Machine** (**EBM**) makes LightGBM ideal for training on sparse features efficiently and effectively, such as those in our `X_train_nlp_fit` sparse matrix! That pretty much sums up why we are using LightGBM for this exercise.

To train the LightGBM model, we first initialize the model by setting the maximum tree depth (`max_depth`), the learning rate (`learning_rate`), the number of boosted trees to fit (`n_estimators`), the `objective`, which is binary classification, and—last but not least—the `random_state` for reproducibility. With `fit`, we train the model using our vectorized NLP training dataset (`X_train_nlp_fit`) and the same labels used for the SVM model (`y_train`). Once trained, we can evaluate using the `evaluate_class_mdl` we used with the SVM. The code is illustrated in the following snippet:

```
lgb_mdl = lgb.LGBMClassifier(
    max_depth=13,
```

```
    learning_rate=0.05,
    n_estimators=100,
    objective='binary',
    random_state=rand
)
fitted_lgb_mdl = lgb_mdl.fit(X_train_nlp_fit, y_train)
y_train_lgb_pred, y_test_lgb_prob, y_test_lgb_pred =\
    mldatasets.evaluate_class_mdl(
        fitted_lgb_mdl, X_train_nlp_fit, X_test_nlp_fit, y_train, y_test
    )
```

The preceding code produces *Figure 5.13*, shown here:
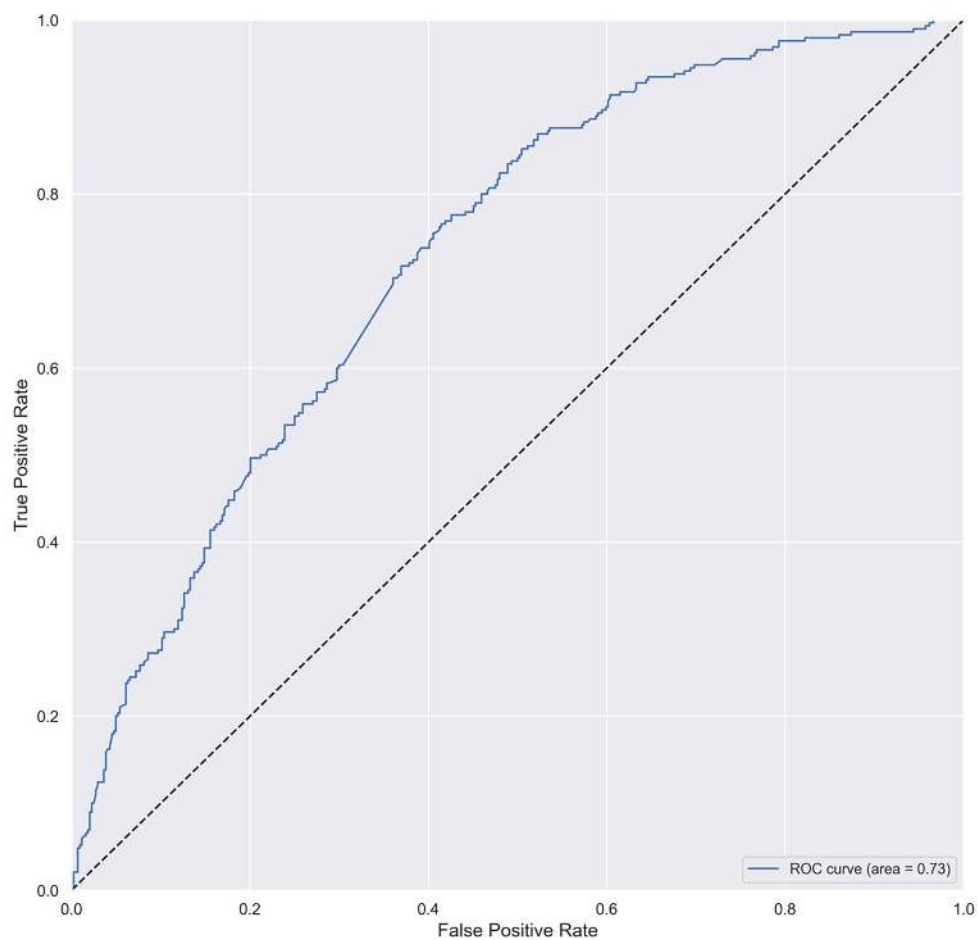

```
Accuracy_train:   0.7980        Accuracy_test:    0.6744
Precision_test:   0.6123        Recall_test:      0.4793
ROC-AUC_test:     0.7298        F1_test:          0.5377        MCC_test: 0.2973
```

Figure 5.13: The predictive performance of our LightGBM model

Figure 5.13 shows the performance achieved by LightGBM is slightly lower than for the SVM (*Figure 5.3*) but it's still pretty good, safely above the coin-toss line. The comments for the SVM about favoring precision over recall for this model also apply here.

Local interpretation for a single prediction at a time using LimeTextExplainer

To interpret any black-box model prediction with LIME, we need to specify a classifier function such as `predict_proba` for your model, and it will use this function to make predictions with perturbed data in the neighborhood of your instance and then train a linear model with it. The instance must be in its numerical form—in other words, vectorized. However, it would be easier if you could provide any arbitrary text, and it could then vectorize it on the fly. This is precisely what a pipeline can do for us. With the `make_pipeline` function from scikit-learn, you can define a sequence of estimators that transform the data, followed by one that can fit it. In this case, we just need `vectorizer` to transform our data, followed by our LightGBM model (`lgb_mdl`) that takes the transformed data, as illustrated in the following code snippet:

```
lgb_pipeline = make_pipeline(vectorizer, lgb_mdl)
```

Initializing a `LimeTextExplainer` is pretty simple. All parameters are optional, but it's recommended to specify names for your classes. Just as with `LimeTabularExplainer`, a `kernel_width` optional parameter can be critical because it defines the neighborhood's size, and there's a default that may not be optimal but can be tuned on an instance-by-instance basis. The code is illustrated here:

```
lime_lgb_explainer = LimeTextExplainer(
    class_names=['Not Highly Recomm.', 'Highly Recomm.']
)
```

Explaining an instance with `LimeTextExplainer` is similar to doing it for `LimeTabularExplainer`. The difference is that we are using a pipeline (`lgb_pipeline`), and the data we are providing (first parameter) is text since the pipeline can transform it for us. The code is illustrated in the following snippet:

```
lime_lgb_explainer.explain_instance(
    X_test_nlp[X_test_nlp.index==5].values[0],
    lgb_pipeline.predict_proba, num_features=4
).show_in_notebook(text=True)
```

According to the LIME text explainer (see *Figure 5.14*), the model predicts *Highly Recommended* for observation #5 because of the word **caramel**. At least according to the local neighborhood, **raspberry** is not a factor. In this case, the local surrogate is making a different prediction from the LightGBM model. In some cases, the LIME prediction disagrees with the model prediction. If the disagreement rate is too high, we would refer to this as "low fidelity."
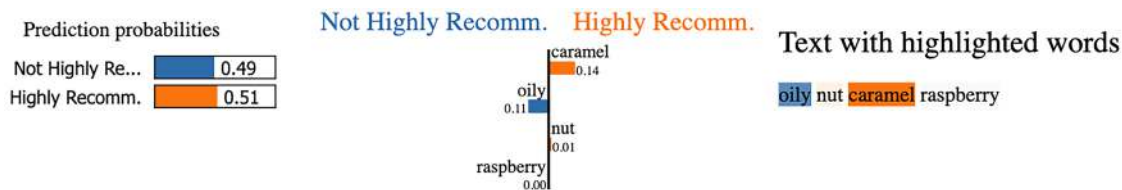
Figure 5.14: LIME's text explanation for observation #5 (Outstanding)

Now, let's contrast the interpretation for observation #5 with that of #24, as we've done before. We can use the same code but simply replace 5 with 24, as follows:

```
lime_lgb_explainer.explain_instance(
    X_test_nlp[X_test_nlp.index==24].values[0],
    lgb_pipeline.predict_proba, num_features=4
).show_in_notebook(text=True)
```

According to *Figure 5.15*, you can tell that observation #24, described as tasting like **burnt wood earthy choco**, is *Not Highly Recommended* because of the words **earthy** and **burnt**:
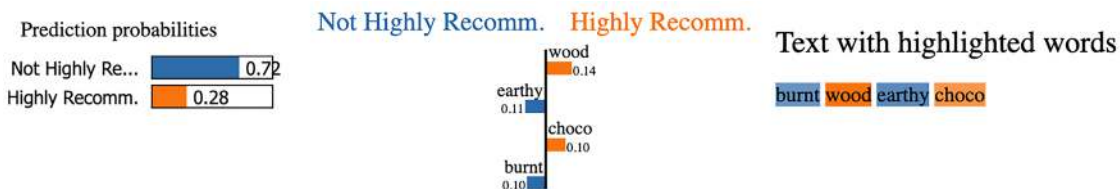
Figure 5.15: LIME's tabular explanation for observation #24 (Disappointing)

Given that we are using a pipeline that can vectorize any arbitrary text, let's have some fun with that! We will first try a phrase made out of adjectives we suspect that our model favors, then try one with unfavorable adjectives, and lastly, try using words that our model shouldn't be familiar with, as follows:

```
lime_lgb_explainer.explain_instance(
    'creamy rich complex fruity',
    lgb_pipeline.predict_proba, num_features=4
).show_in_notebook(text=True)

lime_lgb_explainer.explain_instance(
    'sour bitter roasty molasses',
    lgb_pipeline.predict_proba, num_features=4
).show_in_notebook(text=True)

lime_lgb_explainer.explain_instance(
    'nasty disgusting gross stuff',
    lgb_pipeline.predict_proba, num_features=4
).show_in_notebook(text=True)
```

In *Figure 5.16*, the explanations are spot-on for **creamy rich complex fruity** and **sour bitter roasty molasses** since the model knows these words to be either very favorable or unfavorable. These words are also common enough to be appreciated on a local level.

You can see the output here:
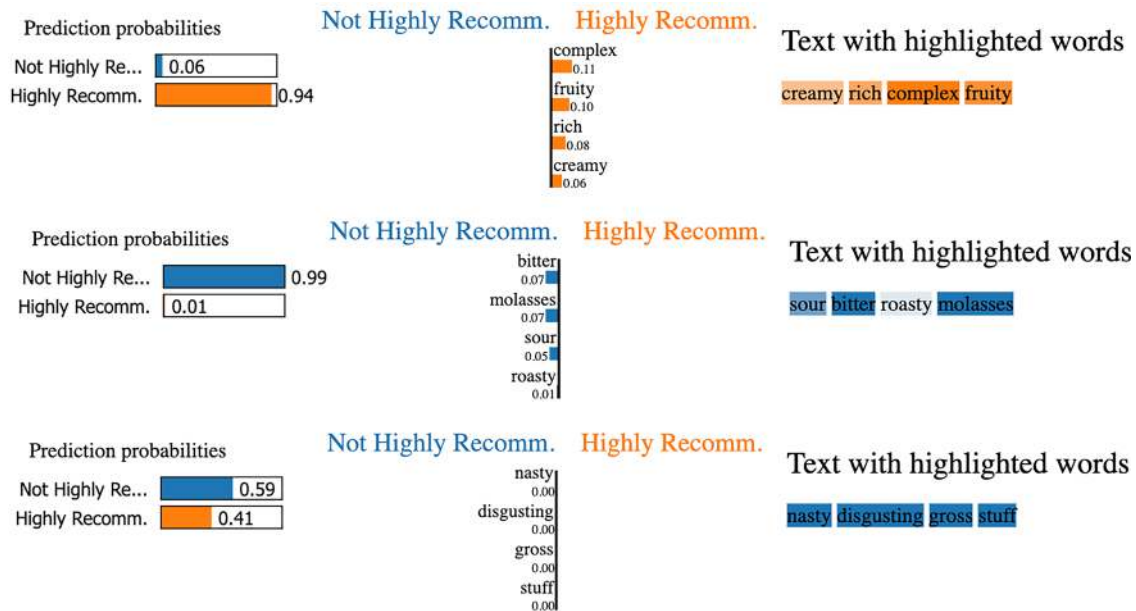
Figure 5.16: Arbitrary phrases not in the training or test dataset can be effortlessly explained with LIME, as long as words are in the corpus

However, you'd be mistaken to think that the prediction of *Not Highly Recommended* for **nasty disgusting gross stuff** has anything to do with the words. The LightGBM model hasn't seen these words before, so the prediction has more to do with *Not Highly Recommended* being the majority class, which is a good guess, and the sparse matrix for this phrase is all zeros. Therefore, LIME likely found few distant points—if any at all—in its neighborhood, so the zero coefficients of LIME's local surrogate model reflect this.

Trying SHAP for NLP

Most of SHAP's explainers will work with tabular data. `DeepExplainer` can do text but is restricted to deep learning models, and, as we will cover in *Chapter 7*, *Visualizing Convolutional Neural Networks*, three of them do images, including `KernelExplainer`. In fact, SHAP's `KernelExplainer` was designed to be a general-purpose, truly model-agnostic method, but it's not promoted as an option for NLP. It is easy to understand why: it's slow, and NLP models tend to be very complex and with hundreds—if not thousands—of features to boot. In cases such as this one, where word order is not a factor and you have a few hundred features, but the top 100 are present in most of your observations, `KernelExplainer` could work.

In addition to overcoming the high computation cost, there are a couple of technical hurdles you would need to overcome. One of them is that `KernelExplainer` is compatible with a pipeline, but it expects a single set of predictions back. But LightGBM returns two sets, one for each class: *Not Highly Recommended* and *Highly Recommended*. To overcome this problem, we can create a `lambda` function (`predict_fn`) that includes a `predict_proba` function, which returns only those predictions for *Highly Recommended*. This is illustrated in the following code snippet:

```
predict_fn = lambda X: lgb_mdl.predict_proba(X)[:,1]
```

The second technical hurdle is to do with SHAP's incompatibility with SciPy's sparse matrices, and for our explainer, we will need sample vectorized test data, which is in this format. To overcome this issue, we can convert our data in the SciPy sparse-matrix format in to a `numpy` matrix and then in to a `pandas` DataFrame (`X_test_nlp_samp_df`). To overcome any slowness, we can use the same `kmeans` trick we used last time. Other than the adjustments made to overcome obstacles, the following code is exactly the same as SHAP performed with the SVM model:

```
X_test_nlp_samp_df = pd.DataFrame(
    shap.sample(X_test_nlp_fit, 50).todense()
)
shap_lgb_explainer = shap.KernelExplainer(
    predict_fn, shap.kmeans(X_train_nlp_fit.todense(), 10)
)
shap_lgb_values_test = shap_lgb_explainer.shap_values(
    X_test_nlp_samp_df, l1_reg="num_features(20)"
)
shap.summary_plot(
    shap_lgb_values_test,
    X_test_nlp_samp_df,
    plot_type="dot",
    feature_names=vectorizer.get_feature_names_out()
)
```

By using SHAP's summary plot in *Figure 5.17*, you can tell that globally the words **creamy**, **rich**, **cocoa**, **fruit**, **spicy**, **nutty**, and **berry** have a positive impact on the model toward predicting *Highly Recommended*. On the other hand, **sweet**, **sour**, **earthy**, **hammy**, **sandy**, and **fatty** have the opposite effect. These results shouldn't be entirely unexpected given what we learned with our prior SVM model, with the tabular data and local LIME interpretations. That being said, the SHAP values were derived from samples of a sparse matrix, and they could be missing details and perhaps even be partially incorrect, especially for underrepresented features. Therefore, we should take the conclusions with a grain of salt, especially toward the bottom half of the plot. To increase the interpretation fidelity, it's best to increase sample size, but given the slowness of `KernelExplainer`, there's a trade-off to consider.

You can view the output here:
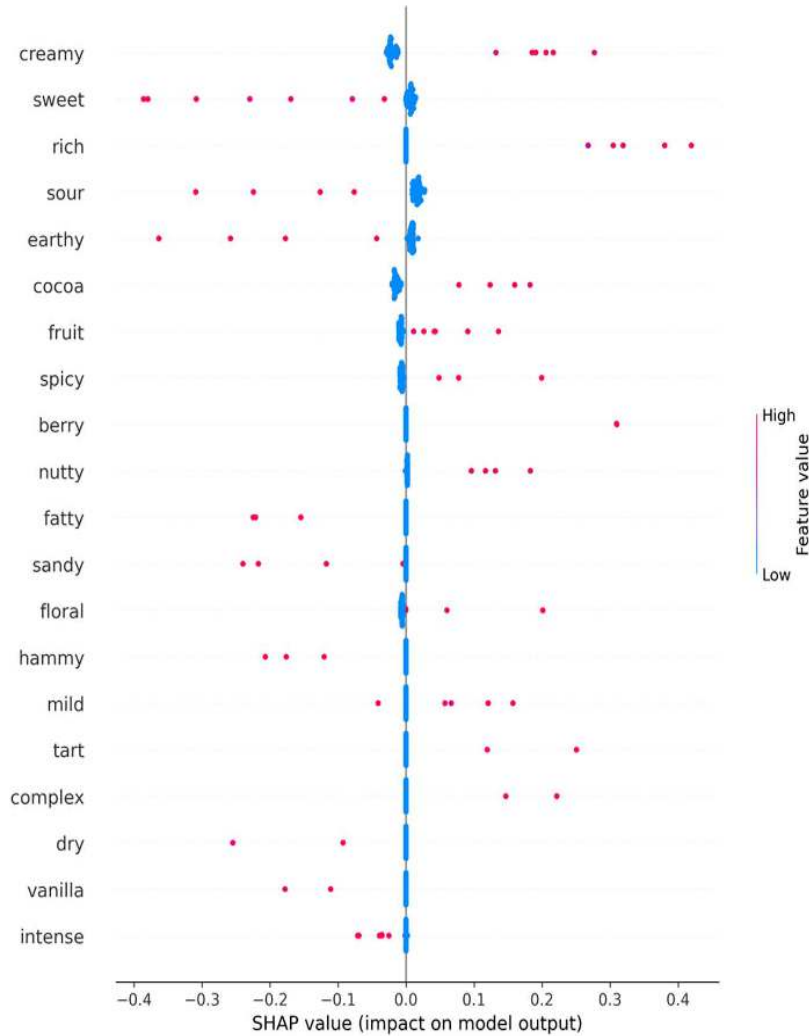
Figure 5.17: The SHAP summary plot for the LightGBM NLP model

Now that we have validated our SHAP values globally, we can use them for local interpretation with a force plot. Unlike LIME, we cannot use arbitrary data for this. With SHAP, we are limited to those data points we have previously generated SHAP values for. For instance, let's take the 18th observation from our test dataset sample, as follows:

```
print(shap.sample(X_test_nlp, 50).to_list()[18])
```

The preceding code outputs this phrase:

```
woody earthy medicinal
```

It's important to note which words are represented in the 18th observation because the `X_test_nlp_` `samp_df` DataFrame contains the vectorized representation. The 18th observation's row in this DataFrame is what you use to generate the force plot, along with the SHAP values for this observation and the expected value for the class, as illustrated in the following code snippet:

```
shap.force_plot(
    shap_lgb_explainer.expected_value,
    shap_lgb_values_test[18,:],
    X_test_nlp_samp_df.iloc[18,:],
    feature_names=vectorizer.get_feature_names_out()
)
```

Figure 5.18 is the force plot for **woody earthy medicinal**. As you can tell, **earthy** and **woody** weigh heavily in a prediction against *Highly Recommended*. The word **medicinal** is not featured in the force plot and instead you get a lack of **creamy** and **cocoa** as negative factors. As you can imagine, *medicinal* is not a word used often to describe chocolate bars, so there was only one observation in the sampled dataset that included it. Therefore, its average marginal contribution across possible coalitions would be greatly diminished:

Figure 5.18: The SHAP force plot for the 18th observation of the sampled test dataset

Let's try another one, as follows:

```
print(shap.sample(X_test_nlp, 50).to_list()[9])
```

The 9th observation is the following phrase:

```
intense spicy floral
```

Generating a `force_plot` for this observation is the same as before, except you replace 18 with 9. If you run this code, you produce the output shown here in *Figure 5.19*:

Figure 5.19: The SHAP force plot for the 9th observation of the sampled test dataset

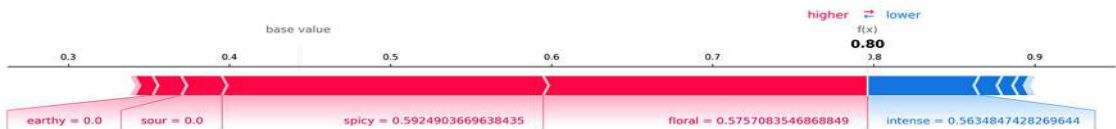As you can appreciate in *Figure 5.19*, all the words in the phrase are featured in the force plot: **floral** and **spicy** pushing toward *Highly Recommended*, and **intense** toward *Not Highly Recommended*. So, now you know how to perform both tabular and NLP interpretations with SHAP, how does it compare with LIME?

Comparing SHAP with LIME

As you will have noticed by now, both SHAP and LIME have limitations, but they also have strengths. SHAP is grounded in game theory and approximate Shapley values, so its SHAP values are supported by theory. These have great properties such as additivity, efficiency, and substitutability that make them consistent but violate the dummy property. It always adds up and doesn't need parameter tuning to accomplish this. However, it's more suited for global interpretations, and one of its most model-agnostic explainers, `KernelExplainer`, is painfully slow. `KernelExplainer` also deals with missing values by using random ones, which can put too much weight on unlikely observations.

LIME is speedy, very model-agnostic, and adaptable to all kinds of data. However, it's not grounded on strict and consistent principles but has the intuition that neighbors are alike. Because of this, it can require tricky parameter tuning to define the neighborhood size optimally, and even then, it's only suitable for local interpretations.

Mission accomplished

The mission was to understand why one of your client's bars is *Outstanding* while another one is *Disappointing*. Your approach employed the interpretation of machine learning models to arrive at the following conclusions:

- According to SHAP on the tabular model, the *Outstanding* bar owes that rating to its berry taste and its cocoa percentage of 70%. On the other hand, the unfavorable rating for the *Disappointing* bar is due mostly to its earthy flavor and bean country of origin (`Other`). Review date plays a smaller role, but it seems that chocolate bars reviewed in that period (2013–15) were at an advantage.

- LIME confirms that `cocoa_percent<=70` is a desirable property, and that, in addition to **berry**, **creamy**, **cocoa**, and **rich** are favorable tastes, while **sweet**, **sour**, and **molasses** are unfavorable.

- The commonality between both methods using the tabular model is that despite the many non-taste-related attributes, taste features are among the most salient. Therefore, it's only fitting to interpret the words used to describe each chocolate bar via an NLP model.

- The *Outstanding* bar was represented by the phrase **oily nut caramel raspberry**, of which, according to `LIMETextExplainer`, **caramel** is positive and **oily** is negative. The other two words are neutral. On the other hand, the *Disappointing* bar was represented by **burnt wood earthy choco**, of which **burnt** and **earthy** are unfavorable and the other two are favorable.

- The inconsistencies between the tastes in tabular and NLP interpretations are due to the presence of lesser-represented tastes, including **raspberry**, which is not as common as **berry**.

- According to SHAP's global explanation of the NLP model, **creamy**, **rich**, **cocoa**, **fruit**, **spicy**, **nutty**, and **berry** have a positive impact on the model toward predicting *Highly Recommended*. On the other hand, **sweet, sour, earthy, hammy, sandy**, and **fatty** have the opposite effect.

With these notions of which chocolate bar characteristics and tastes are considered less attractive by *Manhattan Chocolate Society* members, a client can apply changes to their chocolate bar formulas to appeal to a broader audience—that is, if the assumption is correct about that group being representative of their target audience.

It could be argued that it is pretty apparent that words such as **earthy** and **burnt** are not favorable words to associate with chocolate bars, while **caramel** is. Therefore, we could have reached this conclusion without machine learning! But first of all, a conclusion not informed by data would have been an opinion, and, secondly, context is everything. Furthermore, humans can't always be relied upon to place one point objectively in its context—especially considering it's among thousands of records!

Also, local model interpretation is *not only about the explanation for one prediction* because it's connected to how a model makes all predictions but, more importantly, how it makes predictions for similar points—in other words, in the local neighborhood! In the next chapter, we will expand on what it means to be in the local neighborhood by looking at the commonalities (*anchors*) and inconsistencies (*counterfactuals*) we can find there.

Summary

In this chapter, we learned how to use SHAP's `KernelExplainer`, as well as its decision and force plot to conduct local interpretations. We carried out a similar analysis using LIME's instance explainer for both tabular and text data. Lastly, we looked at the strengths and weaknesses of SHAP's `KernelExplainer` and LIME. In the next chapter, we will learn how to create even more human-interpretable explanations of a model's decisions, such as *if X conditions are met, then Y is the outcome*.

Dataset sources

- Brelinski, Brady (2020). *Manhattan Chocolate Society*: `http://flavorsofcacao.com/mcs_index.html`

Further reading

- Platt, J. C., 1999, *Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods*. Advances in Large Margin Classifiers, MIT Press: `https://www.cs.colorado.edu/~mozer/Teaching/syllabi/6622/papers/Platt1999.pdf`
- Lundberg, S. and Lee, S., 2017, *A Unified Approach to Interpreting Model Predictions*. `https://arxiv.org/abs/1705.07874` (documentation for SHAP: `https://github.com/slundberg/shap`)
- Ribeiro, M. T., Singh, S., and Guestrin, C., 2016, *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining: `http://arxiv.org/abs/1602.04938`
- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T., 2017, *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*. Advances in Neural Information Processing Systems vol. 30, pp. 3149–3157: `https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

```
https://packt.link/inml
```

6

Anchors and Counterfactual Explanations

In previous chapters, we learned how to attribute model decisions to features and their interactions with state-of-the-art global and local model interpretation methods. However, the decision boundaries are not always easy to define or interpret with these methods. Wouldn't it be nice to be able to derive human-interpretable rules from model interpretation methods? In this chapter, we will cover a few human-interpretable, local, classification-only model interpretation methods. We will first learn how to use scoped rules called **anchors** to explain complex models with statements such as *if X conditions are met, then Y is the outcome*. Then, we will explore **counterfactual** explanations that follow the form *if Z conditions aren't met, then Y is not the outcome*.

These are the main topics we are going to cover in this chapter:

- • Understanding anchor explanations
- • Exploring counterfactual explanations

Let's begin!

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `catboost`, `matplotlib`, `seaborn`, `alibi`, `tensorflow`, `shap`, and `witwidget` libraries. Instructions on how to install all of these libraries are in the *Preface*.

> The code for this chapter is located here: `https://packt.link/tH0y7`.

The mission

In the United States, for the last two decades, private companies and nonprofits have developed criminal **Risk Assessment Instruments/Tools** (**RAIs**), most of which employ statistical models. As many states can no longer afford their large prison populations, these methods have increased in popularity, *guiding* judges and parole boards through every step of the prison system.

These are high-impact decisions that can determine if a person is released from prison. Can we afford for these decisions to be wrong? Can we accept the recommendations from these systems without understanding why they were made? Worst of all, we don't exactly know how an assessment was made. The risk is usually calculated with a white-box model, but, in practice, a black-box model is used because it is proprietary. Predictive performance is also relatively low, with median AUC scores for a sample of nine tools ranging between 0.57 and 0.74 according to the paper *Performance of Recidivism Risk Assessment Instruments in U.S. Correctional Settings*.

Even though traditional statistical methods are still the norm for criminal justice models, to improve performance, some researchers have proposed leveraging more complex models, such as Random Forest with larger datasets. Far from being science fiction drawn from *Minority Report* or *Black Mirror*, in some countries, scoring people based on their likelihood of engaging in antisocial, or even antipatriotic, behavior with big data and machine learning is already a reality.

As more and more AI solutions attempt to make life-changing predictions about us with our data, fairness must be properly assessed, and all its ethical and practical implications must be adequately discussed. *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, covered how fairness is an integral concept for machine learning interpretation. You can evaluate fairness in any model, but fairness is especially difficult when it involves human behavior. The dynamics between human psychological, neurological, and sociological factors are extremely complicated. In the context of predicting criminal behavior, it boils down to what factors are potentially to blame for a crime, as it wouldn't be fair to include anything else in a model, and how these factors interact.

Quantitative criminologists are still debating the best predictors of criminality and their root causes. They're also debating whether it is ethical to *blame* a criminal for these factors to begin with. Thankfully, demographic traits such as race, gender, and nationality are no longer used in criminal risk assessments. But this doesn't mean that these methods are no longer biased. Scholars recognize the problem and are proposing solutions.

This chapter will examine racial bias in one of the most widely used risk assessment tools. Given this topic's sensitive and relevant nature, it was essential to provide a modicum of context about criminal risk assessment tools and how machine learning and fairness connect with all of them. We won't go into much more detail, but understanding the context is important to appreciate how machine learning could perpetuate structural inequality and unfair biases.

Now, let's introduce you to your mission for this chapter!

Unfair bias in recidivism risk assessments

Imagine a scenario of an investigative journalist writing an article on an African American defendant detained while awaiting trial. A tool called **Correctional Offender Management Profiling for Alternative Sanction** (**COMPAS**) deemed him as being at risk of recidivism. **Recidivism** is when someone relapses into criminal behavior. And the score convinced the judge that the defendant had to be detained without considering any other arguments or testimonies. He was locked up for many months, and in the trial, was found not guilty. Over five years have passed since the trial, and he hasn't been accused of any crime. You could say the prediction for recidivism was a false positive.

The journalist has reached out to you because she would like to ascertain with data science whether there was unfair bias in this case. The COMPAS risk assessment is computed using 137 questions (`https://www.documentcloud.org/documents/2702103-Sample-Risk-Assessment-COMPAS-CORE.html`). It includes questions such as the following:

- "Based on the screener's observations, is this person a suspected or admitted gang member?"
- "How often have you moved in the last 12 months?"
- "How often do you have barely enough money to get by?"
- Psychometric LIKERT scale questions such as "I have never felt sad about things in my life."

Even though race is not one of the questions, many of these questions may correlate with race. Not to mention, in some cases, they can be more a question of subjective opinion than fact, and thus be prone to bias.

The journalist cannot provide you with the 137 answered questions or the COMPAS model because this data is not publicly available. However, thankfully, all defendants' demographic and recidivism data for the same county in Florida is available.

The approach

You have decided to do the following:

- **Train a proxy model:** You don't have the original features or model, but all is not lost because you have the COMPAS scores—the labels. And we also have features relevant to the problem we can connect to these labels with models. By approximating the COMPAS model via the proxies, you can assess the fairness of the COMPAS decisions. In this chapter, we will train a CatBoost model.
- **Anchor explanations:** Using this method will unearth insights into why the proxy model makes specific predictions using a series of rules called anchors, which tell you where the decision boundaries lie. The boundaries are relevant for our mission because we want to know why the defendant has been wrongfully predicted to recidivate. It's an approximate boundary to the original model, but there's still some truth to it.
- **Counterfactual explanations:** While Anchor explains why a decision was made, counterfactuals can be useful to examine why a decision was not made. This is particularly useful in inspecting the fairness of decisions. We will use an unbiased method to find counterfactuals and then use the **What-If Tool** (**WIT**) to explore counterfactuals and fairness further.

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/06/Recidivism.ipynb`.

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate the dataset
- `sklearn` (scikit-learn), and `catboost` to split the data and fit the models
- `matplotlib`, `seaborn`, `alibi`, `tensorflow`, `shap`, and `witwidget` to visualize the interpretations

You should load all of them first:

```
import math
import mldatasets
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import metrics
from catboost import CatBoostClassifier
import matplotlib.pyplot as plt
import seaborn as sns
from alibi.utils.mapping import ohe_to_ord, ord_to_ohe
import tensorflow as tf
from alibi.explainers import AnchorTabular, CounterFactualProto
import shap
import witwidget
from witwidget.notebook.visualization import WitWidget, \ WitConfigBuilder
```

Let's check that TensorFlow has loaded the right version with `print(tf.__version__)`. It should be 2.0 or above. We should also disable eager execution and verify that it worked with this command. The output should say that it's `False`:

```
tf.compat.v1.disable_eager_execution()
print('Eager execution enabled:', tf.executing_eagerly())
```

Understanding and preparing the data

We load the data like this into a DataFrame called `recidivism_df`:

```
recidivism_df = mldatasets.load("recidivism-risk", prepare=True)
```

There should be almost 15,000 records and 23 columns. We can verify this was the case with `info()`:

```
recidivism_df.info()
```

The following output checks out. All features are numeric with no missing values, and categorical features have already been one-hot-encoded for us:

```
Int64Index: 14788 entries, 0 to 18315
Data columns (total 23 columns):
 #   Column                 Non-Null Count  Dtype
--   ------                 --------------  -----
 0   age                    14788 non-null  int8
 1   juv_fel_count          14788 non-null  int8
 2   juv_misd_count         14788 non-null  int8
 3   juv_other_count        14788 non-null  int64
 4   priors_count           14788 non-null  int8
 5   is_recid               14788 non-null  int8
 6   sex_Female             14788 non-null  uint8
 7   sex_Male               14788 non-null  uint8
 8   race_African-American  14788 non-null  uint8
 9   race_Asian             14788 non-null  uint8
13   race_Other             14788 non-null  uint8
14   c_charge_degree_(F1)   14788 non-null  uint8
15   c_charge_degree_(F2)   14788 non-null  uint8
21   c_charge_degree_Other  14788 non-null  uint8
22   compas_score           14788 non-null  int64
```

The data dictionary

There are only nine features, but they become 22 columns because of the categorical encoding:

- `age`: Continuous; the age of the defendant (between 18 and 96).
- `juv_fel_count`: Ordinal; the number of juvenile felonies (between 0 and 2).
- `juv_misd_count`: Ordinal; the number of juvenile misdemeanors (between 0 and 1).
- `juv_other_count`: Ordinal; the number of juvenile convictions that are neither felonies nor misdemeanors (between 0 and 1).
- `priors_count`: Ordinal; the number of prior crimes committed (between 0 and 13).
- `is_recid`: Binary; did the defendant recidivate within 2 years (1 for yes, 0 for no)?
- `sex`: Categorical; the gender of the defendant.
- `race`: Categorical; the race of the defendant.
- `c_charge_degree`: Categorical; the degree of what the defendant is currently being charged with. The United States classifies criminal offenses as felonies, misdemeanors, and infractions, ordered from most serious to least. These are subclassified in the form of degrees, which go from 1st (most serious offenses) to 3rd or 5th (least severe). However, even though this is standard for federal offenses, it is tailored to state law. For felonies, Florida (`http://www.dc.state.fl.us/pub/scoresheet/cpc_manual.pdf`) has a level system that determines the severity of a crime regardless of the degree, and this goes from 10 (most severe) to 1 (least).

The categories of this feature are prefixed with *F* for felonies and *M* for misdemeanors. They are followed by a number, which is a level for felonies and a degree for misdemeanors.

- `compas_score`: Binary; COMPAS scores defendants as "low," "medium," or "high" risk. In practice, "medium" is often treated as "high" by decision-makers, so this feature has been converted to binary to reflect this behavior: 1: high/medium risk and 0: low risk.

Examining predictive bias with confusion matrices

There are two binary features in the dataset. The first one is the recidivism risk prediction made by COMPAS (`compas_score`). The second one (`is_recid`) is the *ground truth* because it's what happened within 2 years of the defendant's arrest. Just as you would with the prediction of any model against its training labels, you can build confusion matrices with these two features. scikit-learn can produce one with the `confusion_matrix` function (`cf_matrix`), and we can then create a Seaborn `heatmap` with it. Instead of plotting the number of **True Negatives** (**TNs**), **False Positives** (**FPs**), **False Negatives** (**FNs**), and **True Positives** (**TPs**), we can plot percentages with a simple division (`cf_matrix/np.sum(cf_matrix)`). The other parameters of `heatmap` only assist with formatting:

```
cf_matrix = metrics.confusion_matrix(
    recidivism_df.is_recid,
    recidivism_df.compas_score
)
sns.heatmap(
    cf_matrix/np.sum(cf_matrix),
    annot=True,
    fmt='.2%',
    cmap='Blues',
    annot_kws={'size':16}
)
```

The preceding code outputs *Figure 6.1*. The top-right corner is FPs, which are nearly one-fifth of all predictions, and together with the FNs in the bottom-left corner, they make up over two-thirds:
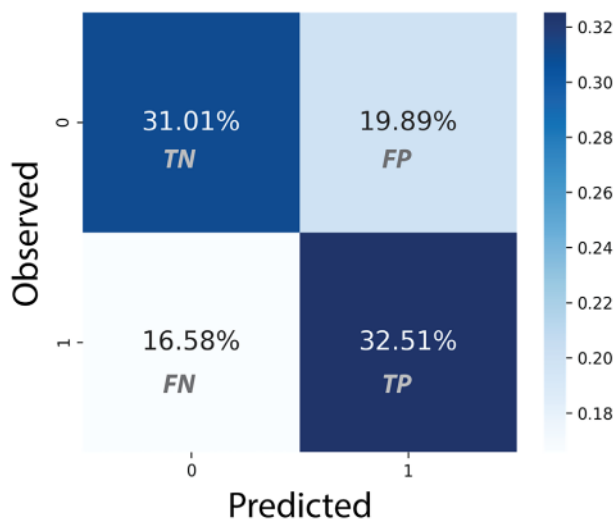
Figure 6.1: Confusion matrix between the predicted risk of recidivism (compas_score) and the ground truth (is_recid)

Figure 6.1 shows that the COMPAS model's predictive performance is not very good, especially if we assume that criminal justice decision-makers are taking medium or high risk assessments at face value. It also tells us that FP and FNs occur at a similar rate. Nevertheless, simple visualizations such as the confusion matrix obscure predictive disparities between subgroups of a population. We can quickly compare disparities between two subgroups that historically have been treated differently by the United States criminal justice system. To this end, we first subdivide our DataFrame into two DataFrames: one for Caucasians (`recidivism_c_df`) and another for African Americans (`recidivism_aa_df`). Then we can generate confusion matrices for each DataFrame and plot them side by side with the following code:

```python
recidivism_c_df =\
recidivism_df[recidivism_df['race_Caucasian'] == 1]
recidivism_aa_df =\
recidivism_df[recidivism_df['race_African-American'] == 1]

_ = mldatasets.compare_confusion_matrices(
    recidivism_c_df.is_recid,
    recidivism_c_df.compas_score,
    recidivism_aa_df.is_recid,
    recidivism_aa_df.compas_score,
    'Caucasian',
    'African-American',
    compare_fpr=True
)
```

The preceding snippet generated *Figure 6.2*. At a glance, you can tell that it's like the confusion matrix for Caucasians has been flipped 90 degrees to form the African American confusion matrix, and even then, it is still less unfair. Pay close attention to the difference between FPs and TNs. As a Caucasian defendant, a result is more than half as likely to be an FP than a TN, but as an African American, it is a few percentage points more likely. In other words, an African American defendant who doesn't re-offend is predicted to be at risk of recidivating more than half of the time:


```
African-American FPR:    53.1%
Caucasian FPR:           28.8%
Ratio FPRs:              1.84 x
```
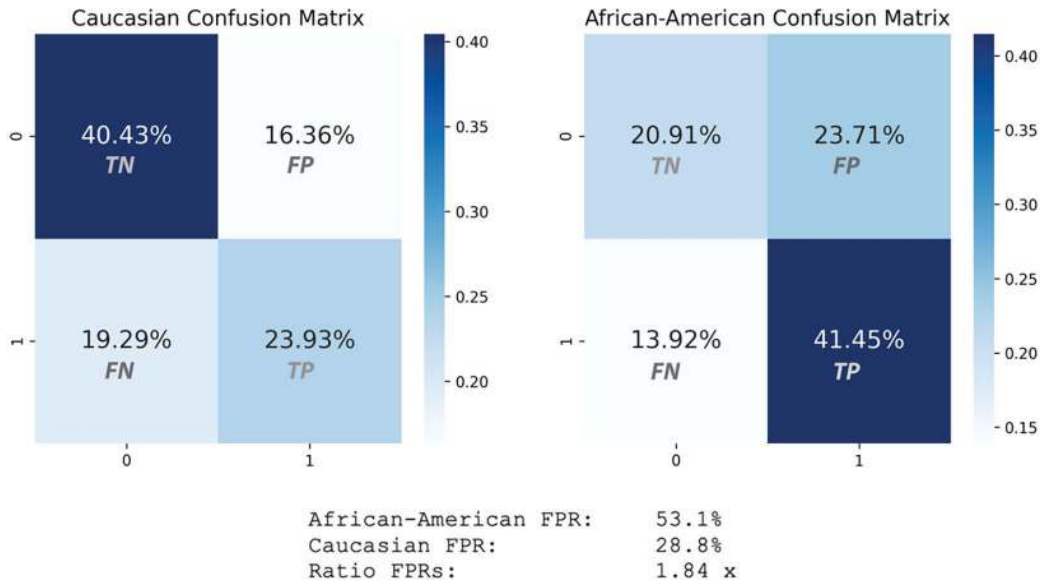
Figure 6.2: Comparison of the confusion matrices for the predicted risk of recidivism (compas_score) and the ground truth (is_recid) between African Americans and Caucasians in the dataset

Instead of eyeballing it by looking at the plots, we can measure the **False Positive Rate** (**FPR**), which is the ratio between these two measures (FP / (FP + TN)) at risk of recidivism more often.

Data preparation

Before we move on to the modeling and interpretation, we have one last step.

Since `prepare=True` for the data loading, all we do now is split the data into a training and test dataset. As usual, it is critical to set your random states so that all your findings are reproducible. We will then set `y` to be our target variable (`compas_score`) and set `X` as every other feature except for `is_recid`, because this is the ground truth. Lastly, we split `y` and `X` into train and test datasets as we have before:

```python
rand = 9
np.random.seed(rand)
y = recidivism_df['compas_score']
X = recidivism_df.drop(
    ['compas_score', 'is_recid'],
    axis=1).copy()
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=rand
)
```

Now, let's get started!

Modeling

First, let's quickly train the model we will use throughout this chapter.

Proxy models are a means to emulate output from a black-box model just like **global surrogate models**, which we covered in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*. So, are they the same thing? In machine learning, surrogate and proxy are terms that are often used interchangeably. However, semantically, surrogacy relates to substitution and proxy relates more to a representation. So, we call these proxy models to distinguish that we don't have the exact training data. Therefore, you only represent the original model because you cannot substitute it. For the same reason, unlike interpretation with surrogates, which is best served by simpler models, a proxy is best suited to complex models that can make up for the difference in training data with complexity.

We will train a **CatBoost** classifier. For those of you who aren't familiar with CatBoost, it's an efficient boosted ensemble tree method. It's similar to **LightGBM**, except it uses a new technique called **Minimal Variance Sampling** (**MVS**) instead of **Gradient-Based One-Side Sampling** (**GOSS**). Unlike LightGBM, it grows trees in a balanced fashion. It's called CatBoost because it can automatically encode categorical features, and it's particularly good at tackling overfitting, with unbiased treatment of categorical features and class imbalances. We won't go into a whole lot of detail, but it was chosen for this exercise for those reasons.

As a tree-based model class, you can specify a maximum `depth` value for `CatBoostClassifier`. We are setting a relatively high `learning_rate` value and a lower `iterations` value (the default is 1,000). Once we have used `fit` on the model, we can evaluate the results with `evaluate_class_mdl`:

```
cb_mdl = CatBoostClassifier(
    iterations=500,
    learning_rate=0.5,
    depth=8
)
fitted_cb_mdl = cb_mdl.fit(
    X_train,
    y_train,
    verbose=False
)


y_train_cb_pred, y_test_cb_prob, y_test_cb_pred = \
mldatasets.evaluate_class_mdl(
    fitted_cb_mdl, X_train, X_test, y_train, y_test
)
```

You can appreciate the output of `evaluate_class_mdl` for our CatBoost model in *Figure 6.3*:
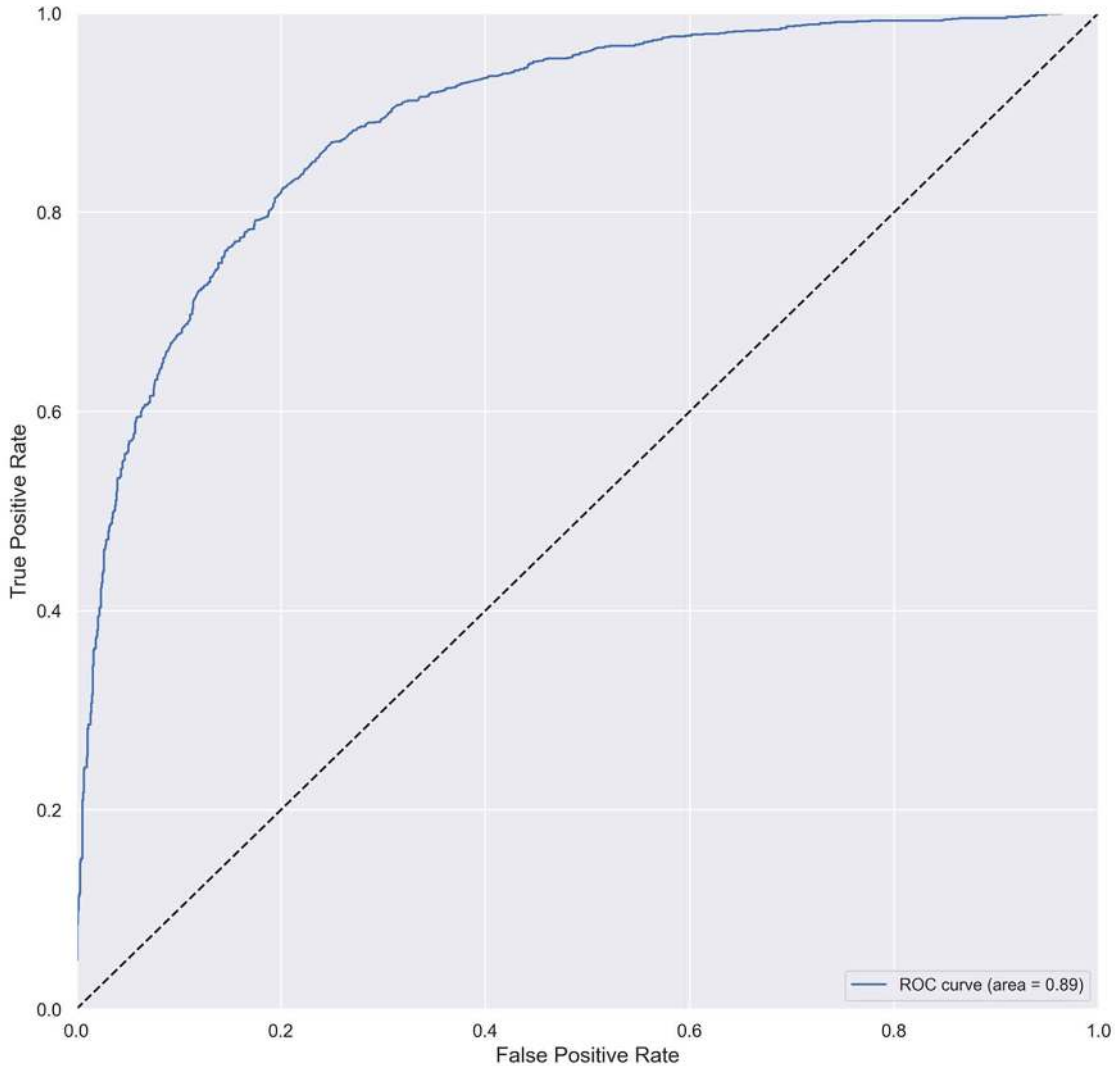

```
Accuracy_train:    0.8790        Accuracy_test:    0.8087
Precision_test:    0.8277        Recall_test:      0.8110
ROC-AUC_test:      0.8927        F1_test:          0.8193        MCC_test: 0.6162
```

Figure 6.3: Predictive performance of our CatBoost model

From the optics of fairness, we care more about FPs than FNs because it's more unfair to put an *innocent* person in prison than it is to leave a *guilty* person on the streets. Therefore, we should aspire to have higher *precision* than *recall*. *Figure 6.3* confirms this, as well as a healthy ROC curve, ROC-AUC, and MCC.

The predictive performance for the model is reasonably accurate considering it's a *proxy model* meant to only approximate the real thing with different, yet related, data.

Getting acquainted with our "instance of interest"

The journalist reached out to you with a case in mind: the African American defendant who was falsely predicted to have a high risk of recidivism. This case is #5231 and is your main *instance of interest*. Since our focus is racial bias, we'd like to compare it with similar instances but of different races. To that end, we found case #10127 (Caucasian) and #2726 (Hispanic).

We can look at the data for all three. Since we will keep referring to these instances throughout this chapter, let's first save the indexes of the African American (`idx_aa`), Hispanic (`idx_h`), and Caucasian (`idx_c`) cases. Then, we can subset the test dataset by these indexes. Since we have to make sure that our predictions match, we will concatenate this subsetted test dataset to the true labels (`y_test`) and the CatBoost predictions (`y_test_cb_pred`):

```python
idx_aa = 5231
idx_h = 2726
idx_c = 10127
eval_idxs = X_test.index.isin([idx_aa, idx_h, idx_c])
X_test_evals = X_test[eval_idxs]
eval_compare_df = pd.concat(
    [
        pd.DataFrame(
            {'y':y_test[eval_idxs]},
            index=[idx_c, idx_h, idx_aa]
        ),
        pd.DataFrame(
            {'y_pred':y_test_cb_pred[eval_idxs]},
            index=[idx_c, idx_h, idx_aa]
        ),
        X_test_evals
    ],
    axis=1
).transpose()
eval_compare_df
```

The preceding code produces the DataFrame in *Figure 6.4*. You can tell that the predictions match the true labels, and our main *instance of interest* was the only one predicted as a medium or high risk of recidivism. Besides race, the only other differences are with `c_charge_degree` and one minor age difference:

	10127	2726	5231
y	0	0	1
y_pred	0	0	1
age	24	23	23
⋮	⋮	⋮	⋮
priors_count	2	2	2
sex_Female	0	0	0
sex_Male	1	1	1
race_African-American	0	0	1
race_Asian	0	0	0
race_Caucasian	1	0	0
race_Hispanic	0	1	0
⋮	⋮	⋮	⋮
c_charge_degree_(F3)	0	1	0
c_charge_degree_(F7)	0	0	1
c_charge_degree_(M1)	1	0	0
⋮	⋮	⋮	⋮

Figure 6.4: Observations #5231, #10127, and #2726 side by side with feature differences highlighted

Throughout this chapter, we will pay close attention to these differences to see whether they played a large role in producing the prediction difference. All the methods we will cover will complete the picture of what can determine or change the proxy model's decision, and, potentially, the COMPAS model by extension. Now that we have completed the setup, we will move forward with employing the interpretation methods.

Understanding anchor explanations

In *Chapter 5*, *Local Model-Agnostic Interpretation Methods*, we learned that **LIME** trains a local surrogate model (specifically a **weighted sparse linear model**) on a **perturbed** version of your dataset in the **neighborhood** of your *instance of interest*. The result is that you approximate a **local decision boundary** that can help you interpret the model's prediction for it.

Like LIME, **anchors** are also derived from a model-agnostic perturbation-based strategy. However, they are not about the *decision boundary* but the **decision region**. Anchors are also known as **scoped rules** because they list some **decision rules** that apply to your instance and its *perturbed* neighborhood. This neighborhood is also known as the **perturbation space**. An important detail is to what extent the rules apply to it, known as **precision**.

Imagine the neighborhood around your instance. You would expect the points to have more similar predictions the closer you got to your instance, right? So, if you had decision rules that defined these predictions, the smaller the area surrounding your instance, the more precise your rules. This concept is called **coverage**, which is the percentage of your *perturbation space* that yields a specific *precision*.

Unlike LIME, anchors don't fit a local surrogate model to explain your chosen instance's prediction. Instead, they explore possible candidate decision rules using an algorithm called **Kullback-Leibler divergence Lower and Upper Confidence Bounds** (**KL-LUCB**), which is derived from a **Multi-Armed Bandit** (**MAB**) algorithm.

MABs are a family of *reinforcement learning algorithms* about maximizing the payoff when you have limited resources to explore all unknown possibilities. The algorithm originated from understanding how casino slot machine players could maximize their payoff by playing multiple machines. It's called multi-armed bandit because slot machine players are known as one-armed bandits. Yet players don't know which machine will yield the highest payoff, can't try all of them at once, and have finite funds. The trick is to learn how to balance exploration (trying unknown slot machines) with exploitation (using those you already have reasons to prefer).

In the anchors' case, each slot machine is a potential decision rule, and the payoff is how much precision it yields. The KL-LUCB algorithm uses confidence regions based on the **Kullback-Leibler divergence** between the distributions to find the decision rule with the highest precision sequentially, yet efficiently.

Preparations for anchor and counterfactual explanations with alibi

Several small steps need to be performed to help the `alibi` library produce human-friendly explanations. The first one pertains to the prediction since the model may output a 1 or 0, but it's easier to understand a prediction by its name. To help us with this, we need a list with the class names where the 0 position matches our negative class name and 1 matches the positive one:

```
class_names = ['Low Risk', 'Medium/High Risk']
```

Next, let's create a `numpy` array with our main *instance of interest* and print it out. Please note that the single-dimension array needs to be expanded (`np.expand_dims`) so that it's understood by `alibi`:

```
X_test_eval = np.expand_dims(
    X_test.values[
        X_test.index.get_loc(idx_aa)
    ],
     axis=0
)
print(X_test_eval)
```

The preceding code outputs an array with the 21 features, of which 12 were the result of **One-Hot Encoding (OHE)**:

```
[[23 0 0 0 2 0 1 1 0 ... 0 1 0 0 0 0]]
```

A problem with making human-friendly explanations arises when you have OHE categories. To both the machine learning model and the explainer, each OHE feature is separate from the others. Still, to the human interpreting the outcomes, they cluster together as categories of their original features.

The `alibi` library has several utility functions to deal with this problem, such as `ohe_to_ord`, which takes a one-hot-encoded instance and puts it in an ordinal format. To use this function, we first define a dictionary (`cat_vars_ohe`) that tells `alibi` where the categorical variables are in our features and how many categories each one has. For instance, in our data, gender starts at the 5th index and has two categories, which is why our `cat_vars_ohe` dictionary begins with `5: 2`. Once you have this dictionary, `ohe_to_ord` can take your instance (`X_test_eval`) and output it in ordinal format, where each categorical variable takes up a single feature. This utility function will prove useful for Alibi's counterfactual explanations, where the explainer will need this dictionary to map categorical features together:

```
cat_vars_ohe = {5: 2, 7: 6, 13: 8}
print(ohe_to_ord(X_test_eval, cat_vars_ohe)[0])
```

The preceding code outputs the following array:

```
[[23 0 0 0 2 1 0 3]]
```

For when it's in ordinal format, Alibi will need a dictionary that provides names for each category and a list of feature names:

```
category_map = {
    5: ['Female', 'Male'],
    6: [
        'African-American',
        'Asian',
        'Caucasian',
        'Hispanic',
        'Native American',
        'Other'
    ],
    7: [
        'Felony 1st Degree',
        'Felony 2nd Degree',
        'Felony 3rd Degree',
        'Felony 7th Degree',
        'Misdemeanor 1st Degree',
        'Misdemeanor 2nd Degree',
        'Misdemeanor 3rd Degree',
```

```
        'Other Charge Degree'
    ]
}
feature_names = [
    'age',
    'juv_fel_count',
    'juv_misd_count',
    'juv_other_count',
    'priors_count',
    'sex',
    'race',
    'c_charge_degree'
]
```

However, Alibi's anchor explanations use the data as it is provided to our models. We are using OHE data, so we need a category map for that format. Of course, the OHE features are all binary, so they only have two "categories" each:

```
category_map_ohe = {
    5: ['Not Female', 'Female'],
    6: ['Not Male', 'Male'],
    7:['Not African American', 'African American'],
    8:['Not Asian', 'Asian'], 9:['Not Caucasian', 'Caucasian'],
    10:['Not Hispanic', 'Hispanic'],
    11:['Not Native American', 'Native American'],
    12:['Not Other Race', 'Other Race'],
    19:['Not Misdemeanor 3rd Deg', 'Misdemeanor 3rd Deg'],
    20:['Not Other Charge Degree', 'Other Charge Degree']
}
```

Local interpretations for anchor explanations

All Alibi explainers require a `predict` function, so we create a `lambda` function called `predict_cb_fn` for our CatBoost model. Please note that we are using `predict_proba` for the classifier's probabilities. Then, to initialize `AnchorTabular`, we also provide it with our features' names as they are in our OHE dataset and the category map (`category_map_ohe`). Once it has initialized, we fit it with our training data:

```
predict_cb_fn = lambda x: fitted_cb_mdl.predict_proba(x)
anchor_cb_explainer = AnchorTabular(
    predict_cb_fn,
    X_train.columns,
    categorical_names=category_map_ohe
)
anchor_cb_explainer.fit(X_train.values)
```

Before we leverage the explainer, it's good practice to check that the anchor "holds." In other words, we should check that the MAB algorithm found decision rules that help explain the prediction. To verify this, you use the `predictor` function to check that the prediction is the same as the one you expect for this instance. Right now, we are using `idx_aa`, which is the case of the African American defendant:

```python
print(
    'Prediction: %s' %  class_names[anchor_cb_explainer.
    predictor(X_test.loc[idx_aa].values)[0]]
)
```

The preceding code outputs the following:

```
Prediction: Medium/High Risk
```

We can proceed to use the `explain` function to generate an explanation for our instance. We can set our precision threshold to `0.85`, which means we expect the predictions on anchored observations to be the same as our instance at least 85% of the time. Once we have an explanation, we can print the anchors as well as their precision and coverage:

```python
anchor_cb_explanation = anchor_cb_explainer.explain(
    X_test.loc[idx_aa].values,threshold=0.85, seed=rand
)

print('Anchor: %s' % (' AND'.join(anchor_cb_explanation.anchor)))
print('Precision: %.3f' % anchor_cb_explanation.precision)
print('Coverage: %.3f' % anchor_cb_explanation.coverage)
```

The following output was generated by the preceding code. You can tell that `age`, `priors_count`, and `race_African-American` are factors at 86% precision. Impressively, this rule applies to almost a third of all the perturbation space's instances with a coverage of 0.29:

```
Anchor: age <= 25.00 AND
    priors_count > 0.00 AND
    race_African-American = African American
Precision: 0.863
Coverage: 0.290
```

We can try the same code but with a 5% bump in the precision threshold set to 0.9. We observe the same three anchors that were generated from the previous example with three additional anchors:

```
Anchor: age <= 25.00 AND
    priors_count > 0.00 AND
    race_African-American = African American AND
    c_charge_degree_(M1) = Not Misdemeanor 1st Deg AND
    c_charge_degree_(F3) = Not Felony 3rd Level AND
    race_Caucasian = Not Caucasian
```

```
Precision: 0.903
Coverage: 0.290
```

Interestingly enough, although precision did increase by a few percentage points, coverage stayed the same. At this level of precision, we may confirm that race is a significant factor because being African American is an anchor, but so is not being Caucasian. Another factor was `c_charge_degree`. The explanation reveals that being accused of a first-degree misdemeanor or third-level felony would have been better. Understandably, a seventh-level felony is a more serious charge than these two.

Another way of understanding why a model made a specific prediction is by looking for a similar datapoint that had the opposite prediction and examining why the alternative decision was made. The decision boundary crosses between both points, so it's helpful to contrast decision explanations from both sides of the boundary. This time, we will use `idx_c`, which is the case for the Caucasian defendant with a threshold of 85% and which outputs the anchors as follows:

```
Anchor: priors_count <= 2.00 AND
    race_African-American = Not African American AND
    c_charge_degree_(M1) = Misdemeanor 1st Deg
Precision: 0.891
Coverage: 0.578
```

The first anchor is `priors_count <= 2.00`, but on the other side of the boundary, the first two anchors were `age <= 25.00` and `priors_count > 0.00`. In other words, for an African American under or equal to the age of 25, any number of priors is enough to categorize them as having a medium/high risk of recidivism (86% of the time). On the other hand, a Caucasian person will be predicted as low risk if the priors don't exceed two and they were not accused of a first-degree misdemeanor (89% of the time and with 58% coverage). These decision rules suggest a biased decision based on race with different standards applied to different racial groups. A double standard is when different rules are applied when, in principle, the situation is the same. In this case, the different rules for `priors_count` and the absence of `age` as a factor for Caucasians constitute double standards.

We can now try a Hispanic defendant (`idx_h`) to observe whether double standards are also found in this instance. We just run the same code as before but replace `idx_c` with `idx_h`:

```
Anchor: priors_count <= 2.00 AND
    race_African-American = Not African American AND
    juv_fel_count <= 0.00 AND
    sex_Male = Male
Precision: 0.851
Coverage: 0.578
```

The explanations for the Hispanic defendant confirm the different standard with `priors_count` and that `race` continues to be a strong factor since there's one anchor for not being African American and another one for being Hispanic.

For specific model decisions, anchor explanations answer the question *why?* However, by comparing similar instances that are only slightly different but have different predictions, we have explored the question *what if?* In the next section, we will expand on this question further.

Exploring counterfactual explanations

Counterfactuals are an integral part of human reasoning. How many of us have muttered the words "If I had done *X* instead, my outcome *y* would have been different"? There's always one or two things that, if done differently, could lead to the outcomes we prefer!

In machine learning outcomes, you can leverage this way of reasoning to make for extremely human-friendly explanations where we can explain decisions in terms of what would need to change to get the opposite outcome (the **counterfactual class**). After all, we are often interested in knowing how to make a negative outcome better. For instance, how do you get your denied loan application approved or decrease your risk of cardiovascular disease from high to low? However, hopefully, answers to those questions aren't a huge list of changes. You prefer the smallest number of changes required to change your outcome.

Regarding fairness, counterfactuals are an important interpretation method, in particular when there are elements involved that *we can't change* or shouldn't have to change. For instance, if you perform exactly the same job and have the same level of experience as your co-worker, you expect to have the same salary, right? If you and your spouse share the same assets and credit history but have different credit scores, you have to wonder why. Does it have to do with gender, race, age, or even political affiliations? Whether it's a compensation, credit rating, or recidivism risk model, you'd hope that similar people were treated similarly.

Finding counterfactuals is not particularly hard. All we have to do is change our *instance of interest* slightly until it changes the outcome. And maybe there's an instance already in the dataset just like that!

In fact, you could say that the three instances we examined with anchors in the previous section are close enough to be counterfactuals of each other, except for the Caucasian and Hispanic cases, which have the same outcome. But the Caucasian and Hispanic instances were "*cherry-picked*" by looking for datapoints with the same criminal history but different races to the *instance of interest*. Perhaps by comparing similar points, mostly except for race, we limited the scope in such a way that we confirmed what we hoped to confirm, which is that race matters to the model's decision-making.

This is an example of *selection bias*. After all, counterfactuals are inherently selective because they focus on a few feature changes. And even with a few features, there are so many possible permutations that change the outcome, which means that a single point could have hundreds of counterfactuals. And not all of these will tell a consistent story. This phenomenon is called the **Rashomon effect.** It is named after a famous Japanese movie about a murder mystery. And as we have come to expect from murder mysteries, witnesses have different recollections of what happened. But in the same way that it's difficult to rely on a single witness, you cannot rely on a single counterfactual. Also, in the same way that great detectives are trained to look for clues everywhere in connection to the scene of a crime (even if it contradicts their instincts), counterfactuals shouldn't be "cherry-picked" because they conveniently tell the story we want them to tell.

Fortunately, there are algorithmic ways of looking for counterfactual instances in an unbiased manner. Typically, these involve finding the closest points with different outcomes, but there are different ways of measuring the distance between points. For starters, there's the **L1** distance (also known as the **Manhattan distance**) and **L2** distance (also known as the **Euclidean distance**), among many others. But there's also the question of normalizing the distances because not all features have the same scale. Otherwise, they would be biased against features with smaller scales, such as one-hot-encoded features. There are many normalization schemes to choose from too. You could use **standard deviation**, **min-max scaling**, or even **median absolute deviation** [9].

In this section, we will explain and use one advanced counterfactual finding method. Then, we will explore Google's **What If Tool** (**WIT**). It has a simple L1- and L2-based counterfactual finder, which is limited to the dataset but makes up for it with other useful interpretation features.

Counterfactual explanations guided by prototypes

The most sophisticated counterfactual finding algorithms do the following:

- **Loss:** These leverage a *loss function* that helps us optimize finding the counterfactuals closest to our *instance of interest*.
- **Perturbation:** These tend to operate with a *perturbation space* much like anchors do, changing as few features as possible. Please note that counterfactuals don't have to be real points in your dataset. That would be far too limiting. Counterfactuals exist in the realm of the possible, not of the necessarily known.
- **Distribution:** However, counterfactuals have to be realistic, and therefore, interpretable. For example, a loss function could help determine that `age < 0` alone is enough to make any medium-/high-risk instance low-risk. This is why counterfactuals should lie close to the statistical distributions of your data, especially *class-specific distributions*. They also should not be biased against smaller-scale features, namely categorical variables.
- **Speed:** These run fast enough to be useful in real-world scenarios.

Alibi's **Counterfactuals Guided by Prototypes** (`CounterFactualProto`) has all these properties. It has a loss function that includes both L1 (*Lasso*) and L2 (*Ridge*) regularization as a linear combination, just like **Naïve Elastic Net** does $\beta(L1 + L2)$ but with a weight (β) only on the L1 term. The clever part of this algorithm is that it can (optionally) use an *autoencoder* to understand the distributions. We will leverage one in *Chapter 7*, *Visualizing Convolutional Neural Networks*. However, what's important to note here is that autoencoders, in general, are neural networks that learn a compressed representation of your training data. This method incorporates loss terms from the autoencoder, such as one for the nearest prototype. A prototype is the dimensionality-reduced representation of the counterfactual class.

If an autoencoder is not available, the algorithm uses a tree often used for multidimensional search (*k-d trees*) instead. With this tree, the algorithm can efficiently capture the class distributions and choose the nearest prototype. Once it has the prototype, the perturbations are guided by it. Incorporating a prototype loss term in the loss function ensures that the resulting perturbations will be close enough to the prototype that is in distribution for the counterfactual class. Many modeling classes and interpretation methods overlook the importance of treating continuous and categorical features differently.

`CounterFactualProto` can use two different distance metrics to compute the pairwise distances between categories of a categorical variable: **Modified Value Difference Metric** (**MVDM**) and **Association-Based Distance Metric** (**ABDM**) and can even combine both. Another way in which `CounterFactualProto` ensures meaningful counterfactuals is by limiting permutated features to predefined ranges. We can use the minimum and maximum values of features to generate a tuple of arrays (`feature_range`):

```python
feature_range = (
    X_train.values.min(axis=0).reshape(1,21).astype(np.float32),\
    X_train.values.max(axis=0).reshape(1,21).astype(np.float32)
)
print(feature_range)
```

The preceding code outputs two arrays – the first one with the minimum and the second with the maximum of all features:

```
(array([[18.,  0.,  ... , 0.,  0.,  0.]], dtype=float32), array([[96., 20., ...
 , 1.,  1.,  1.]], dtype=float32))
```

We can now instantiate an explainer with `CounterFactualProto`. As arguments, it requires the model's `predict` function (`predict_cb_fn`), the shape of the instance you want to explain (`X_test_eval.shape`), the maximum amount of optimization iterations to perform (`max_iterations`), and the feature range for perturbed instances (`feature_range`). Many hyperparameters can be chosen, including the β weight to apply to the L1 loss (`beta`) and the θ weight to apply to the prototype loss (`theta`). Also, you must specify whether to use the *k-d tree* or not (`use_kdtree`) when the autoencoder model isn't provided. Once the explainer is instantiated, you fit it to the test dataset. We are specifying the distance metric for categorical features (`d_type`) as the combination of ABDM and MVDM:

```python
cf_cb_explainer = CounterFactualProto(
    predict_cb_fn,
    c_init=1,
    X_test_eval.shape,
    max_iterations=500,
    feature_range=feature_range,
    beta=.01,
    theta=5,
    use_kdtree=True
)
cf_cb_explainer.fit(X_test.values, d_type='abdm-mvdm')
```

Creating an explanation with an explainer is similar to how it was with anchors. Just pass the instance (`X_test_eval`) to the `explain` function. However, outputting the results is not as straightforward, mainly because of the features converting between one-hot-encoded and ordinal, and interactions among the features. The documentation for Alibi (`https://docs.seldon.io/projects/alibi/`) has a detailed example of how this is done.

We will instead use a utility function called `describe_cf_instance` that does this for us using the *instance of interest* (`X_test_eval`), explanation (`cf_cb_explanation`), class names (`class_names`), one-hot-encoded category locations (`cat_vars_ohe`), category map (`category_map`), and feature names (`feature_names`):

```
cf_cb_explanation = cf_cb_explainer.explain(X_test_eval) mldatasets.describe_
cf_instance(
    X_test_eval,
    cf_cb_explanation,
    class_names,
    cat_vars_ohe,
    category_map,
    feature_names
)
```

The following output was produced by the preceding code:

```
Instance Outcomes and Probabilities
----------------------------------------------
     original:  Medium/High Risk
                [0.46732193 0.53267807]
counterfactual:  Low Risk
                [0.50025815 0.49974185]
Categorical Feature Counterfactual Perturbations
----------------------------------------------
           sex:  Male  -->  Female
          race:  African-American  -->  Asian
 c_charge_degree:  Felony 7th Degree  -->  Felony 1st Degree
Numerical Feature Counterfactual Perturbations
----------------------------------------------
     priors_count:  2.00  -->  1.90
```

You can appreciate from the output that the *instance of interest* ("original") has a 53.26% probability of being *Medium/High Risk*, but the counterfactual is barely on the *Low Risk* side with 50.03%! A counterfactual that is slightly on the other side is what we would like to see because that likely means that it is as close as possible to our *instance of interest*. There are four feature differences between them, three of which are categorical (`sex`, `race`, and `c_charge_degree`). The fourth difference is the `priors_count` numerical feature, which is treated as continuous since the explainer doesn't know it's discrete. In any case, the relationship should be *monotonic*, meaning an increase in one variable is consistent with a decrease or increase in the other. In this case, fewer priors should always mean lower risk, which means we can interpret the 1.90 as a 1 because if 0.1 fewer priors helped reduce the risk, a whole prior should also do so.

A more powerful insight derived from `CounterFactualProto`'s output is that two demographic features were present in the closest counterfactual to this feature. One was found with a method that is designed to follow our classes' statistical distributions and isn't biased against or in favor of specific types of features. And even though it is surprising to see Asian females in our counterfactual because it doesn't fit the narrative that Caucasian males are getting preferential treatment, it is concerning to realize that `race` appears in the counterfactual at all.

The Alibi library has several counterfactual finding methods, including one that leverages reinforcement learning. Alibi also uses *k-d trees* for its trust score, which I highly recommend as well! The trust score measures the agreement between any classifier and a modified nearest neighbors classifier. The reasoning behind this is that a model's predictions should be consistent on a local level to be trustworthy. In other words, if you and your neighbor are almost the same in every way, why would you be treated differently?

Counterfactual instances and much more with WIT

Google's WIT is a very versatile tool. It requires very little input or preparation and opens up in your Jupyter or Colab notebook as an interactive dashboard with three tabs:

- **Datapoint editor:** To visualize your datapoints, edit them, and explain their predictions.
- **Performance:** To see high-level model performance metrics (for all regression and classification models). For binary classification, this tab is called **Performance and Fairness** because, in addition to high-level metrics, predictive fairness can be compared between your dataset's feature-based slices.
- **Features:** To view general feature statistics.

Given that the **Features** tab doesn't relate to model interpretations, we will explore only the first two in this section.

Configuring WIT

Optionally, we can enrich our interpretations in WIT by creating attributions, which are values that explain how much each feature contributes to each prediction. You could use any method to generate attributions, but we will use SHAP. We covered SHAP first in *Chapter 4, Global Model-Agnostic Interpretation Methods*. Since we will interpret our CatBoost model in the WIT dashboard, the SHAP explainer that is most suitable is `TreeExplainer`, but `DeepExplainer` would work for the neural network (and `KernelExplainer` for both). To initialize `TreeExplainer`, we need to pass the fitted model (`fitted_cb_mdl`):

```
shap_cb_explainer = shap.TreeExplainer(fitted_cb_mdl)
```

WIT requires all the features in the dataset (including the labels). We will use the test dataset, so you could concatenate `X_test` and `y_test`, but even those two exclude the ground truth feature (`is_recid`). One way of getting all of them is to subset `recidivism_df` with the test dataset indexes (`y_test.index`). WIT also needs our data and columns in list format so we can save them as variables for later use (`test_np` and `cols_l`).

Lastly, for predictions and attributions, we will need to remove our ground truth (`is_recid`) and classification label (`compas_score`), so let's save the index of these columns (`delcol_idx`):

```
test_df = recidivism_df.loc[y_test.index]
test_np = test_df.values
cols_l = test_df.columns
delcol_idx = [
    cols_l.get_loc("is_recid"),
    cols_l.get_loc("compas_score")
]
```

WIT has several useful functions for customizing the dashboard, such as setting a custom distance metric (`set_custom_distance_fn`), displaying class names instead of numbers (`set_label_vocab`), setting a custom `predict` function (`set_custom_predict_fn`), and a second `predict` function to compare two models (`compare_custom_predict_fn`).

In addition to `set_label_vocab`, we are going to only use a custom `predict` function (`custom_predict_with_shap`). All it needs to function is to take an array with your `examples_np` dataset and produce some predictions (`preds`). However, we first must remove features that we want in the dashboard but weren't used for the training (`delcol_idx`). This function's output is a dictionary with the predictions stored in a `predictions` key. But we'd also like some attributions, which is why we need an `attributions` key in that dictionary. Therefore, we take our SHAP explainer and generate `shap_values`, which is a `numpy` array. However, attributions need to be a list of dictionaries to be understood by the WIT dashboard. To this end, we iterate `shap_output` and convert each observation's SHAP values array into a dictionary (`attrs`) and then append this to a list (`attributions`):

```
def custom_predict_with_shap(examples_np):
    #For shap values, we only need the same features
    #that were used for training
    inputs_np = np.delete(np.array(examples_np),delcol_idx,axis=1)
    #Get the model's class predictions
    preds = predict_cb_fn(inputs_np)
    #With test data, generate SHAP values which converted
    #to a list of dictionaries format
    keepcols_l = [c for i, c in enumerate(cols_l)\
                if i not in delcol_idx]
    shap_output = shap_cb_explainer.shap_values(inputs_np)
    attributions = []
    for shap in shap_output:
        attrs = {}
        for i, col in enumerate(keepcols_l):
            attrs[col] = shap[i]
        attributions.append(attrs)
    #Prediction function must output predictions/attributions
```

```
    #in dictionary
    output = {'predictions': preds, 'attributions': attributions}
    return output
```

Before we build the WIT dashboard, it's important to note that to find our *instance of interest* in the dashboard, we need to know its position within the numpy array provided to WIT because these don't have indexes as pandas DataFrames do. To find the position, all we need to do is provide the get_loc function with the index:

```
print(y_test.index.get_loc(5231))
```

The preceding code outputs as 2910, so we can take note of this number. Building the WIT dashboard is fairly straightforward now. We first initialize a config (WitConfigBuilder) with our test dataset in numpy format (test_np) and our list of features (cols_l). Both are converted to lists with tolist(). Then, we set our custom predict function with set_custom_predict_fn and our target feature (is_recid) and provide our class names. We will use the ground truth this time to evaluate fairness from the perspective of what really happened. Once the config is initialized, the widget (WitWidget) builds the dashboard with it. You can optionally provide a height (the default is 1,000 pixels):

```
wit_config_builder = WitConfigBuilder(
    test_np.tolist(),
    feature_names=cols_l.tolist()
).set_custom_predict_fn(custom_predict_with_shap
).set_target_feature("is_recid").set_label_vocab(class_names)
WitWidget(wit_config_builder, height=800)
```

Datapoint editor

In *Figure 6.5*, you can see the WIT dashboard with its three tabs. We will first explore the first tab (**Datapoint editor**). It has **Visualize** and **Edit** panes on the left, and on the right, it can show you either **Datapoints** or **Partial dependence plots**. When you have **Datapoints** selected, you can visualize the datapoints in many ways using the controls in the upper right (the highlighted area *A*). What we have done in *Figure 6.5* is set the following:

- **Binning | X-axis:** c_charge_degree_(F7)
- **Binning | Y-axis:** compas_score
- **Color By:** race_African-American

Everything else stays the same.

These settings resulted in all our datapoints being neatly organized into two rows and two columns and color-coded by African American or not. The right column is for those with a level 7 charge degree, and the upper row is for those with a *Medium/High Risk* COMPAS score. We can look for datapoint 2910 in this subgroup (*B*) by clicking on the top-rightmost item. It should appear in the **Edit** pane (*C*). Interestingly enough, the SHAP attributions for this datapoint are three times higher for `age` than they are for `race_African-American`. But still, race altogether is second to age in importance. Also, notice that in the **Infer** pane, you see the predicted probability for *Medium/High Risk* is approximately 83%:
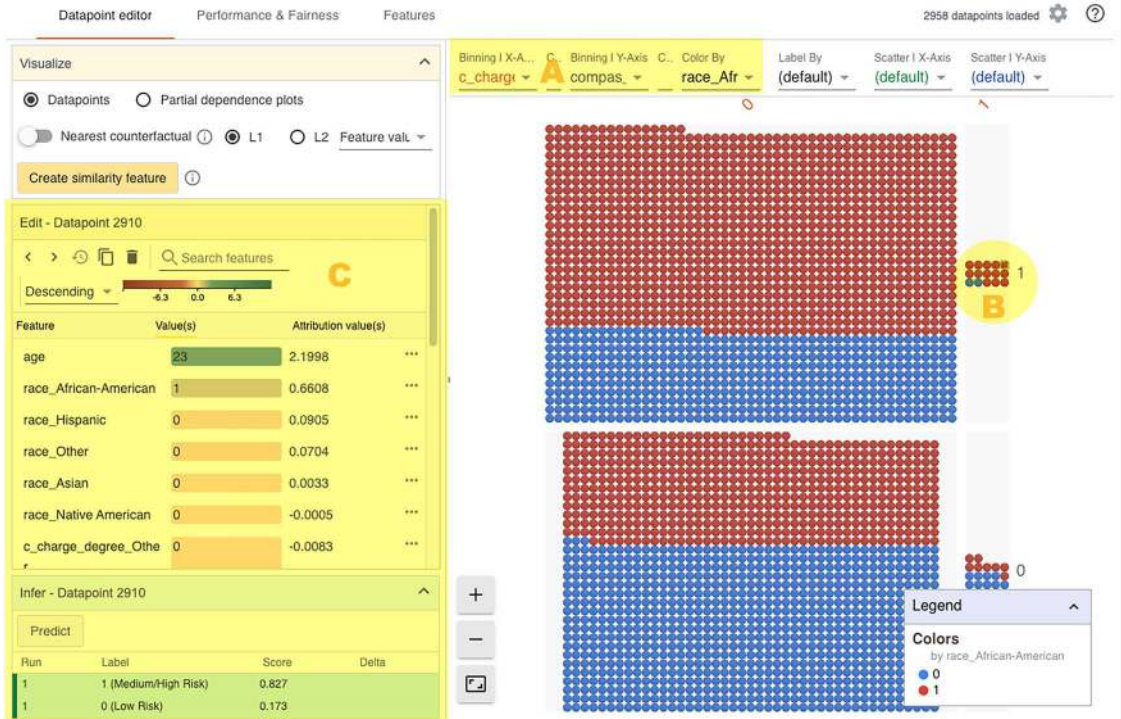

Figure 6.5: WIT dashboard with our instance of interest

WIT can find the nearest counterfactual using L1 or L2 distances. And it can use either feature values or attributions to calculate the distances. As mentioned earlier, WIT can also include a custom distance-finding function if you add it to the configuration. For now, we will select **L2** with the **Feature value**. In *Figure 6.6*, these options appear in the highlighted *A* area. Once you choose a distance metric and enable **Nearest counterfactual**, it appears side by side with our *instance of interest* (area *B*), and it compares their predictions as shown in *Figure 6.6* (area *C*). You can sort the features by **Absolute attribution** for a clearer understanding of feature importance on a local level. The counterfactual is only 3 years older but has zero priors instead of two, yet that was enough to reduce the **Medium/High Risk** to nearly 5%:
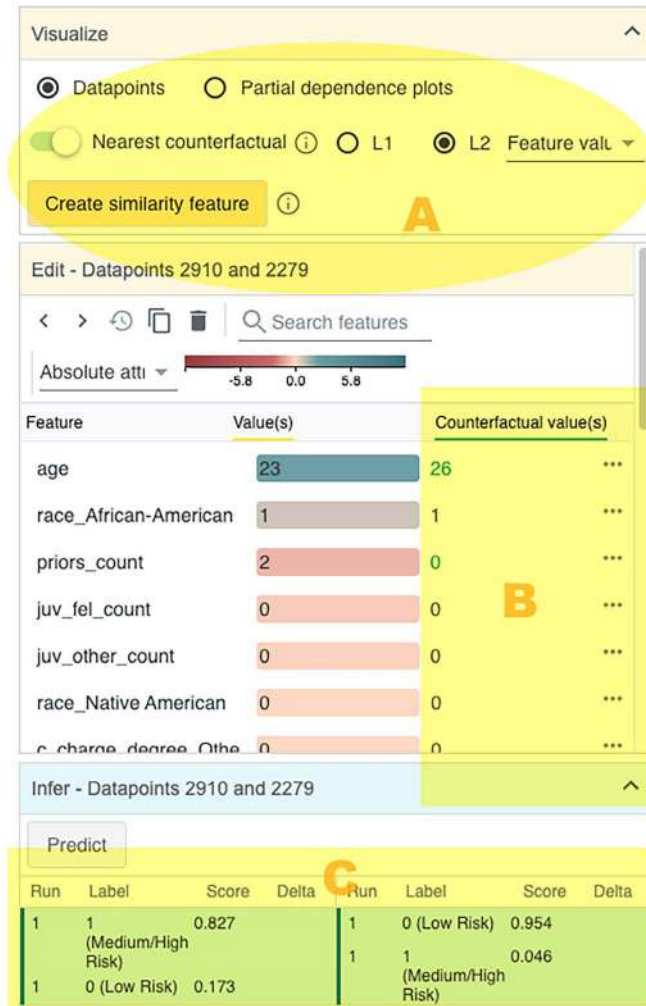

Figure 6.6: How to find the nearest counterfactual in WIT

While both our *instance of interest* and counterfactual remain selected, we can visualize them along with all other points. By doing this, you take insights from local interpretations and can create enough context for global understandings. For instance, let's change our visualization settings to the following:

- • **Binning** | **X-axis**: `Inference label`
- • **Binning** | **Y-axis**: `(none)`
- • **Scatter** | **X-axis**: `age`
- • **Scatter** | **Y-axis**: `priors_count`

Everything else stays the same.

The result of this visualization is depicted in *Figure 6.7*. You can tell that the **Low Risk** bins' points tend to hover in the lower end of `priors_count`. Both bins show that `prior_count` and `age` have a slight correlation, although this is substantially more pronounced in the **Medium/High Risk** bin. However, what is most interesting is the sheer density of African American datapoints deemed **Medium/High Risk** in age ranging 18–25 and with `prior_count` below three, compared to those in the **Low Risk** bin. It suggests that both lower `age` and higher `priors_count` increases risk more for African Americans than others:
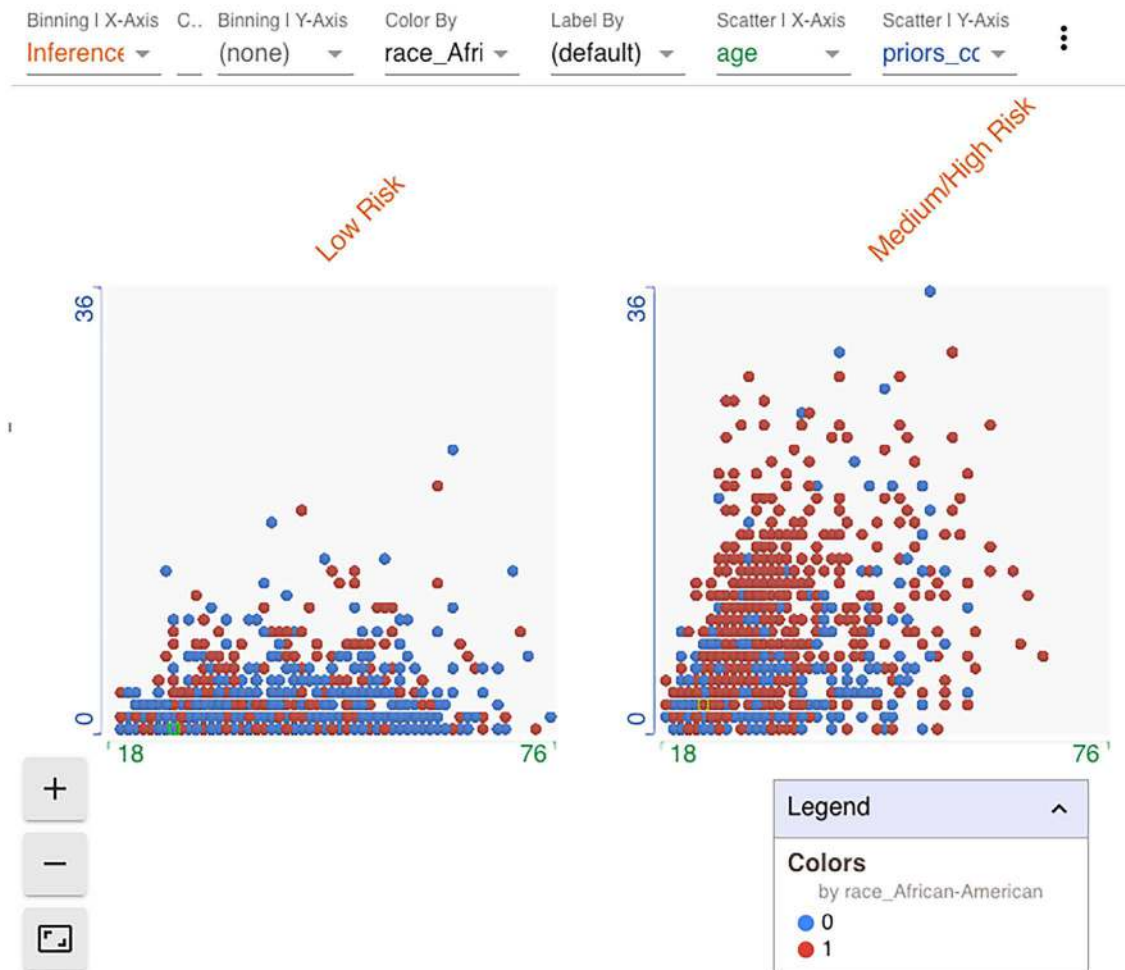
Figure 6.7: Visualizing age versus priors_count in WIT

We can try creating our own counterfactuals by editing the datapoint. What happens when we reduce `priors_count` to one? The answer to this question is depicted in *Figure 6.8*. Once you make the change and click on the **Predict** button in the **Infer** pane, it adds an entry to the last prediction history in the **Infer** pane. You can tell in **Run #2** that the risk reduces nearly to 33.5%, down nearly 50%!
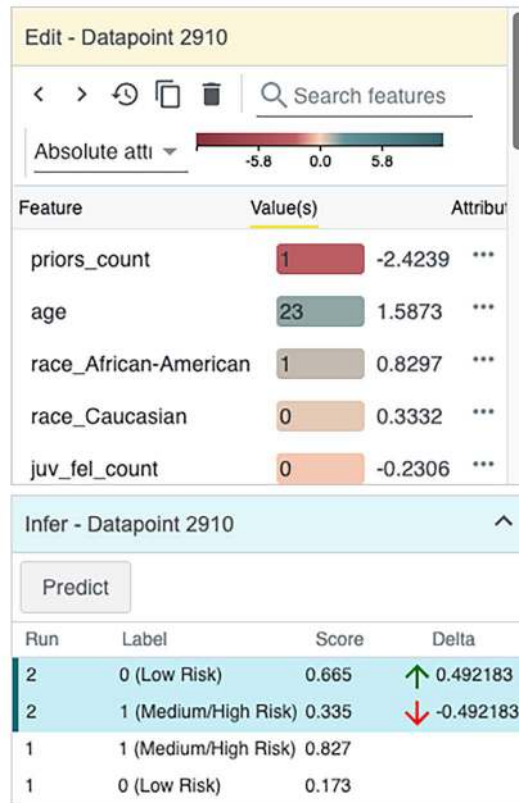
Figure 6.8: Editing the datapoint to decrease priors_count in WIT

Now, what happens if `age` is only 2 years older but there are two priors? In *Figure 6.9*, **Run #3** tells you that it barely made it inside the **Low Risk** score:
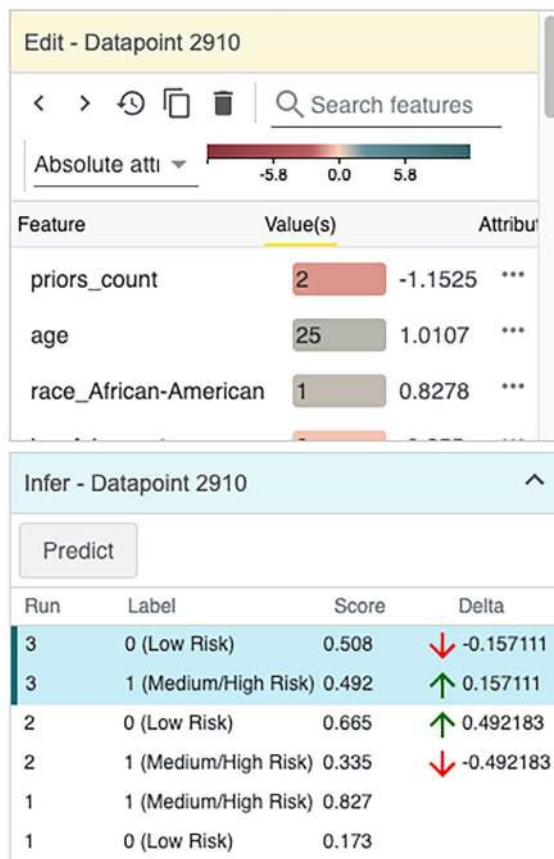
Figure 6.9: Editing the datapoint to increase the age in WIT

Another feature that the **Datapoint editor** tab has is **Partial dependence plots,** which we covered in *Chapter 4, Global Model-Agnostic Interpretation Methods*. If you click on this radio button, it will modify the right pane to look like *Figure 6.10*. By default, if a datapoint is selected, the PDPs are local, meaning they pertain to the chosen datapoint. But you can switch to global. In any case, it's best to sort plots by variation as done in *Figure 6.10*, where `age` and `priors_count` have the highest variation. Interestingly, neither of them is monotonic, which doesn't make sense. The model should learn that an increase in `priors_count` should consistently increase risk. It should be the same with a decrease in `age`. After all, academic research shows that crime tends to peak in the mid-20s and that higher priors increase the likelihood of recidivism. The relationship between these two variables is also well understood, so perhaps some data engineering and monotonic constraints could make sure a model was consistent with known phenomena rather than learning the inconsistencies in the data that lead to unfairness. We will cover this in *Chapter 12, Monotonic Constraints and Model Tuning for Interpretability*:
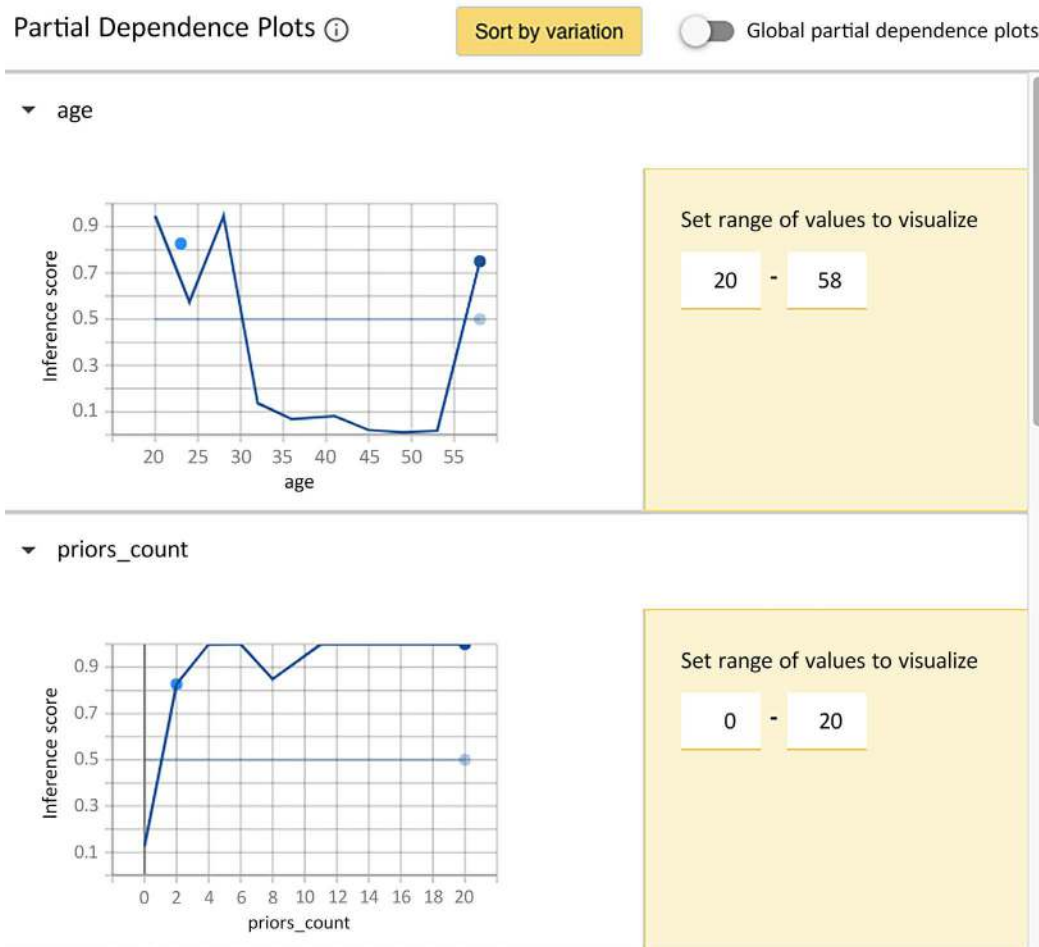
Figure 6.10: Local partial dependence plot for age and priors_count

Is there something that can be done to improve fairness in a model that has already been trained? Indeed, there is. The **Performance & Fairness** tab can help with that.

Performance & Fairness

When you click on the **Performance & Fairness** tab, you will see that it has **Configure** and **Fairness** panes on the left. And on the right, you can explore the overall performance of the model (see *Figure 6.11*). In the upper part of this pane, it has **False Positives (%)**, **False Negatives (%)**, **Accuracy (%)**, and **F1** fields. If you expand the pane, it shows the ROC curve, PR curve, confusion matrix, and mean attributions – the average Shapley values. We covered these terms in the previous chapters of this book either directly or indirectly, except for the PR curve. The **Precision-Recall** (**PR**) is very much like the ROC curve, except it plots precision against recall instead of TPR versus FPR. In this plot, precision is expected to decrease as recall decreases. Unlike ROC, it's considered worse than a coin toss when the line is close to the *x*-axis, and it's best suited to imbalanced classification problems:

Figure 6.11: Performance & Fairness tab initial view

A classification model will output probabilities that an observation belongs to a class label. We usually take every observation above or equal to 0.5 to belong to the positive class. Otherwise, we predict it to belong to the negative class. This threshold is called the **classification threshold**, and you don't always have to use the standard 0.5.

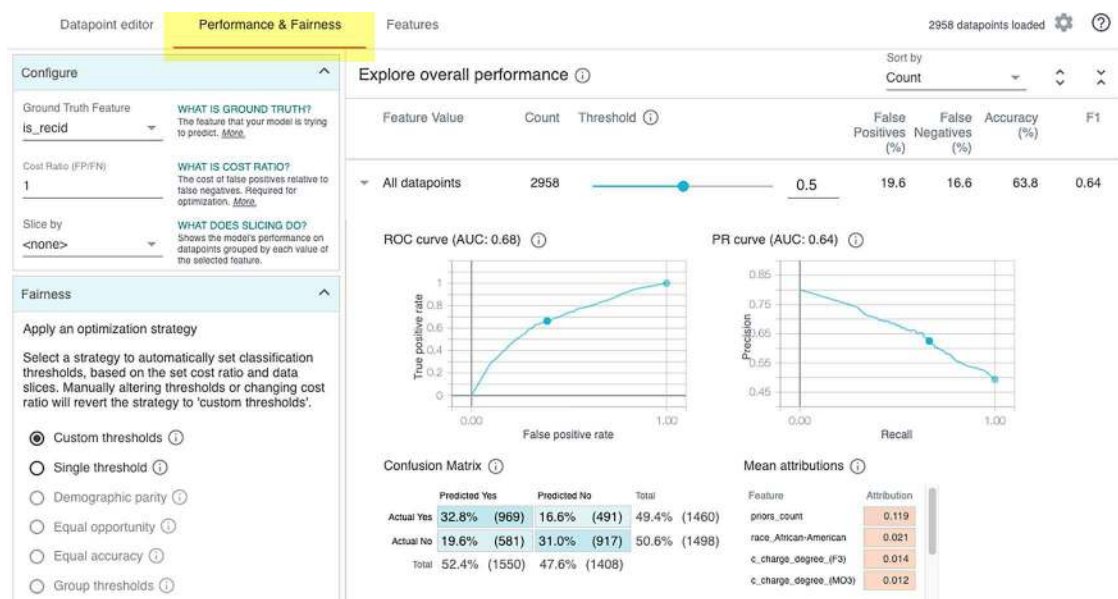There are many cases in which it is appropriate to perform **threshold tuning.** One of the most compelling reasons is imbalanced classification problems because often models optimize performance on accuracy alone but end up with bad recall or precision. Adjusting the threshold will improve the metric you care most about:
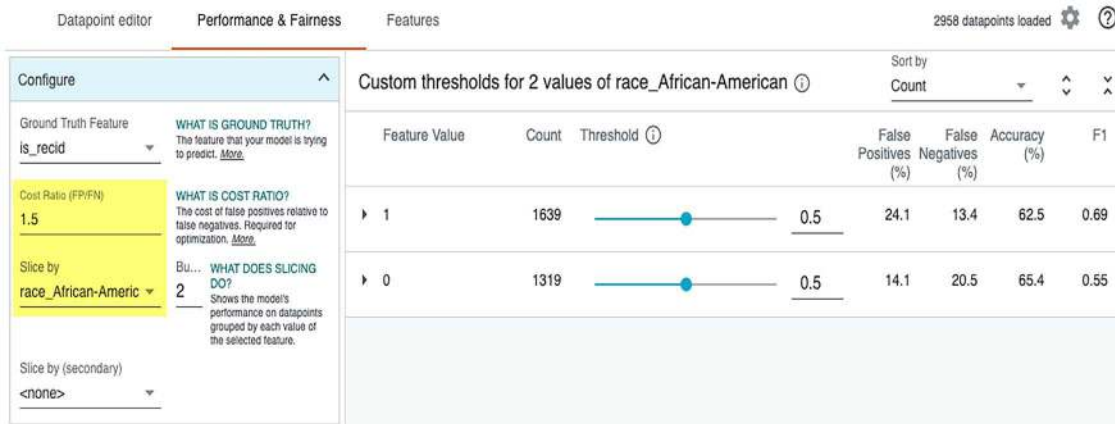
Figure 6.12: Slicing performance metrics by race_African-American

Another primary reason to adjust thresholds is for fairness. To this end, you need to examine the metric you most care about across different slices of your data. In our case, **False Positives (%)** is where we can appreciate unfairness the most. For instance, take a look at *Figure 6.12*. In the **Configure** pane, we can slice the data by race_African-American, and to the right of it, we can see what we observed at the beginning of this chapter, which is that FPs for African Americans are substantially higher than for other segments. One way to fix this is through an automatic optimization method such as **Demographic parity** or **Equal opportunity**. If you are to use one of these, it's best to adjust **Cost Ratio (FP/FN)** to tell the optimizer that FPs are worth more than FNs:
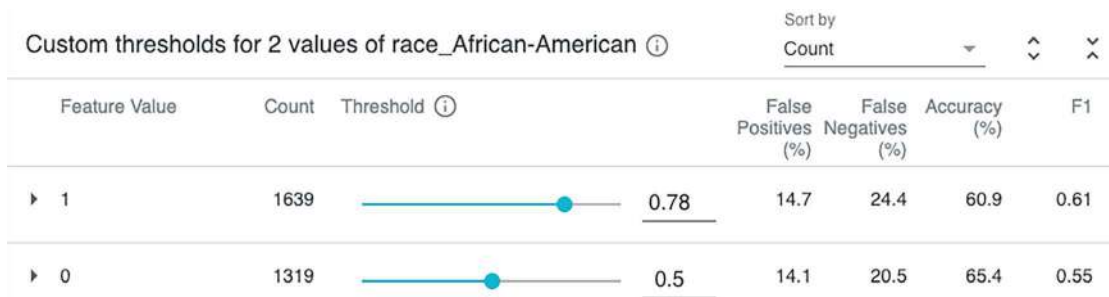
Figure 6.13: Adjusting the classification threshold for the dataset sliced by race_African-American

We can also adjust thresholds manually using the default **Custom thresholds** setting (see *Figure 6.13*). For these slices, if we want approximate parity with our FPs, we should use 0.78 as our threshold for when race_African-American=1. The drawback is that FNs will increase for this group, not achieving parity on that end. A cost ratio would help determine whether 14.7% in FPs justifies 24.4% in FNs, but to do this, we would have to understand the average costs involved. We will examine odds calibration methods further in *Chapter 11, Bias Mitigation and Causal Inference Methods*.

Mission accomplished

This chapter's mission was to see whether there was unfair bias in predicting whether a particular defendant would recidivate. We demonstrated that the FPR for African American defendants is 1.87 times higher than for Caucasian defendants. This disparity was confirmed with WIT, indicating that the model in question is much more likely to misclassify the positive class based on race. However, this is a global interpretation method, so it doesn't answer our question regarding a specific defendant. Incidentally, in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*, we will cover other global interpretation methods for unfairness.

To ascertain whether the model was racially biased against the defendant in question, we leveraged anchor and counterfactual explanations – they both output race as a primary feature in their explanations. The anchor did it with relatively high precision and coverage, and *Counterfactuals Guided by Prototypes* found that the closest decision has a different race. That being said, in both cases, race wasn't the only feature in the explanations. The features usually included any or all of the following: `priors_count`, `age`, `charge_degree`, and `sex`. The inconsistent rules involving the first three regarding `race` suggests double standards and the involvement of `sex` suggests intersectionality. **Double standards** are when rules are applied unfairly to different groups. **Intersectionality** is how overlapping identities create different systems of interconnected modes of discrimination. However, we know that females of all races are less likely to recidivate according to academic research. Still, we have to ask ourselves whether females have a structural advantage that makes them privileged in this context. A healthy dose of skepticism can help, since when it comes to bias, there's usually a more elaborate dynamic going on than meets the eye. The bottom line is that despite all the other factors that interplay with race, and provided that there's no relevant criminological information that we are missing, yes—there's racial bias involved in this particular prediction.

Summary

After reading this chapter, you should know how to leverage anchors, to understand the decision rules that impact a classification, and counterfactuals, to grasp what needs to change for the predicted class to change. You also learned how to assess fairness using confusion matrices and Google's WIT. In the next chapter, we will study interpretation methods for **Convolutional Neural Networks** (**CNNs**).

Dataset sources

- ProPublica Data Store, 2019, COMPAS Recidivism Risk Score Data and Analysis. Originally retrieved from `https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis`

Further reading

- Desmarais, S.L., Johnson, K.L., and Singh, J.P., 2016, *Performance of recidivism risk assessment instruments in U.S. correctional settings*. Psychol Serv;13(3):206-222: `https://doi.org/10.1037/ser0000075`
- Berk, R., Heidari, H., Jabbari, S., Kearns, M., and Roth, A., 2017, *Fairness in Criminal Justice Risk Assessments: The State of the Art*. Sociological Methods & Research.

• Angwin, J., Larson, J., Mattu, S., and Kirchner, L., 2016, *Machine Bias. There's software used across the county to predict future criminals*: `https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing`

• Ribeiro, M.T., Singh, S., and Guestrin, C., 2018, *Anchors: High-Precision Model-Agnostic Explanations*. Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society: `https://doi.org/10.1609/aaai.v32i1.11491`

• Rocque, M., Posick, C., & Hoyle, J., 2015, *Age and Crime*. The encyclopedia of crime and punishment, 1–8: `https://doi.org/10.1002/9781118519639.wbecpx275`

• Dhurandhar, A., Chen, P., Luss, R., Tu, C., Ting, P., Shanmugam, K., and Das, P., 2018, *Explanations based on the Missing: Towards Contrastive Explanations with Pertinent Negatives*. NeurIPS: `https://arxiv.org/abs/1802.07623`

• Jiang, H., Kim, B., and Gupta, M.R., 2018, *To Trust Or Not To Trust A Classifier*. NeurIPS: `https://arxiv.org/pdf/1805.11783.pdf`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

7

Visualizing Convolutional Neural Networks

Up to this point, we have only dealt with tabular data and, briefly, text data, in *Chapter 5*, *Local Model-Agnostic Interpretation Methods*. This chapter will exclusively explore interpretation methods that work with images and, in particular, with the **Convolutional Neural Network** (**CNN**) models that train image classifiers. Typically, deep learning models are regarded as the epitome of black box models. However, one of the benefits of a CNN is how easily it lends itself to visualization, so we can not only visualize outcomes but also every step of the learning process with **activations**. The possibility of interpreting these steps is rare among so-called black box models. Once we have grasped how CNNs learn, we will study how to use state-of-the-art gradient-based attribution methods, such as *saliency maps* and *Grad-CAM* to debug class attribution. Lastly, we will extend our attribution debugging know-how with perturbation-based attribution methods such as *occlusion sensitivity* and `KernelSHAP`.

These are the main topics we are going to cover:

- Assessing the CNN classifier with traditional interpretation methods
- Visualizing the learning process with an activation-based method
- Evaluating misclassifications with gradient-based attribution methods
- Understanding classifications with perturbation-based attribution methods

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `tqdm`, `torch`, `torchvision`, `pytorch-lightning`, `efficientnet-pytorch`, `torchinfo`, `matplotlib`, `seaborn`, and `captum` libraries. Instructions on how to install all of these libraries are in the *Preface*.

> The code for this chapter is located here: `https://packt.link/qzUvD`.

The mission

Over two billion tons of waste is produced annually globally, and it's expected to grow to over 3.5 billion tons by 2050. The alarming rise in global waste production and the need for effective waste management systems have become increasingly critical in recent years. Over half of all household trash in high-income countries is recyclable, with 20% in lower-income countries and rising. Currently, most waste ends up in landfills or incinerated, contributing to environmental pollution and climate change. This is avoidable, considering that, globally, a significant portion of all recyclable materials is not recycled.

Assuming recyclable waste is collected, it can still be hard and costly to sort it. Previously, waste classification technologies included:

- Separating materials by size with rotating cylindrical screens with holes ("trommel screens")
- Separating ferrous and non-ferrous metals with magnetic forces and magnetic fields ("eddy current separators")
- Separating by weight with air
- Separating by density with water ("sink-float separation")
- Manual sorting performed by humans

Implementing all of these techniques effectively can be challenging, even for a large, wealthy, urban municipality. To tackle this challenge, **smart recycling systems** have emerged, leveraging computer vision and AI to classify waste efficiently and accurately.

The development of smart recycling systems can be traced back to the early 2010s when researchers and innovators started exploring the potential of computer vision and AI to improve waste management processes. They first developed basic image recognition algorithms, utilizing features such as color, shape, and texture to identify waste materials. These systems were primarily used in research settings with limited commercial applications. As machine learning and AI became more advanced, smart recycling systems underwent significant improvements. CNNs and other deep learning techniques enabled these systems to learn from vast amounts of data and improve their waste classification accuracy. Additionally, the integration of AI-driven robotics allowed for automated sorting and handling of waste materials, increasing efficiency in recycling plants.

Costs are significantly lower than a decade ago for cameras, robots, and even chips that run deep learning models in low-latency, high-volume scenarios, making state-of-the-art smart recycling systems accessible to even smaller and poorer municipal waste management departments. One of these municipalities in Brazil is looking to revamp their 20-year-old recycling plant made up of a patchwork of machines with a collective sorting accuracy of only 70%. Human sorting can only partially compensate for the difference, leading to inevitable pollution and contamination issues. The Brazilian municipality want to replace the current system with a single conveyor belt that sorts waste efficiently from 12 different categories into bins with a series of robots.

They purchased the conveyor belt, industrial robots, and cameras. Then, they paid an AI consultancy company to develop a model to classify the recyclables. Still, they wanted models of different sizes because they weren't sure how quickly these would run on the hardware they had.

As requested, the consultancy returned with models of various sizes between 4 and 64 million parameters. The largest model (b7) is over six times slower than the smallest one (b0). Still, the largest model has a significantly higher validation F1 score at 96% (F1 val), as opposed to approximately 90% for the smallest one:

Model	b0	b1	b2	b3	b4	b5	b6	b7
# Parameters (millions)	4	6½	8	11	18	28	41	64
Size (Mb)	49	79	93	129	212	342	491	768
Inference Speed	1x	1.3x	1.4x	1.8x	2.4x	3.4x	4.3x	6.2x
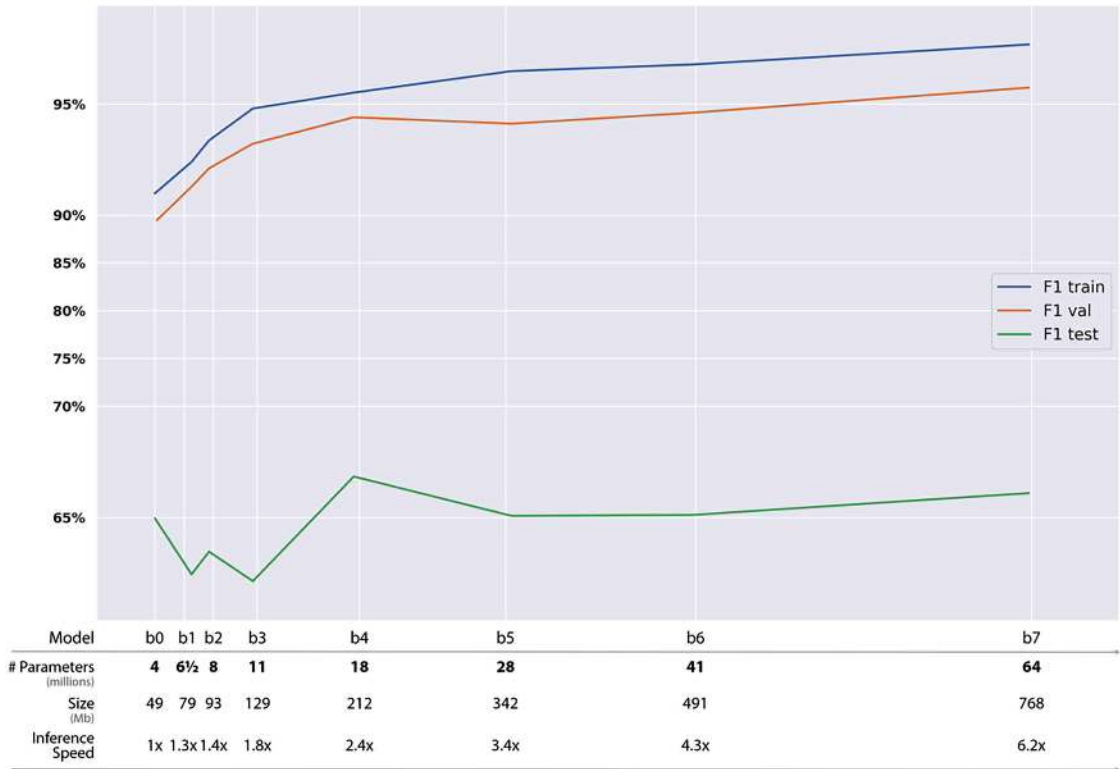
Figure 7.1: F1 scores for models delivered by the AI consultancy company

The municipal leadership was delighted with the results but also surprised because the consultants asked for no domain knowledge or data to train the models, which made them very skeptical. They asked their recycling plant workers to test the models with a batch of recyclables. They got a 25% misclassification rate with that one batch.

To seek a second opinion and an honest evaluation of the model, the municipality has approached another AI consultancy firm – yours!

The first order of business was to assemble a test dataset that was more realistic of the edge cases that the recycling plant workers found among the misclassifications. Your colleague obtained F1 scores with the test dataset between 62% and 66% (F1 test). Next, they have asked you to understand what's causing those misclassifications.

The approach

No single interpretation method is perfect, and even the best scenario can only tell you one part of the story. Therefore, you have decided to, first, assess the model's predictive performance using traditional interpretation methods, including the following:

- ROC curves and ROC-AUC
- Confusion matrices and some metrics derived from them, such as accuracy, precision, recall, and F1

Then, you'll examine the model using an activation-based method:

- Intermediate activation

This is followed by evaluating decisions with three gradient-based methods:

- Saliency maps
- Grad-CAM
- Integrated gradients

And a backpropagation-based method:

- DeepLIFT

This is followed by three perturbation-based methods:

- Occlusion sensitivity
- Feature ablation
- Shapley value sampling

I hope that you understand why the model is not performing as it should and how to fix it by the end of this process. You can also leverage the many plots and visualizations you will produce to communicate this story to the municipality's executives.

Preparations

You will find most of the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/07/GarbageClassifier.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `torchvision` to load the dataset
- `mldatasets`, `pandas`, `numpy`, and `sklearn` (scikit-learn) to manipulate the dataset
- `torch`, `pytorch-lightning`, `efficientnet-pytorch`, and `torchinfo` to predict with the models and show info about the models
- `matplotlib`, `seaborn`, `cv2`, `tqdm`, and `captum` to make and visualize the interpretations

You should load all of them first:

```python
import math
import os, gc
import random
import mldatasets
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

import torchvision
import torch
import pytorch_lightning as pl
import efficientnet_pytorch
from torchinfo import summary

import matplotlib.pyplot as plt
from matplotlib.cm import ScalarMappable
from matplotlib.colors import LinearSegmentedColormap
import seaborn as sns
import cv2
from tqdm.notebook import tqdm
from captum import attr
```

Next, we will load and prepare the data.

Understanding and preparing the data

The data used to train the model is publicly available at Kaggle (`https://www.kaggle.com/datasets/mostafaabla/garbage-classification`). It's called "Garbage Classification" and is a compilation of several different online sources, including web scraping. It has already been split into training and test datasets and also comes with an additional smaller test dataset taken from Wikimedia Commons that your colleague used to test the models. These test images come in a slightly higher resolution too.

We download the data from a ZIP file like this:

```python
dataset_file = "garbage_dataset_sample"
dataset_url = f"https://github.com/PacktPublishing/Interpretable-Machine-
Learning-with-Python-2E/raw/main/datasets/{dataset_file}.zip"

torchvision.datasets.utils.download_url(dataset_url, ".")
torchvision.datasets.utils.extract_archive(f"{dataset_file}.zip",\
                                           remove_finished=True)
```

It will also extract the ZIP file into four folders corresponding to the three datasets and the larger resolution test dataset. Please note that `garbage_dataset_sample` has only a fraction of the training and validation datasets. If you want to download the full datasets, then use `dataset_file = "garbage_dataset"`. It won't impact the size of the test dataset either way. Next, we can initialize the transformation and loading of the datasets like this:

```python
X_train, norm_mean = (0.485, 0.456, 0.406)
norm_std  = (0.229, 0.224, 0.225)
transform = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(norm_mean, norm_std),
    ]
)


train_data = torchvision.datasets.ImageFolder(
    f"{dataset_file}/train", transform
)
val_data = torchvision.datasets.ImageFolder(
    f"{dataset_file}/validation", transform
)
test_data = torchvision.datasets.ImageFolder(
    f"{dataset_file}/test", transform
)
test_400_data = torchvision.datasets.ImageFolder(
    f"{dataset_file}/test_400", transform
)
```

What the above code does is compose a series of standard transforms such as normalization and converting images to tensors. Then, it instantiates PyTorch datasets corresponding to each folder – that is, one for the training, validation, and test datasets, as well as the larger resolution test dataset (`test_400_data`). These datasets also include transforms. That way, each time an image is loaded from one of the datasets, it is automatically transformed. We can verify that the shapes of the datasets match our expectations with the following code:

```python
print(f"# Training Samples:    \t{len(train_data)}")
print(f"# Validation Samples:  \t{len(val_data)}")
print(f"# Test Samples:        \t{len(test_data)}")
print(f"Sample Dimension:      \t{test_data[0][0].shape}")
print("="*50)
print(f"# Test 400 Samples:    \t{len(test_400_data)}")
print(f"# 400 Sample Dimension:\t{test_400_data[0][0].shape}")
```

The preceding code outputs the number of images in each dataset and the dimensions of the images in the datasets. You can tell that there are over 3,700 training images, 900 validation images, and 120 test images of 3 x 224 x 224 dimensions. The first number corresponds to the channels (red, green, and blue) and the following two to the width and height in pixels, which is what the model uses for inference. The Test 400 dataset is the same as the Test dataset, the except images have a larger height and width. We won't need the Test 400 dataset for inference, so it's Okay that it doesn't meet the model's dimension requirements:

```
# Training Samples:          3724
# Validation Samples:        931
# Test Samples:              120
Sample Dimension:            torch.Size([3, 224, 224])
=================================================
# Test 400 Samples:          120
# 400 Sample Dimension:      torch.Size([3, 400, 400])
```

Data preparation

If you print(`test_data[0]`), you'll notice that it will first output a tensor with the image and then a single integer, which we call a scalar. This integer is a number between 0 and 11, which corresponds to the labels used. For quick reference, these are the 12 labels:

```
labels_l = ['battery', 'biological', 'brown-glass', 'cardboard',\
            'clothes', 'green-glass', 'metal', 'paper', 'plastic',\
            'shoes', 'trash', 'white-glass']
```

Interpreting often involves taking single samples and extracting them from the dataset to later perform inference with the model. To that end, it's important to get familiar with extracting any image from the dataset, say the very first sample from the test dataset:

```
tensor, label = test_400_data[0]
img = mldatasets.tensor_to_img(tensor, norm_std, norm_mean)
plt.figure(figsize=(5,5))
plt.title(labels_l[label], fontsize=16)
plt.imshow(img)
plt.show()
```

In the preceding snippet, we are taking the first sample (`0`) from the higher resolution version of the test dataset (`test_400_data`) and extracting the `tensor` and `label` portion from it. Then, we are using the convenience function `tensor_to_img` to convert the PyTorch tensor to a `numpy` array but also reversing the standardization that had been previously performed on the tensor. Then, we plot the image with `matplotlib`'s `imshow` and use the `labels_l` list to convert the label into a string, which we print in the title. The result can be seen in *Figure 7.2*:
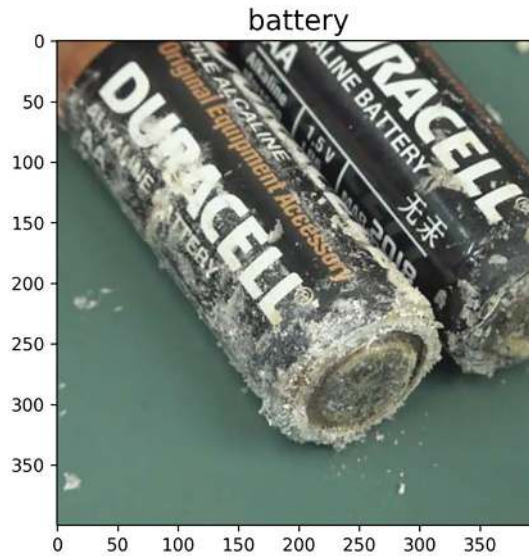

Figure 7.2: A test sample for a recyclable alkaline battery

Another preprocessing step we will need to perform is the **One-Hot Encoding** (**OHE**) of the `y` labels because we will need the OHE form to evaluate the model's predictive performance. Once we initialize the `OneHotEncoder`, we will need to `fit` it to the test labels (`y_test`) in array format. But first, we will need to put the test labels into a list (`y_test`). We can do the same with the validation labels because these will also be useful for easy evaluation:

```
y_test = np.array([l for _, l in test_data])
y_val = np.array([l for _, l in val_data])

ohe = OneHotEncoder(sparse=False).\
            fit(np.array(y_test).reshape(-1, 1))
```

Also, for the sake of reproducibility, always initialize your random seeds like this:

```
rand = 42
os.environ['PYTHONHASHSEED']=str(rand)
np.random.seed(rand)
random.seed(rand)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
if device == 'cuda':
    torch.cuda.manual_seed(rand)
else:
    torch.manual_seed(rand)
```

It is acknowledged that determinism is very difficult with deep learning and is often session-, platform-, and architecture-dependent. If you are using an **NVIDIA GPU**, you can attempt to use PyTorch to avoid nondeterministic algorithms with the command `torch.use_deterministic_algorithms(True)`. It's not a guarantee, but it will throw an error when the operation that you are attempting can't be accomplished deterministically. If it succeeds, it will be much slower. It's only worth it if you need to make model outcomes identical – for instance, for scientific research or regulatory compliance. For further details about reproducibility and PyTorch, look here: `https://pytorch.org/docs/stable/notes/randomness.html`.

Inspect data

Now, let's take a peek at what images are in our datasets. We know that the training and validation datasets are very similar, so we will start with the validation dataset. We can iterate every class in `labels_l` and randomly select a single one from the validation dataset with `np.random.choice`. We place each image on a 4 × 3 grid with the class label above it:

```
plt.subplots(figsize=(14,10))
for c, category in enumerate(labels_l):
    plt.subplot(3, 4, c+1)
    plt.title(labels_l[c], fontsize=12)
    idx = np.random.choice(np.where(y_test==c)[0], 1)[0]
    im = mldatasets.tensor_to_img(test_data[idx][0], norm_std,\
                                  norm_mean)
    plt.imshow(im, interpolation='spline16')
    plt.axis("off")
plt.show()
```

The preceding code generates *Figure 7.3*. You can tell that there is significant pixelation around the edges of the items; some items appear much darker than others, and some of the pictures are from odd angles:

Figure 7.3: A random sample of the validation dataset

Let's now do the same for the test dataset to compare it to the validation/training datasets. We can use the same code as before, except we replace `y_val` with `y_test`, and `val_data` with `test_data`. The resulting code generates *Figure 7.4*. You can tell that the test set has less pixelated and more consistently lit items, mostly from the top- and side-facing angles:
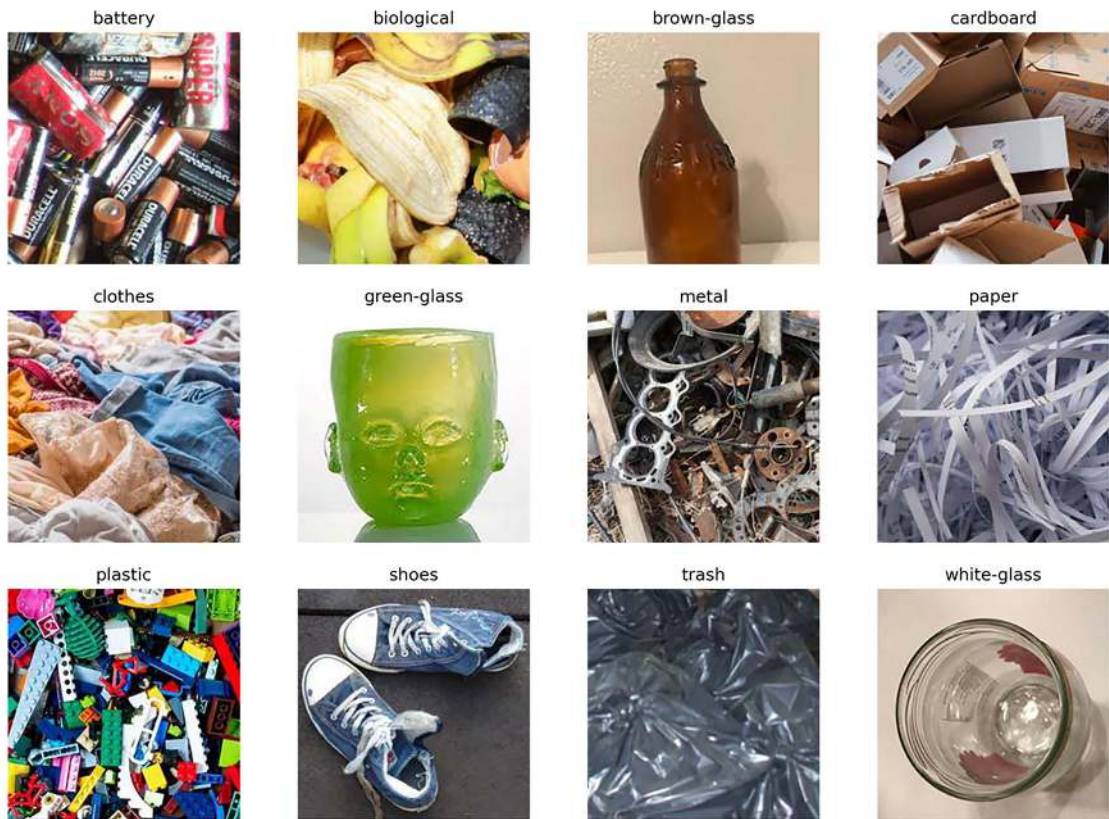
Figure 7.4: A random sample of the test dataset

We won't need to train a CNN in this chapter. Thankfully, it has been provided to us by the client.

The CNN models

The models trained by the other consultancy company are fine-tuned EfficientNet models. In other words, the AI consultancy company took a previously trained model with the EfficientNet architecture and trained it further with the garbage classification dataset. This technique is called **transfer learning** because it allows a model to utilize previously learned knowledge from a large dataset (in this case, a million images from the ImageNet database) and apply it to new tasks with smaller datasets. The advantage is it significantly reduces training time and computational resources while maintaining high performance because it has already learned to extract useful features from images, which can be a valuable starting point for a new task and only needs to adapt to the specific task at hand.

It makes sense that they chose EfficientNet. After all, EfficientNet is a family of CNNs introduced by Google AI researchers in 2019. The key innovation of EfficientNet is its compound scaling method, which enables the model to achieve higher accuracy and efficiency than other CNNs. In addition, it is based on the observation that different dimensions of the model, such as width, depth, and resolution, contribute to the overall performance in a balanced way. The EfficientNet architecture is built upon a baseline model called EfficientNet-B0. A compound scaling method is employed to create larger and more powerful versions of the baseline model, which simultaneously scales up the width, depth, and resolution of the network. This results in a series of models, EfficientNet-B1 to EfficientNet-B7, with increasing capacity and performance. The largest model, EfficientNet-B7, has achieved state-of-the-art performance on several benchmarks, such as ImageNet.

Load the CNN model

Before we can load the model, we must define the class for EfficientLite – a class that inherits from PyTorch Lightning's `pl.LightningModule`. This class is designed to create a custom model based on the EfficientNet architecture, train it, and perform inference. We only need it for the latter, which is why we have also adapted it to include a `predict()` function – much like scikit-learn models do for the convenience of being able to use similar evaluation functions to would with these models:

```python
class EfficientLite(pl.LightningModule):
    def __init__(self, lr: float, num_class: int,\
                 pretrained="efficientnet-b0", *args, **kwargs):
        super().__init__()

        self.save_hyperparameters()
        self.model = efficientnet_pytorch.EfficientNet.\
                                    from_pretrained(pretrained)
        in_features = self.model._fc.in_features
        self.model._fc = torch.nn.Linear(in_features, num_class)

    def forward(self, x):
        return self.model(x)

    def predict(self, dataset):
        self.model.eval()
        device = torch.device("cuda" if torch.cuda.is_available()\
                              else "cpu")
        with torch.no_grad():
            if isinstance(dataset, np.ndarray):
                if len(dataset.shape) == 3:
                    dataset = np.expand_dims(dataset, axis=0)
                dataset = [(x,0) for x in dataset]
            loader = torch.utils.data.DataLoader(dataset,\
```

```
                                          batch_size=32)
        probs = None
    for X_batch, _ in tqdm(loader):
        X_batch = X_batch.to(device, dtype=torch.float32)
        logits_batch =  self.model(X_batch)
        probs_batch = torch.nn.functional.softmax(logits_batch,\
                                    dim=1).cpu().detach().numpy()
        if probs is not None:
            probs = np.concatenate((probs, probs_batch))
        else:
            probs = probs_batch
        clear_gpu_cache()
    return probs
```

You will notice that the class has three functions:

- `__init__`: This is the constructor for the `EfficientLite` class. It initializes the model by loading a pretrained EfficientNet model using the `efficientnet_pytorch.EfficientNet.from_pretrained()` method. It then replaces the last fully connected layer (`_fc`) with a new `torch.nn.Linear` layer that has the same number of input features but a different number of output features equal to the number of classes (`num_class`).
- `forward`: This method defines the forward pass of the model. It takes an input tensor `x` and passes it through the model, returning the output.
- `predict`: This method takes a dataset and performs inference using the trained model. It first sets the model to evaluation mode (`self.model.eval()`). The input dataset is converted into a DataLoader object with a batch size of 32. The method iterates over the DataLoader, processing each batch of data and computing probabilities using the softmax function. The `clear_gpu_cache()` function is called after each iteration to release unused GPU memory. Finally, the method returns the computed probabilities as a `numpy` array.

If you are using a CUDA-enabled GPU, there's a utility function called `clear_gpu_cache()`, which is run every time there's a GPU-intensive operation. Depending on how powerful your GPU is, you may need to run it more often. Feel free to use another convenience function, `print_gpu_mem_used()`, to check how much GPU memory is utilized at any given moment or to print the entire summary with `print(torch.cuda.memory_summary())`. The next code downloads the pre-trained EfficientNet model, loads the model weights to EfficientLite, and prepares the model for inference. Lastly, it prints a summary:

```
model_weights_file = "garbage-finetuned-efficientnet-b4"
model_url = f"https://github.com/PacktPublishing/Interpretable-Machine-
Learning-with-Python-2E/raw/main/models/{model_weights_file}.ckpt"
torchvision.datasets.utils.download_url(model_url, ".")
garbage_mdl = EfficientLite.load_from_checkpoint(
    f"{model_weights_file}.ckpt"
```

```
  )
  garbage_mdl = garbage_mdl.to(device).eval()
  print(summary(garbage_mdl))
```

The code is pretty straightforward but what's important to note is that we are choosing the b4 model for this chapter, which is in between b0 and b7 in terms of size, speed, and accuracy. You can change the last digit according to your hardware's abilities, but it might change some of the outcomes of this chapter's code. The preceding snippet outputs the following summary:

```
===========================================================================
Layer (type:depth-idx) Param # =============================================
======================
EfficientLite --
├─EfficientNet: 1-1 –
│   └─Conv2dStaticSamePadding: 2-1 1,296
│   │   └─ZeroPad2d: 3-1 –
│   └─BatchNorm2d: 2-2 96
│   └─ModuleList: 2-3 –
│   │   └─MBConvBlock: 3-2 2,940
│   │   └─MBConvBlock: 3-3 1,206
│   │   └─MBConvBlock: 3-4 11,878
│   │   └─MBConvBlock: 3-5 18,120
│   │   └─MBConvBlock: 3-6 18,120
│   │   └─MBConvBlock: 3-7 18,120
│   │   └─MBConvBlock: 3-8 25,848
│   │   └─MBConvBlock: 3-9 57,246
│   │   └─MBConvBlock: 3-10 57,246
│   │   └─MBConvBlock: 3-11 57,246
│   │   └─MBConvBlock: 3-12 70,798
│   │   └─MBConvBlock: 3-13 197,820
│   │   └─MBConvBlock: 3-14 197,820
│   │   └─MBConvBlock: 3-15 197,820
│   │   └─MBConvBlock: 3-16 197,820
│   │   └─MBConvBlock: 3-17 197,820
│   │   └─MBConvBlock: 3-18 240,924
│   │   └─MBConvBlock: 3-19 413,160
│   │   └─MBConvBlock: 3-20 413,160
│   │   └─MBConvBlock: 3-21 413,160
│   │   └─MBConvBlock: 3-22 413,160
```

```
| | └─MBConvBlock: 3-23 413,160
| | └─MBConvBlock: 3-24 520,904
| | └─MBConvBlock: 3-25 1,159,332
| | └─MBConvBlock: 3-26 1,159,332
| | └─MBConvBlock: 3-27 1,159,332
| | └─MBConvBlock: 3-28 1,159,332
| | └─MBConvBlock: 3-29 1,159,332
| | └─MBConvBlock: 3-30 1,159,332
| | └─MBConvBlock: 3-31 1,159,332
| | └─MBConvBlock: 3-32 1,420,804
| | └─MBConvBlock: 3-33 3,049,200
| └─Conv2dStaticSamePadding: 2-4 802,816
| | └─Identity: 3-34 –
| └─BatchNorm2d: 2-5 3,584
| └─AdaptiveAvgPool2d: 2-6 –
| └─Dropout: 2-7 –
| └─Linear: 2-8 21,516
| └─MemoryEfficientSwish: 2-9
=================================================================
Total params: 17,570,132
Trainable params: 17,570,132
Non-trainable params: 0 =====================================================
===============
```

It has pretty much everything we need to know about the model. It has two custom convolutional layers (`Conv2dStaticSamePadding`), each followed by a batch normalization layer (`BatchNorm2d`) and 32 `MBConvBlock` modules.

The network also has a memory-efficient implementation of the Swish activation function (`MemoryEfficientSwish`), which, like all activation functions, introduces non-linearity into the model. It's smooth and non-monotonic, which helps it converge more quickly while learning more complex and nuanced patterns. It also has a global average pooling operation (`AdaptiveAvgPool2d`), which reduces the spatial dimensions of the feature maps. It then has a first `Dropout` layer for regularization, followed by a fully connected layer (`Linear`) that takes it from 1792 nodes to 12. Dropout prevents overfitting by making a fraction of the neurons inactive in each update cycle. If you want to see more details of how the output shape of each layer gets reduced between one layer and another, enter the `input_size` into the summary – like summary(garbage_mdl, input_size=(64, 3, 224, 224)) – because the network was designed with a batch size of 64 in mind. Don't worry if none of these terms sound familiar to you. We will revisit them later.

Assessing the CNN classifier with traditional interpretation methods

We will first evaluate the model using the validation dataset with the `evaluate_multiclass_mdl` function. The arguments include the model (`garbage_mdl`), our validation data (`val_data`), as well as the class names (`labels_l`) and the encoder (`ohe`). Lastly, we won't plot the ROC curves (`plot_roc=False`). This function returns the predicted labels and probabilities, which we can store in variables for later use:

```
y_val_pred, y_val_prob = mldatasets.evaluate_multiclass_mdl(
    garbage_mdl, val_data,\
    class_l=labels_l, ohe=ohe, plot_roc=False
)
```

The preceding code generates both *Figure 7.5* with a confusion matrix and *Figure 7.6* with performance metrics for each class:
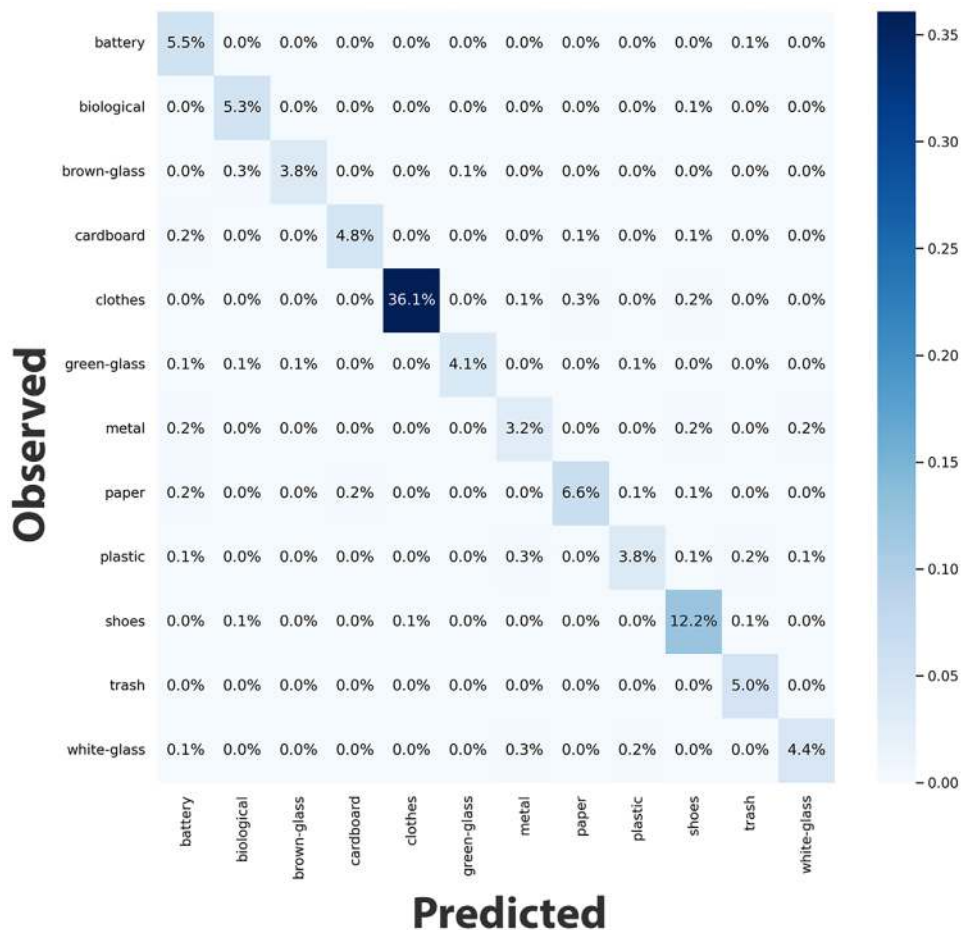

Figure 7.5: The confusion matrix for the validation dataset

Even though the confusion matrix in *Figure 7.5* seems to suggest a perfect classification, once you see the precision and recall breakdown in *Figure 7.6,* you can tell that the model had issues with metal, plastic, and white glass:

```
              precision    recall  f1-score   support

     battery      0.850     0.981     0.911        52
  biological      0.907     0.980     0.942        50
 brown-glass      0.972     0.897     0.933        39
   cardboard      0.957     0.918     0.938        49
     clothes      0.997     0.982     0.990       342
 green-glass      0.974     0.905     0.938        42
       metal      0.811     0.833     0.822        36
       paper      0.938     0.910     0.924        67
     plastic      0.897     0.814     0.854        43
       shoes      0.934     0.974     0.954       117
       trash      0.922     1.000     0.959        47
 white-glass      0.932     0.872     0.901        47

    accuracy                          0.947       931
   macro avg      0.924     0.922     0.922       931
weighted avg      0.949     0.947     0.947       931
```

Figure 7.6: The classification report for the validation dataset

You can expect a model to always reach 100% training accuracy if you train it for enough epochs using optimal hyperparameters. A near-perfect validation accuracy is harder to achieve, depending on how different these two are. We know that the validation dataset is simply a sample of images from the same collection, so it's not particularly surprising that 94.7% was achieved.

Now, let's repeat the same code snippet for the test dataset. This time, we want to see the ROC curves (plot_roc=True) but only the averages, and not on a class-by-class basis (plot_roc_class=False) because there are only four pictures per class. Given the small number of samples, we can display the numbers in the confusion matrix rather than percentages (pct_matrix=False):

```
y_test_pred, y_test_prob = mldatasets.evaluate_multiclass_mdl(
    garbage_mdl, test_data,\
    class_l=labels_l, ohe=ohe,\
    plot_roc=True, plot_roc_class=False, pct_matrix=False
)
```

The preceding code snippet generated the ROC curve in *Figure 7.7*, the confusion matrix in *Figure 7.8*, and the classification report in *Figure 7.9*:
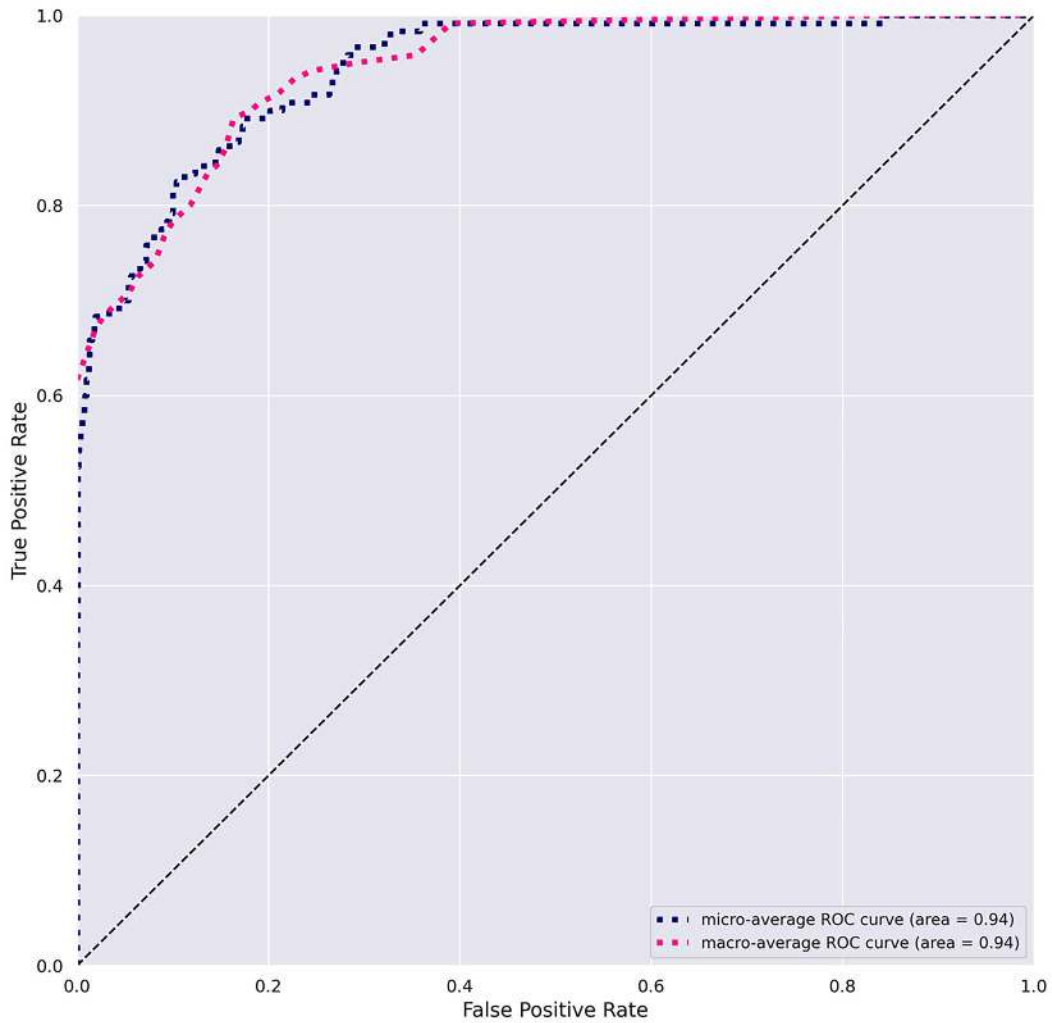

Figure 7.7: The ROC curve for the test dataset

The test ROC plot (*Figure 7.7*) shows the macro-average and micro-average ROC curves. The difference in both of these is in how they are calculated. Macro metrics are computed for each class independently and then averaged, treating each differently, whereas micro-averages factor in the contribution or representation of each class; generally, micro-averages are more reliable.
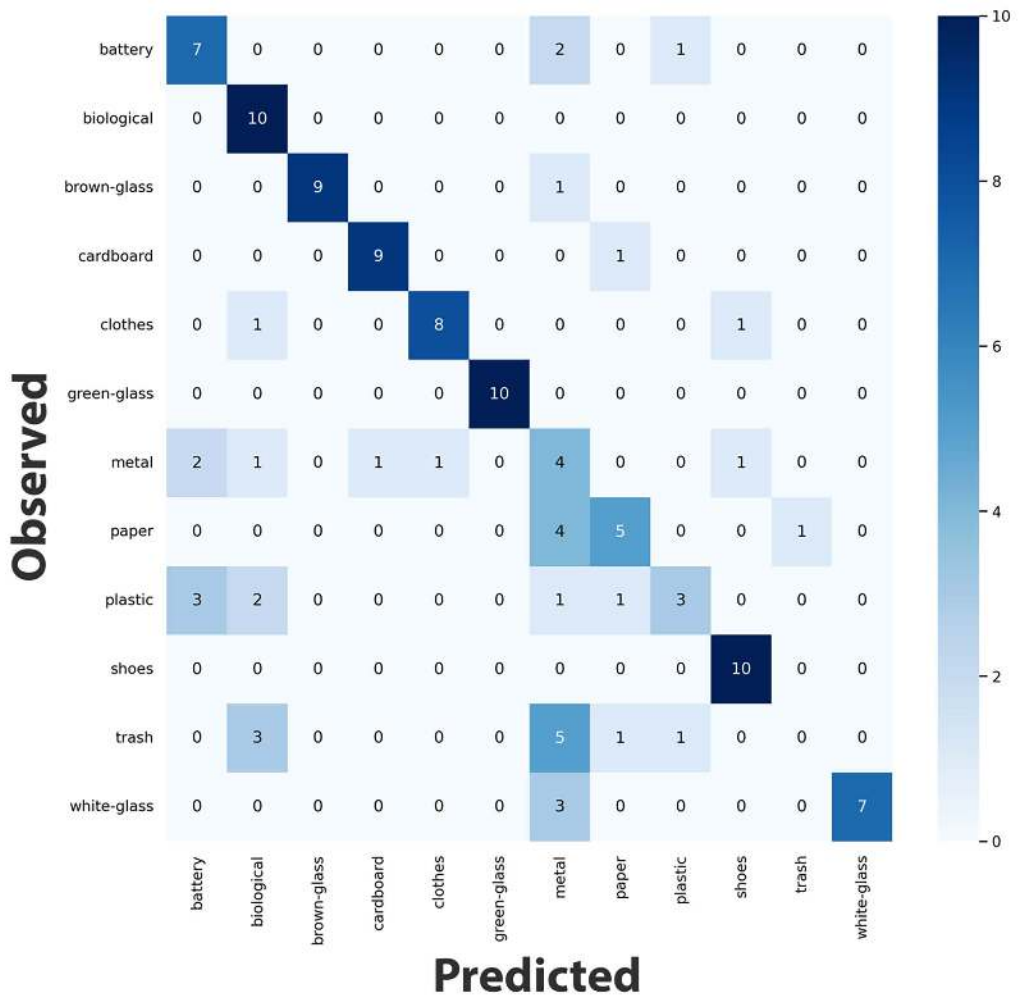
Figure 7.8: The confusion matrix for the test dataset

If we take a look at the confusion matrix in *Figure 7.8*, we can tell that only biological, green glass, and shoes are getting 10-out-of-10 classifications. However, a lot of items are being misclassified as biologicals and shoes. On the other hand, many items are more often than not misclassified, such as metal, paper, and plastic. Many of them are similar in shape or color, so you could understand how that would happen, but how does a piece of metal get confused with white glass, or paper with a battery?

	precision	recall	f1-score	support
battery	0.583	0.700	0.636	10
biological	0.588	1.000	0.741	10
brown-glass	1.000	0.900	0.947	10
cardboard	0.900	0.900	0.900	10
clothes	0.889	0.800	0.842	10
green-glass	1.000	1.000	1.000	10
metal	0.200	0.400	0.267	10
paper	0.625	0.500	0.556	10
plastic	0.600	0.300	0.400	10
shoes	0.833	1.000	0.909	10
trash	0.000	0.000	0.000	10
white-glass	1.000	0.700	0.824	10
accuracy			0.683	120
macro avg	0.685	0.683	0.668	120
weighted avg	0.685	0.683	0.668	120

Figure 7.9: The predictive performance metrics for the test dataset

When classification models are discussed in a business setting, stakeholders are often only interested in one number: accuracy. It's easy to let this drive the discussion, but there's much more nuance to it. For instance, the disappointing test accuracy (68.3%) could mean many things. It could mean that six classes are getting perfect classification, and all others are not, or that 12 classes are getting only half misclassified. There are many possibilities of what could be going on.

In any case, when dealing with a multiclass classification problem, even an accuracy below 50% might not be as bad as it seems. Consider that the **no information rate** represents the accuracy that can be achieved by a naive model that always predicts the most frequent class in the dataset. It serves as a benchmark to ensure that the developed model is providing insights beyond this simplistic approach. And with 12 evenly split, the **no information rate** is likely to be around 8.33% (100%/12 classes), so 68% is still orders of magnitude higher than that. In fact, there is less of a leap to 100%! To a machine learning practitioner, this means that if we judge solely based on test accuracy results, the model is still learning something of value that can be improved upon.

In any case, the predictive performance metrics in *Figure 7.9* for the test dataset are consistent with what we saw in the confusion matrix. Biological gets high recall but low precision and metal, paper, plastic, and trash are low for both.

Determining what misclassifications to focus on

We have already noticed some exciting misclassifications we can focus on:

- **Metal false positives:** 16 out of the 120 samples in the test dataset were misclassified as metal. That's 42% of all misclassifications! What is it about metal that renders it so easily confused with other garbage according to the model?

- **Plastic false negatives:** 70% of all true plastic samples were misclassified. Thus, plastics had the lowest recall of any material besides trash. It's easy to tell why trash would be so difficult to classify because it's exceedingly diverse but not plastic.

We should also examine some true positives to contrast these misclassifications. Namely, batteries because they get many false positives as metals and plastic, and white glass because it gets false negatives as metals 30% of the time. Since there are so many metal false positives, we should narrow them down to just those that are for batteries.

To visualize the tasks ahead, we can create a DataFrame (`preds_df`) with the true labels (`y_true`) in one column and predicted labels in another (`y_pred`). And to understand how certain the models are of these predictions, we can create another DataFrame with the probabilities (`probs_df`). We can generate column totals for these probabilities to sort the columns according to which category the model is most certain about across all samples. Then, we can concatenate our predictions DataFrame with the first 12 columns from our probabilities DataFrame:

```
preds_df = pd.DataFrame({'y_true':[labels_l[o] for o in y_test],\
                         'y_pred':y_test_pred})

probs_df = pd.DataFrame(y_test_prob*100).round(1)
probs_df.loc['Total']= probs_df.sum().round(1)
probs_df.columns = labels_l
probs_df = probs_df.sort_values('Total', axis=1, ascending=False)
probs_df.drop(['Total'], axis=0, inplace=True)
probs_final_df = probs_df.iloc[:,0:12]


preds_probs_df = pd.concat([preds_df, probs_final_df], axis=1)
```

Let's now output the DataFrame with color coding for the prediction instances we are interested in assessing. On one hand, we have the metal false positives and, on the other, the plastic false negatives. But we also have the true positives for battery and white glass. Lastly, we have bolded all probabilities over 50% and hidden all probabilities of 0% so that it's easier to spot any predictions with high probabilities:

```
num_cols_l = list(preds_probs_df.columns[2:])
num_fmt_dict = dict(zip(num_cols_l, ["{:,.1f}%"]*len(num_cols_l)))
preds_probs_df[
    (preds_probs_df.y_true!=preds_probs_df.y_pred)
    | (preds_probs_df.y_true.isin(['battery', 'white-glass']))
```

```
].style.format(num_fmt_dict).apply(
    lambda x: ['background: lightgreen' if (x[0] == x[1])\
               else '' for i in x], axis=1
).apply(
    lambda x: ['background: orange' if (x[0] != x[1] and\
               x[1] == 'metal' and x[0] == 'battery')\
               else '' for i in x], axis=1
).apply(
    lambda x: ['background: yellow' if (x[0] != x[1] and\
                                        x[0] == 'plastic')\
               else '' for i in x], axis=1
).apply(
    lambda x: ['font-weight: bold' if isinstance(i, float)\
                                        and i >= 50\
               else '' for i in x], axis=1
).apply(
    lambda x: ['color:transparent' if i == 0.0\
               else '' for i in x], axis=1)
```

The preceding code snippet produces *Figure 7.10*. We can tell by the highlights which are the metal false positives and the plastic false negatives, as well as which would be the true positives: #0-6 for battery, and #110-113 and #117-119 for white glass:

	y_true	y_pred	biological	metal	shoes	battery	paper	clothes	white-glass
0	battery	battery	0.00%	0.10%	0.00%	**99.70%**	0.10%	0.00%	0.00%
1	battery	battery	0.10%	1.50%	0.30%	**96.70%**	0.30%	0.00%	0.00%
2	battery	battery	0.00%	0.10%	0.00%	**99.60%**	0.20%	0.00%	0.00%
3	battery	battery	0.30%	0.40%	0.10%	**97.40%**	1.00%	0.00%	0.00%
4	battery	battery	1.00%	5.60%	0.60%	**58.80%**	1.20%	0.10%	1.90%
5	battery	battery	0.30%	1.00%	0.50%	**94.70%**	0.50%	0.00%	0.20%
6	battery	battery	0.00%	0.40%	0.00%	**99.00%**	0.30%	0.00%	0.00%
7	battery	metal	0.40%	**82.40%**	2.10%	1.10%	1.50%	1.10%	3.60%
8	battery	metal	0.00%	**92.60%**	0.10%	1.60%	0.60%	0.00%	0.30%
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
65	metal	clothes	2.90%	5.20%	2.70%	3.90%	4.80%	**69.60%**	1.20%
66	metal	battery	1.30%	20.40%	2.00%	**64.30%**	2.50%	0.20%	0.60%
67	metal	cardboard	10.70%	6.20%	3.20%	9.60%	13.80%	1.00%	1.10%
68	metal	shoes	4.10%	14.20%	43.70%	1.80%	28.70%	1.50%	0.90%
69	metal	battery	2.60%	8.30%	18.90%	**57.60%**	8.00%	0.60%	0.20%
73	paper	metal	1.30%	**74.80%**	9.20%	2.50%	5.40%	1.70%	1.40%
77	paper	metal	3.40%	29.40%	5.20%	2.40%	7.70%	1.50%	15.30%
83	plastic	battery	3.90%	5.00%	7.80%	46.70%	10.00%	1.30%	0.70%
84	plastic	metal	11.10%	19.00%	2.50%	8.10%	13.50%	15.10%	4.40%
85	plastic	battery	4.20%	5.20%	5.20%	36.30%	27.90%	0.30%	0.80%
86	plastic	biological	36.70%	2.80%	10.90%	6.40%	18.60%	1.40%	1.50%
87	plastic	paper	1.80%	1.90%	0.90%	5.20%	**74.10%**	0.60%	1.10%
88	plastic	biological	41.10%	2.30%	6.00%	3.30%	21.90%	1.90%	1.90%
89	plastic	battery	1.20%	1.70%	0.40%	**88.60%**	2.00%	0.10%	0.20%
100	trash	biological	49.90%	5.00%	4.10%	10.00%	5.30%	4.60%	1.70%
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
109	trash	metal	19.30%	24.80%	17.50%	7.20%	4.30%	1.30%	2.40%
110	white-glass	white-glass	0.20%	3.40%	0.40%	0.20%	0.20%	0.10%	**90.70%**
111	white-glass	white-glass	0.00%	1.60%	0.50%	0.00%	0.10%	0.00%	**95.80%**
112	white-glass	white-glass	0.00%	0.10%	0.00%	0.00%	0.00%	0.00%	**99.30%**
113	white-glass	white-glass	0.00%	0.10%	0.00%	0.00%	0.00%	0.00%	**95.50%**
114	white-glass	metal	0.10%	**82.50%**	0.10%	2.20%	0.20%	0.00%	1.50%
115	white-glass	metal	0.10%	**88.70%**	0.40%	0.50%	0.30%	0.00%	5.70%
116	white-glass	metal	3.70%	41.90%	3.10%	3.30%	4.00%	1.00%	10.40%
117	white-glass	white-glass	0.10%	1.60%	0.00%	0.10%	0.10%	0.00%	**94.90%**
118	white-glass	white-glass	0.10%	0.30%	0.10%	0.00%	0.00%	0.00%	**97.40%**
119	white-glass	white-glass	0.00%	0.40%	0.10%	0.00%	0.10%	0.00%	**95.70%**

Figure 7.10: Table with all 38 misclassifications in the test dataset, selected true positives, and their true and predicted labels, as well as their predicted probabilities

We can easily store the indexes for these instances in lists with the following code. That way, for future reference, we can iterate through these lists to assess individual predictions or subset arrays with them to perform interpretation tasks for the entire group. As you can tell, we have lists for all four groups:

```python
plastic_FN_idxs = preds_df[
    (preds_df['y_true'] !=preds_df['y_pred'])
    & (preds_df['y_true'] == 'plastic')
].index.to_list()
metal_FP_idxs = preds_df[
    (preds_df['y_true'] != preds_df['y_pred'])
    & (preds_df['y_pred'] == 'metal')
    & (preds_df['y_true'] == 'battery')
].index.to_list()
battery_TP_idxs = preds_df[
    (preds_df['y_true'] ==preds_df['y_pred'])
    & (preds_df['y_true'] == 'battery')
].index.to_list()
wglass_TP_idxs = preds_df[
    (preds_df['y_true'] == preds_df['y_pred'])
    & (preds_df['y_true'] == 'white-glass')
].index.to_list()
```

Now that we have all our data preprocessed, the model is fully loaded and lists the groups of predictions to debug. Now we can move forward. Let the interpretation begin!

Visualizing the learning process with activation-based methods

Before we get into discussing activations, layers, filters, neurons, gradients, convolutions, kernels, and all the fantastic elements that make up a CNN, let's first briefly revisit the mechanics of a CNN and one in particular.

The convolution layer is the essential building block of a CNN, which is a sequential neural network. It convolves the input with **learnable filters**, which are relatively small but are applied across the entire width, height, and depth at specific distances or **strides**. Each filter produces a two-dimensional **activation map** (also known as a **feature map**). It's called an activation map because it denotes positions of activations in the images – in other words, where specific "features" are located. In this context, a feature is an abstract spatial representation that, downstream in the process, is reflected in the learned weights of fully connected (**linear**) layers. For instance, in the garbage CNN case, the first convolutional layer has 48 filters with a 3 × 3 kernel, a 2 × 2 stride, and static padding, which ensure that the output maps maintain the same size as the inputs. Filters are template matching because they end up activating areas of the activation map when certain patterns are found in the input image.

But before we get to our fully connected layers, we have to reduce the dimensions of our filters until they have a workable size. For instance, if we flattened the output of our first convolution ($48 \times 112 \times 112$), we would have over 602,000 features. I think we can all agree that that would be too much to feed into a fully connected layer. Even if we used enough neurons to handle this workload, we probably wouldn't have captured enough spatial representations for the neural network to make sense of the images. For this reason, convolutional layers are often paired with pooling layers, which downsample the input – in other words, they reduce the dimensionality of the data. In this case, there's an adaptive average pooling layer (`AdaptiveAvgPool2d`) that performs an average across all the channels as well as many pooling layers within the **Mobile Inverted Bottleneck Convolution Blocks** (`MBConvBlock`).

Incidentally, `MBConvBlock`, `Conv2dStaticSamePadding`, and `BatchNorm2d` are the building blocks of the EfficientNet architecture. These components work together to create a highly efficient and accurate convolutional neural network:

- `MBConvBlock`: Mobile inverted bottleneck convolution blocks that form the core of the EfficientNet architecture. In traditional convolutional layers, filters are applied across all input channels simultaneously, resulting in a high number of computations, but `MBConvBlocks` divide the process into two steps: first, they apply depthwise convolutions that handle each input channel separately, and then use pointwise (1 x 1) convolutions to combine the information from different channels. For this reason, inside the `MBConvBlock` modules for B0, there are three convolutional layers: a depthwise convolution, a pointwise (1 x 1) convolution (called project convolution), and another pointwise (1 x 1) convolution (called expand convolution) in some blocks. However, the first block only contains two convolutional layers (depthwise and project convolutions) because it doesn't have an expand convolution. For B4, the architecture is similar except more convolutions are stacked in each block and there are twice as many `MBConvBlocks`. Naturally, B7 has many more blocks and convolutional layers. For B4, there are a total of 158 convolutional operations between the 32 `MBConvBlocks`.

- `Conv2dStaticSamePadding`: Unlike traditional convolutional layers (such as `Conv2d`), these don't reduce the dimensions. It ensures the input and output feature maps have the same spatial dimensions.

- `BatchNorm2d`: Batch normalization layers that help stabilize and accelerate training by normalizing the input features, which helps keep the distribution of the input features consistent during training.

Once the over 230 convolutional and pooling operations are performed, we are left with a flattened output of a more workable size: 1,792 features, which the fully connected layer converts into 12, which, leveraging **softmax** activation, outputs probabilities between 0 and 1 for each of the classes. In the garbage CNN, there is a **dropout** layer involved to help regularize the training. We can ignore this entirely because, for inference, they are ignored.

If this wasn't entirely clear, don't fret! The sections that follow will demonstrate visually through activations, gradients, and perturbations how the network probably learned or did not learn image representations.

Intermediate activations

For inference, the image goes through the network's input and the prediction comes out through the output traversing every single layer. However, one of the advantages of having a sequential and layered architecture is that we can extract any layer's output and not just the final layer. The **intermediate activations** are simply the outputs of any of the convolution or pooling layers. They are activation maps because, after an activation function has been applied, the brighter spots map to the image's features. In this case, the model used ReLU on all convolutional layers, so that is what activates the spots. We are only interested in the convolutional layers' intermediate activations because the pooling layers are simply downsampled versions of these ones. Why not see the higher-resolution version instead?

As the filters become smaller in width and height, the learned representations will be larger. In other words, the first convolutional layer may be about details such as texture, the following one about edges, and the last one about shapes. We must then flatten the convolutional layers' output to feed it to the multilayer perceptron that takes over from then on.

What we will do now is extract activations for some of the convolutional layers. In B4, there are 158, so we can't do all of them! To this end, we will obtain the first level of layers with `model.children()`, and iterate across them. We will append the two `Conv2dStaticSamePadding` layers from this top level into a `conv_layers` list. But we will also go deeper, appending the first convolutional layer for the first six `MBConvBlock` layers in the `ModuleList` layers. In the end, we should have eight convolutional layers – the six in the middle belonging to Mobile Inverted Bottleneck Convolution blocks:

```python
conv_layers = []

model_children = list(garbage_mdl.model.children())
for model_child in model_children:
    if (type(model_child) ==\
                efficientnet_pytorch.utils.Conv2dStaticSamePadding):
        conv_layers.append(model_child)
    elif (type(model_child) == torch.nn.modules.container.ModuleList):
        module_children = list(model_child.children())
        module_convs = []
        for module_child in module_children:
            module_convs.append(list(module_child.children())[0])
        conv_layers.extend(module_convs[:6])
print(conv_layers)
```

Before we iterate across all of them producing activation maps for each convolutional layer, let's do it for a single filter and layer:

```python
idx = battery_TP_idxs[0]
tensor = test_data[idx][0][None, :].to(device)
label = y_test[idx]
method = attr.LayerActivation(garbage_mdl, conv_layers[layer]
```

```
attribution = method.attribute(tensor).detach().cpu().numpy()
print(attribution.shape)
```

The preceding snippet extracts the `tensor` for the first battery true positive (`battery_TP_idxs[0]`). Then, it initializes the `LayerActivation` attribution method with the model (`garbage_mdl`) and the first convolutional layer (`conv_layers[0]`). Using the `attribute` function, it creates an `attribution` with this method. For the shape of the attribution, we should get (`1, 48, 112, 112`). The tensor was for a single image, so it makes sense that the first number is a one. The next number corresponds to the number of filters, followed by the width and height dimensions of each filter. Regardless of the kind of attribution, the numbers inside each attribution relate to how a pixel in the input is seen by the model. Interpretation varies according to the method. However, generally, it is interpreted that higher numbers mean more of an impact on the outcome, but attributions may also have negative numbers, which mean the opposite.

Let's visualize the first filter, but before we do that, we must decide what colormap to use. A colormap will determine what colors to assign to different numbers as a gradient. For instance, the following colormap has white for `0` (`#ffffff` in hexadecimal), a medium gray for `0.25`, and black (`#000000` in hexadecimal) for `1` with a gradient between these colors:

```
cbinary_cmap = LinearSegmentedColormap.from_list('custom binary',
                                        [(0, '#ffffff'),
                                         (0.25, '#777777'),
                                         (1, '#000000')])
```

You can also use any of the named colormaps from `https://matplotlib.org/stable/tutorials/colors/colormaps.html`, rather than using your own. Next, let's plot the attribution for the first filter like this:

```
filter = 0
filter_attr = attribution[0,filter]
filter_attr = mldatasets.apply_cmap(filter_attr, cbinary_cmap, 'positive')

y_true = labels_l[label]
y_pred = y_test_pred[idx]

fig, ax = plt.subplots(1, 1, figsize=(8, 6))
fig.suptitle(f"Actual label: {y_true}, Predicted: {y_pred}", fontsize=16)
ax.set_title(
    f"({method.get_name()} Attribution for Filter #{filter+1} for\
        Convolutional Layer #{layer+1})",
    fontsize=12
)

ax.imshow(filter_attr)
ax.grid(False)
```

```
fig.colorbar(
    ScalarMappable(norm='linear', cmap=cbinary_cmap),
    ax=ax,
    orientation="vertical"
)
plt.show()
```

The code saves the first filter for the first image from the attribution to `filter_attr` and then leverages the `apply_cmap` utility function to convert the attribution array to a normalized colormap, which is then plotted with `imshow`. The rest of the code plots the title with the actual and predicted label for the image and a `colorbar` with the colormap. The snippet will output *Figure 7.11*:
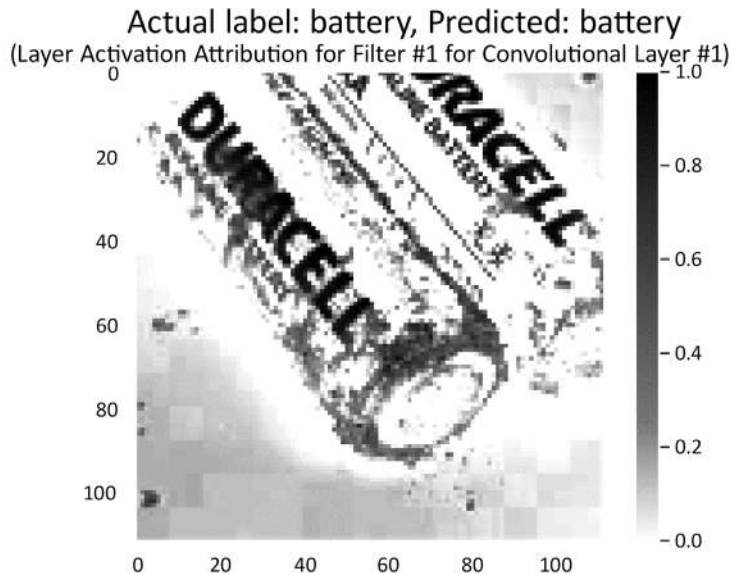
Figure 7.11: Intermediate activation map for the first filter for the first convolutional layer for the first true positive battery sample

As you can tell in *Figure 7.11*, it seems like the intermediate activations for the first filter are finding the edges of the battery and the most prominent text.

Next, we will iterate across all computational layers and every battery and visualize attributions for each one. Now, some of these attribution operations can be computationally expensive, so it's important to clear the GPU cache (`clear_gpu_cache()`) in between them:

```python
for l, layer in enumerate(conv_layers):
    layer = conv_layers[l]
    method = attr.LayerActivation(garbage_mdl, layer)
    for idx in battery_TP_idxs:
        orig_img = mldatasets.tensor_to_img(test_400_data[idx][0],\
                                            norm_std, norm_mean,\
                                            to_numpy=True)
        tensor = test_data[idx][0][None, :].to(device)
        label = int(y_test[idx])
        attribution = method.attribute(tensor).detach().cpu().numpy()
        viz_img =  mldatasets.create_attribition_grid(attribution,\
                            cmap='copper', cmap_norm='positive')
        y_true = labels_l[label]
        y_pred = y_test_pred[idx]
        probs_s = probs_df.loc[idx]
        name = method.get_name()
        title = f'CNN Layer #{l+1} {name} Attributions for Sample #{idx}'
        mldatasets.compare_img_pred_viz(orig_img, viz_img, y_true,\
                                        y_pred, probs_s, title=title)
    clear_gpu_cache()
```

The preceding snippet should look fairly familiar. Where it's different is that it's placing every attribution map for every filter in a grid (`viz_img`) with `create_attribition_grid`. It could just then display it with `plt.imshow` as before, but instead, we will leverage a utility function called `compare_img_pred_viz` to visualize the attribution(s) side by side with the original image (`orig_img`). It also takes the sample's actual label (`y_true`) and predicted label (`y_pred`). Optionally, we can provide a `pandas` series with the probabilities for this prediction (`probs_s`) and a `title`. It generates 56 images in total, including *Figures 7.12*, *7.13*, and *7.14*.

As you can tell from *Figure 7.12*, the first convolutional layer seems to be picking up on the battery's letters as well as its contours:
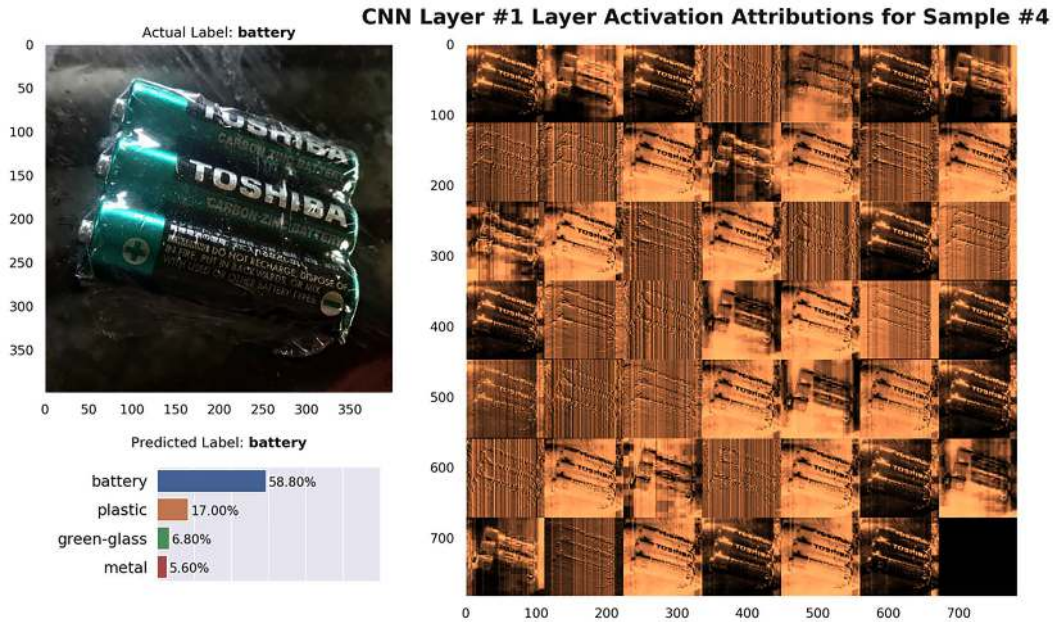
Figure 7.12: Intermediate activations for the first convolutional layer for battery #4

However, *Figure 7.13* shows how, by the fourth convolutional layer, the network understands a battery's contours better:
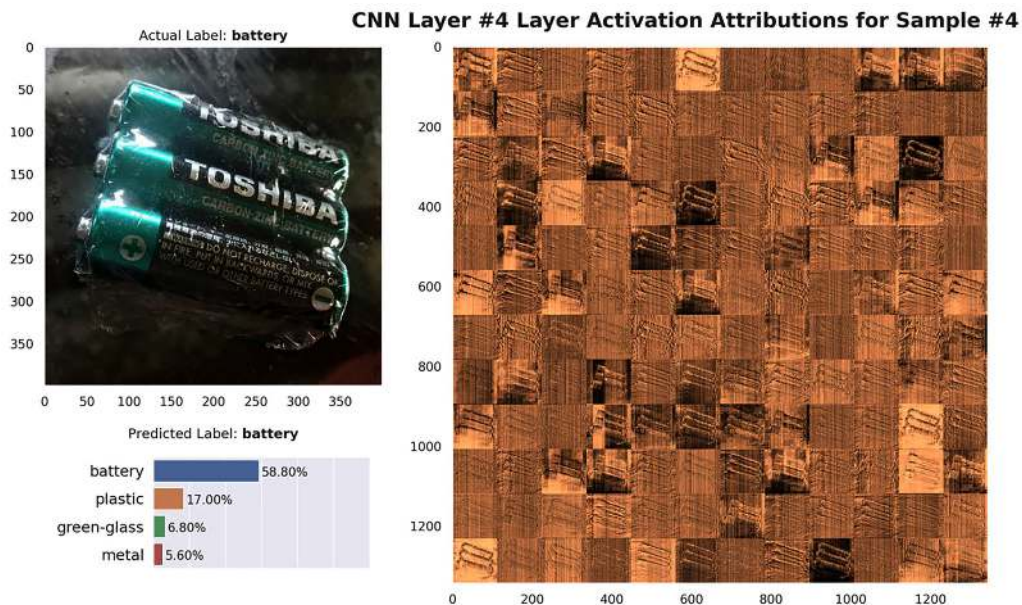

Figure 7.13: Intermediate activations for the fourth convolutional layer for battery #4

The last convolutional layer in *Figure 7.14* is impossible to interpret because there are 1,792 filters that are 7 pixels wide and high, but rest assured, there are some very high-level features encoded in those tiny maps:
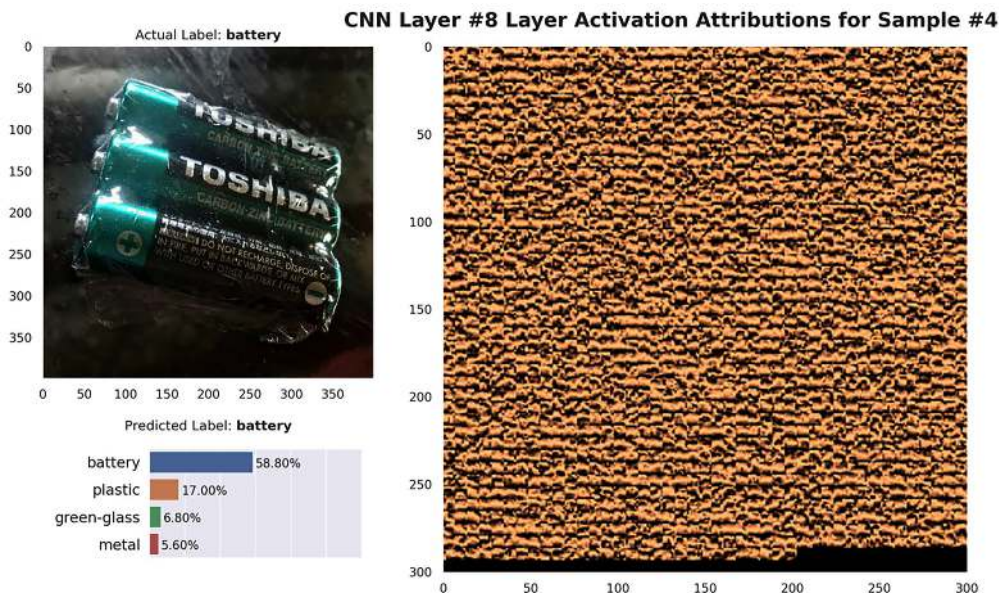
Figure 7.14: Intermediate activations for the last convolutional layer for battery #4

Extracting intermediate activations can provide you with some insight on a sample-by-sample basis. In other words, it's a **local model interpretation method**. It's by no means the only layerwise-attribution method. Captum has more than ten layer attribution methods: `https://github.com/pytorch/captum#about-captum`.

Evaluating misclassifications with gradient-based attribution methods

Gradient-based methods calculate **attribution maps** for each classification with both forward and background passes through the CNN. As the name suggests, these methods leverage the gradients in the backward pass to compute the attribution maps. All of these methods are local interpretation methods because they only derive a single interpretation per sample. Incidentally, attributions in this context mean that we are attributing the predicted labels to areas of an image. They are often called **sensitivity maps** in academic literature, too.

To get started, we will first need to create an array with all of our misclassification samples (`X_misclass`) from the test dataset (`test_data`) using the combined indexes for all of our misclassifications of interest (`misclass_idxs`). Since there aren't that many misclassifications, we are loading a single batch of them (`next`):

```
misclass_idxs = metal_FP_idxs + plastic_FN_idxs[-4:]
misclass_data = torch.utils.data.Subset(test_data, misclass_idxs)
```

```
misclass_loader = torch.utils.data.DataLoader(misclass_data,\
                                              batch_size = 32)
X_misclass, y_misclass = next(iter(misclass_loader))
X_misclass, y_misclass = X_misclass.to(device), y_misclass.to(device)
```

The next step is to create a utility function we can reuse to obtain the attribution maps for any method. Optionally, we can smooth the map with a method called `NoiseTunnel` (https://github.com/pytorch/captum#getting-started). We will cover this method in more detail later:

```
def get_attribution_maps(method, model, device,X,y=None,\
                         init_args={}, nt_type=None, nt_samples=10,\
                         stdevs=0.2, **kwargs):
    attr_maps_size = tuple([0] + list(X.shape[1:]))
    attr_maps = torch.empty(attr_maps_size).to(device)

    attr_method = method(model, **init_args)
    if nt_type is not None:
        noise_tunnel = attr.NoiseTunnel(attr_method)
        nt_attr_maps = torch.empty(attr_maps_size).to(device)

    for i in tqdm(range(len(X))):
        X_i = X[i].unsqueeze(0).requires_grad_()
        model.zero_grad()
        extra_args = {**kwargs}
        if y is not None:
            y_i = y[i].squeeze_()
            extra_args.update({"target":y_i})

        attr_map = attr_method.attribute(X_i, **extra_args)
        attr_maps = torch.cat([attr_maps, attr_map])

        if nt_type is not None:
            model.zero_grad()
            nt_attr_map = noise_tunnel.attribute(
                X_i, nt_type=nt_type, nt_samples=nt_samples,\
                stdevs=stdevs, nt_samples_batch_size=1, **extra_args)
            nt_attr_maps = torch.cat([nt_attr_maps, nt_attr_map])
        clear_gpu_cache()
    if nt_type is not None:
        return attr_maps, nt_attr_maps

    return attr_maps
```

The preceding code can create attribution maps for any Captum method for a given model and device. To that end, it takes tensors for the images, `X`, and their corresponding labels, `y`. The labels are optional and only needed if the attribution method is targeted – most methods are. Most attribution methods (`attr_method`) are initialized with only the model, but some require some additional arguments (`init_args`). Where they tend to have the most arguments is when the attribution is generated with the `attribute` function, which is why we have the `**kwargs` collect additional arguments in the `get_attribution_maps` function and place them in this call.

One important thing to note is that, in this function, we iterate across all the samples in the `X` tensor and create the attribute maps for each one independently. This is often unnecessary because the attribute methods are all equipped to process a batch at once. However, there's a risk the hardware can't handle an entire batch, and at the time of this writing, very few methods come with an `internal_batch_size` argument, which can limit how many samples are processed at a time. What we are doing here is essentially equivalent to setting this number to 1 every single time in an effort to ensure that we don't run into memory issues. However, if you have powerful hardware, you can rewrite the function to process the `X` and `y` tensors directly.

Next, we will perform our first gradient-based attribution method.

Saliency maps

Saliency maps rely on the absolute value of gradients. The intuition is that it will find the pixels in the image that can be perturbed the least so that the output changes the most with these values. It doesn't perform perturbations, so it doesn't validate the hypothesis, and the use of absolute values prevents it from finding other evidence to the contrary.

This first saliency map method was groundbreaking at the time and has inspired a bunch of different methods. It's typically nicknamed "vanilla" to distinguish it from other saliency maps.

Generating saliency maps for all of our misclassified samples is relatively simple with our `get_attribution_maps` function. All you need is the Captum attribution method (`attr.Saliency`), model (`garbage_mdl`), device, and the tensors for the misclassified samples (`X_misclass` and `y_misclass`):

```
saliency_maps = get_attribution_maps(attr.Saliency, garbage_mdl,\
                                     device, X_misclass, y_misclass)
```

We can plot the output of one of these saliency maps, the fifth one, side by side with the sample image to provide context. Matplotlib can do this easily with a `subplots` grid. We will make a 1 × 3 grid and place the sample image in the first spot, its saliency heatmap in the second, and one overlayed over the other in the third. As we have done with previous attribution maps, we can use `tensor_to_img` to convert the images to `numpy` arrays while also applying a colormap to the attribution. It uses the jet colormap (`cmap='jet'`) by default to make the salient areas appear more striking:

```
pos = 4
orig_img = mldatasets.tensor_to_img(X_misclass[pos], norm_std,\
                                    norm_mean, to_numpy=True)
attr_map = mldatasets.tensor_to_img(
    saliency_maps[pos], to_numpy=True,\
```

```
      cmap_norm='positive'
)

fig, axs = plt.subplots(1, 3, figsize=(15,5))

axs[0].imshow(orig_img)
axs[0].grid(None)
axs[0].set_title("Original Image")

axs[1].imshow(attr_map)
axs[1].grid(None)
axs[1].set_title("Saliency Heatmap")

axs[2].imshow(np.mean(orig_img, axis=2), cmap="gray")
axs[2].imshow(attr_map, alpha=0.6)
axs[2].grid(None)
axs[2].set_title("Saliency Overlayed")

idx = misclass_idxs[pos]
y_true = labels_l[int(y_test[idx])]
y_pred = y_test_pred[idx]
plt.suptitle(f"Actual label: {y_true}, Predicted: {y_pred}")
plt.show()
```

The preceding code generates the plot in *Figure 7.15*:
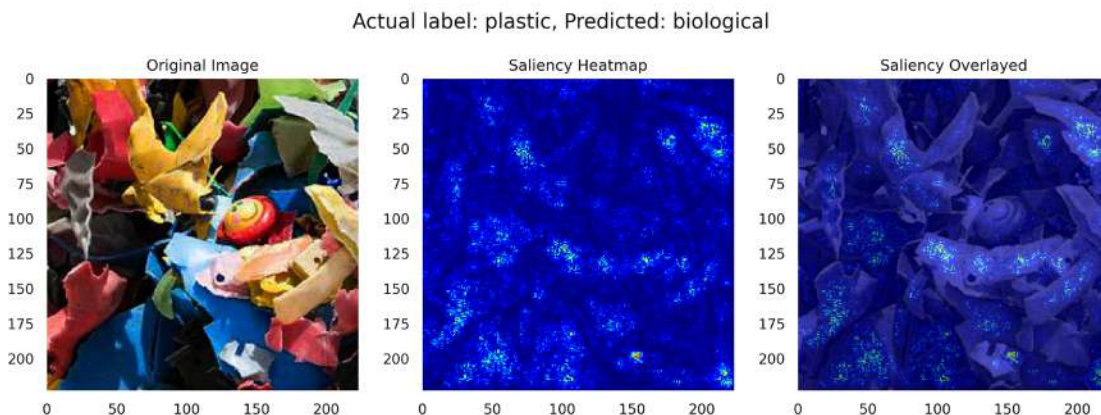
Figure 7.15: Saliency maps for plastic misclassified as biological waste

The sample image in *Figure 7.15* appears to be shredded plastic, but the prediction is for biological waste. The vanilla saliency map attributes that prediction mostly to the smoother duller areas of the plastic. It appears that the lack of specular highlights has thrown the model off, but typically, older broken pieces of plastic lose their shine.

> Specular highlights are bright spots of light that appear on the surface of an object when it reflects light. They are often the direct reflections of a light source and are more pronounced on shiny or glossy surfaces, such as metal, glass, or water.

Guided Grad-CAM

To discuss **guided Grad-CAM**, we first ought to discuss **CAM**, which stands for **Class Activation Map**. The way CAM works is that it removes all but the last fully connected layers, and it replaces the last **MaxPooling** layer with a **Global Average Pooling** (**GAP**) layer. A GAP layer calculates the average value of each feature map, reducing it to a single value per map, while a MaxPooling layer downsizes feature maps by selecting the maximum value from a set of values in a local region of the map. For instance, in this case:

1. The last convolutional layer outputs a tensor that is $1792 \times 7 \times 7$.
2. GAP reduces dimensions by merely averaging the last two dimensions of this tensor, producing a $1792 \times 1 \times 1$ tensor.
3. It then feeds this to a fully connected layer with 12 neurons corresponding to each class.
4. Once you retrain a CAM model and pass a sample image through the CAM model, it takes the weights from the last layer (a 1792×12 tensor) and extracts the values corresponding to the predicted class (a 1792×1 tensor).
5. Then, you calculate the dot product of the last convolutional layer's output ($1792 \times 7 \times 7$) with the weight tensor (1792×1).
6. This weighted sum will end with a $1 \times 7 \times 7$ tensor.
7. With bilinear interpolation to stretch it out to $1 \times 224 \times 224$, this becomes an upsampled class activation map. When you upsample data, you increase its dimensions.

The intuition behind CAM is that CNNs inherently retain spatial details in convolutional layers but they are, sadly, lost in fully connected layers. In fact, each filter in the last convolutional layer represents visual patterns at different spatial locations. Once weighted, they represent the most salient regions in the entire image. However, to apply CAM, you must radically modify a model and retrain it, and some models don't lend themselves easily to this.

As the name suggests, Grad-CAM is a similar concept but lacks the modifying and retraining hassle, and uses gradients instead – specifically, those of the class score (prior to softmax) concerning the convolutional layer's activation maps. GAP is performed on these gradients to obtain **neuron importance weights**. Then, we compute a weighted linear combination of activation maps with these weights, followed by a ReLU. The ReLU is very important because it ensures locating features that only positively influence the outcome. Like CAM, it is upsampled with bilinear interpolation to match the dimensions of the image.

Grad-CAM does have some shortcomings too, such as failing to identify multiple occurrences or the entirety of the object represented by the predicted class. Like CAM, the resolution of the activation maps may be limited by the final convolutional layer's dimensions, hence the upsampling.

For these reasons, we are using **guided Grad-CAM** instead. Guided Grad-CAM is a combination of Grad-CAM and guided backpropagation. Guided backpropagation is another visualization method that computes the gradients of the target class with respect to the input image, but it modifies the backpropagation process to only propagate positive gradients for positive activations. This results in a higher-resolution, more detailed visualization. This is achieved by performing an element-wise multiplication of the Grad-CAM heatmap (upsampled to the input image resolution) with the guided backpropagation result. The output is a visualization that emphasizes the most relevant features in the image for the given class, with higher spatial detail than Grad-CAM alone.

Generating Grad-CAM attribution maps for all of our misclassified samples can be done with our `get_attribution_maps` function. All you need is the Captum attribution method (`attr.GuidedGradCam`), model (`garbage_mdl`), device, and the tensors for the misclassified samples (`X_misclass` and `y_misclass`), and, within the method initialization arguments, a layer for which Grad-CAM attributions are computed:

```
gradcam_maps = get_attribution_maps(
    attr.GuidedGradCam, garbage_mdl, device, X_misclass,\
    y_misclass, init_args={'layer':conv_layers[3]}
)
```

Notice that we aren't using the last layer (which can be indexed with `7` or `-1`) but the fourth one (3). This is just to keep things interesting, but we can change it. Next, let's plot the attributions just as we have before. Nearly the same code is used except `saliency_maps` is replaced by `gradcam_maps`. The output is depicted in *Figure 7.16*.
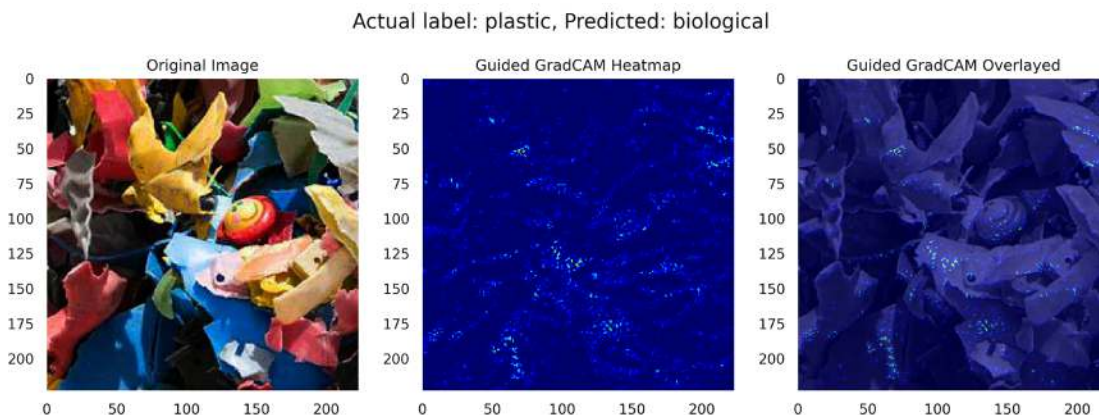
Figure 7.16: Guided Grad-CAM heatmaps for plastic misclassified as biological waste

As you can observe in *Figure 7.16*, similar smooth matte areas are highlighted as with the saliency attribution maps, that except guided Grad-CAM yields a few bright areas and edges.

Take all of this with a grain of salt. There is still a lot of ongoing debate in the CNN interpretation domain. And researchers are still coming up with new and better methods, and even techniques that are nearly perfect for most use cases still have flaws. Regarding CAM-like methods, there are many newer ones, such as **Score-CAM**, **Ablation-CAM**, and **Eigen-CAM**, which provide similar functionality but don't rely on gradients, which can be unstable and, therefore, occasionally unreliable. We won't discuss them here because, of course, they aren't gradient-based! But it's essential to note that it doesn't hurt to try different methods to see what works for your use case.

Integrated gradients

Integrated gradients (**IG**), also known as **path-integrated gradients**, is a technique that is not exclusive to CNNs. You can apply it to any neural network architecture because it computes the gradients of the output with respect to the inputs averaged all along a path between a **baseline** and the actual input. It is agnostic to the presence of convolutional layers. However, it requires the definition of a baseline, which is supposed to convey a lack of signal, like a uniformly colored image. In practice, for CNNs in particular, this is what a zero baseline represents, which, for every pixel, would usually mean a completely black image. Also, although the name suggests the use of **path integrals**, integrals aren't computed but approximated, with summation in sufficiently small intervals for a certain number of steps. For a CNN, this means it makes variations of the input image progressively darker or lighter until it becomes the baseline corresponding to the predefined number of steps. It then feeds these variations to the CNN, computes the gradients for each one, and averages them. The IG is the dot product of the image times the gradient averages.

Like Shapley values, IG is grounded in solid mathematical theory. In this case, it's the **fundamental theorem of calculus for line integrals**. The mathematical proof of the IG method ensures that the attributions of all the features add up to the difference between the model's prediction on the input data and its prediction on the baseline input. In addition to this property, which they call **completeness**, there is linearity preservation, symmetry preservation, and sensitivity. We won't describe each of these properties here. However, it's important to note that some interpretation methods satisfy notable mathematical properties, while others demonstrate their effectiveness in practical terms.

In addition to IG, we will also leverage `NoiseTunnel` to perform small random perturbations on the sample image – in other words, to add noise. It creates different noisy versions of the same sample image multiple times and then computes the attribution method for each. It then averages these attributions, potentially making the attribution maps much smoother, which is why this method is called **SmoothGrad**.

But wait, you may ask: Shouldn't it be a perturbation-based method then?! We've already dealt with several perturbation-based methods before in this book, from SHAP to anchors, and something they have in common is that they perturb the input to measure the effect on the output. SmoothGrad doesn't measure the impact on the outputs. It only helps yield a more robust attribution map because the mean attribution of perturbed inputs should make for more trustworthy attribution maps. We perform cross-validation to evaluate machine learning models for the same reason: the average metrics performed on different test datasets with slightly different distributions make for better metrics.

For IG, we will use very similar code as we did for Saliency, except we will add several arguments related to `NoiseTunnel`, such as the type of noise tunnel (`nt_type='smoothgrad'`), the sample variations to produce (`nt_samples=20`), and an amount of random noise to add to each one in standard deviations (`stdevs=0.2`). We will find that the more permuted samples to generate, the better, up to a point, and then it doesn't have much effect. However, there is such a thing as too much noise, and if you use too little, there won't be any effect:

```
ig_maps, smooth_ig_maps = get_attribution_maps(
    attr.IntegratedGradients, garbage_mdl, device, X_misclass,\
    y_misclass, nt_type='smoothgrad', nt_samples=20, stdevs=0.2
)
```

We can also optionally define the number of steps for the IG (`n_steps`). It's set to `50` by default, and we can also modify the baselines, which is a tensor of zeros by default. As we've done with Grad-CAM, we can plot our first sample image side by side with the IG map, but this time, we will modify the code to plot the SmoothGrad integrated gradients (`smooth_ig_maps`) in the third position, like this:

```
nt_attr_map = mldatasets.tensor_to_img(
    smooth_ig_maps[pos], to_numpy=True, cmap_norm='positive'
)

axs[2].imshow(nt_attr_map)
axs[2].grid(None)
axs[2].set_title("SmoothGrad Integrated Gradients")
```

The modified snippet outputs *Figure 7.17*:
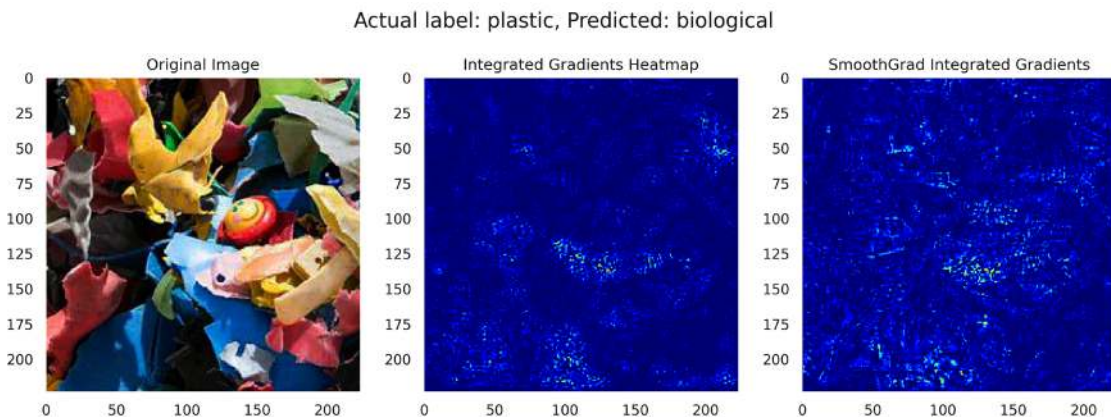
Figure 7.17: Integrated gradient heatmaps for plastic misclassified as biological waste

The areas in the IG heatmap in *Figure 7.17* coincide with many of the regions spotted by the saliency and guided Grad-CAM maps. However, there are more clusters of strong attributions in the bright yellow areas as well as in brownish shadowed areas, which is consistent with how some foods look when they are disposed of (like banana peels and rotten leafy greens). On the other hand, the bright orange and green areas aren't.

As for the SmoothGrad IG heatmap, it is striking how different this map is compared to the non-smooth IG heatmap. This is not always the case; often, it's just a smoother version. What likely happened was that the `0.2` noise distorted the attributions a bit too much, or that 20 perturbed samples weren't enough. However, it's tough to tell because it's also possible that SmoothGrad more accurately depicts the real story.

We won't do this now, but you can visually "tune" the `stdevs` and `nt_samples` parameters. You can try it with less noise and more samples, using a series of combinations, such as `0.1` and `80`, and `0.15` and `40`, trying to figure out whether you see a commonality between them. The one you go with is the one that most clearly depicts this consistent story. One of the shortcomings of SmoothGrad is having to define optimal parameters. Incidentally, IG also has the same issue with defining the baselines and number of steps (`n_steps`). The default baseline won't work in cases where the input image is too large or small, so it must be changed, and the authors of the IG paper suggest that 20-300 steps will approximate the integral within 5%.

Bonus method: DeepLIFT

IG has its detractors, who have made similar methods that avoid using gradients, such as **DeepLIFT**. IG can be sensitive to zero-valued gradients and discontinuities with gradients, which can lead to misleading attributions. But these point to general disadvantages shared by all gradient-based methods. For this reason, we are introducing the **Deep Learning Important FeaTures** algorithm (**DeepLIFT**). It's neither a gradient-based nor a perturbation-based method. It's a backpropagation-based approach!

In this section, we will contrast it with IG. Like IG and Shapley values, DeepLIFT was designed for **completeness**, and as such, complies with remarkable mathematical properties. In addition to that, like IG, DeepLIFT can also be applied to various deep learning architectures, including CNNs and **recurrent neural networks** (**RNNs**), making it versatile for different use cases.

DeepLIFT works by decomposing the output prediction of the model into contributions from each input feature, using the concept of "difference-from-reference." It backpropagates these contributions through the network layers to assign an importance score to each input feature.

More specifically, like IG, it uses a baseline that represents no specific information about any class. However, it then calculates the difference in the activations of each neuron between the input and the baseline, and it backpropagates these differences through the network, calculating each neuron's contribution to the output prediction. Then we sum the contributions for each input feature to obtain its importance score (attribution).

It's advantages over IG are as follows:

- **Reference-based:** Unlike gradient-based methods such as IG, DeepLIFT explicitly compares the input to a reference input, making the attributions more interpretable and meaningful.
- **Non-linear interactions:** DeepLIFT considers the non-linear interactions between neurons when computing attributions. It captures these interactions by considering the multipliers (the change in output due to the change in input) in each layer of the neural network.
- **Stability:** DeepLIFT is more stable than gradient-based methods, as it is less sensitive to small changes in the input, providing more consistent attributions. So, using a SmoothGrad is unnecessary on DeepLIFT attributions although highly recommended for gradient-based methods.

Overall, DeepLIFT provides a more interpretable, stable, and comprehensive approach to attributions, making it a valuable tool for understanding and explaining deep learning models.

Next, we will create DeepLIFT attribution maps in a similar fashion as we have done the others:

```
deeplift_maps = get_attribution_maps(attr.DeepLift, garbage_mdl,\
                                     device, X_misclass, y_misclass)
```

To plot an attribution map, nearly the same code as with Grad-CAM is used, except `gradcam_maps` is replaced by `deeplift_maps`. The output is depicted in *Figure 7.18*.
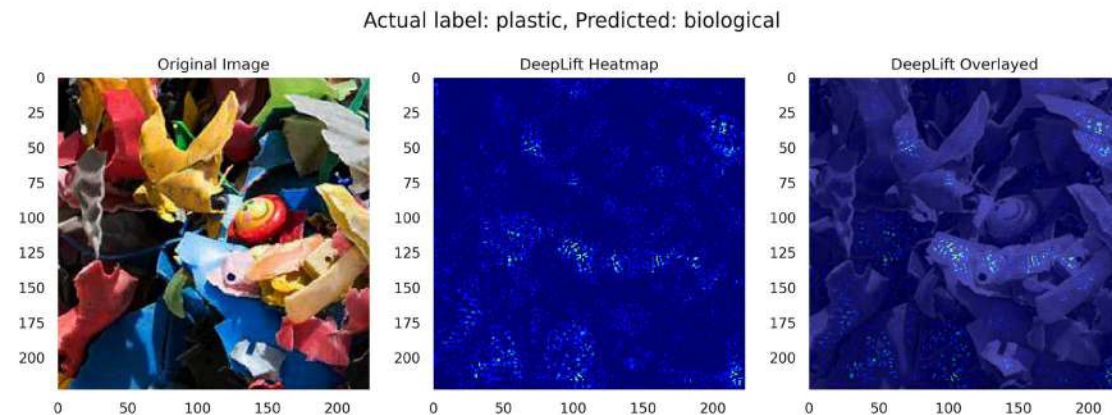
Figure 7.18: DeepLIFT heatmaps for plastic misclassified as biological waste

The attributions of *Figure 7.18* are not as noisy as in IG. But they also seem to cluster around some dull yellows and dark areas in the shadows; it also points toward dull greens near the top-right corner.

Tying it all together

Now, we will take everything that we have learned about gradient-based attribution methods and use it to understand the reasons for all the chosen misclassifications (the plastic false negatives and metal false positives). As we did with intermediate activation maps, we can leverage the `compare_img_pred_viz` function to place the higher-resolution sample image side by side with four attribution maps: saliency, Grad-CAM, SmoothGrad IG, and DeepLift. To this end, we first have to iterate all the misclassifications' positions and indexes and extract all the maps. Note that we are using `overlay_bg` in the `tensor_to_img` function to produce a new image overlaying the original image with the heatmap for each. Lastly, we concatenate the four attribution outputs into a single image (`viz_img`). Just as we have done before, we extract the actual label (`y_true`), predicted label (`y_pred`), and `pandas` series with the probabilities (`probs_s`) to add some context to the plot we will produce. The `for` loop will produce six plots but, for brevity's sake, we are only going to discuss three of them:

```python
for pos, idx in enumerate(misclass_idxs):
    orig_img = mldatasets.tensor_to_img(test_400_data[idx][0],\
                                    norm_std, norm_mean, to_numpy=True)
    bg_img = mldatasets.tensor_to_img(test_data[idx][0],\
                                    norm_std, norm_mean, to_numpy=True)
    map1 = mldatasets.tensor_to_img(
        saliency_maps[pos], to_numpy=True,\
        cmap_norm='positive', overlay_bg=bg_img
    )
    map2 = mldatasets.tensor_to_img(
        smooth_ig_maps[pos],to_numpy=True,\
        cmap_norm='positive', overlay_bg=bg_img
    )
    map3 = mldatasets.tensor_to_img(
        gradcam_maps[pos], to_numpy=True,\
        cmap_norm='positive', overlay_bg=bg_img
    )
    map4 = mldatasets.tensor_to_img(
        deeplift_maps[pos], to_numpy=True,\
        cmap_norm='positive', overlay_bg=bg_img
    )
    viz_img = cv2.vconcat([
        cv2.hconcat([map1, map2]),
        cv2.hconcat([map3, map4])
    ])
    label = int(y_test[idx])
    y_true = labels_l[label]
    y_pred = y_test_pred[idx]
    probs_s = probs_df.loc[idx]
    title = 'Gradient-Based Attr for Misclassification Sample #{}'.\
                                                    format(idx)
    mldatasets.compare_img_pred_viz(orig_img, viz_img, y_true,\
                                    y_pred, probs_s, title=title)
```

The preceding code generates *Figures 7.19* to *7.21*. It's important to note that in all generated plots, we can observe saliency attributions at the top left, SmoothGrad IG at the top right, guided Grad-CAM at the bottom left, and DeepLIFT at the bottom right:
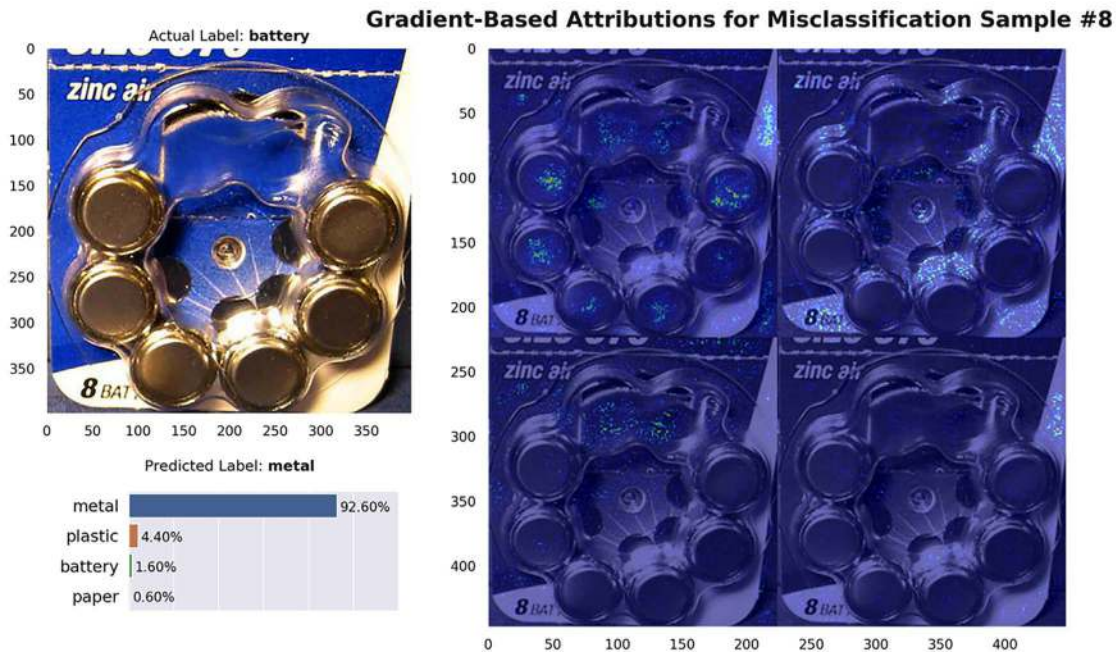
Figure 7.19: Gradient-based attributions for battery as metal misclassification #8

In *Figure 7.19*, there's a lack of consistency between all four attribution methods. The saliency attribution maps show that all the center parts of the batteries are seen as metal surfaces, in addition to the white parts of the cardboard container. On the other hand, SmoothGrad IG zeros in on the white cardboard and Grad-CAM on the blue cardboard almost exclusively. Lastly, DeepLIFT is much more sparse, only pointing to some parts of the white cardboard.

In *Figure 7.20*, the attributions are much more consistent than in *Figure 7.19*. Matte white areas are clearly confusing the model. This makes sense considering that the plastic in the training data was mostly single pieces of empty plastic containers – including white milk jugs. However, people do recycle toys, plastic tools like spatulas, and other plastic objects. Interestingly enough, although all attribution methods were salient around white and light-yellow surfaces, SmoothGrad IG also highlights some edges, like one of the ducks' hats and another one's collar:
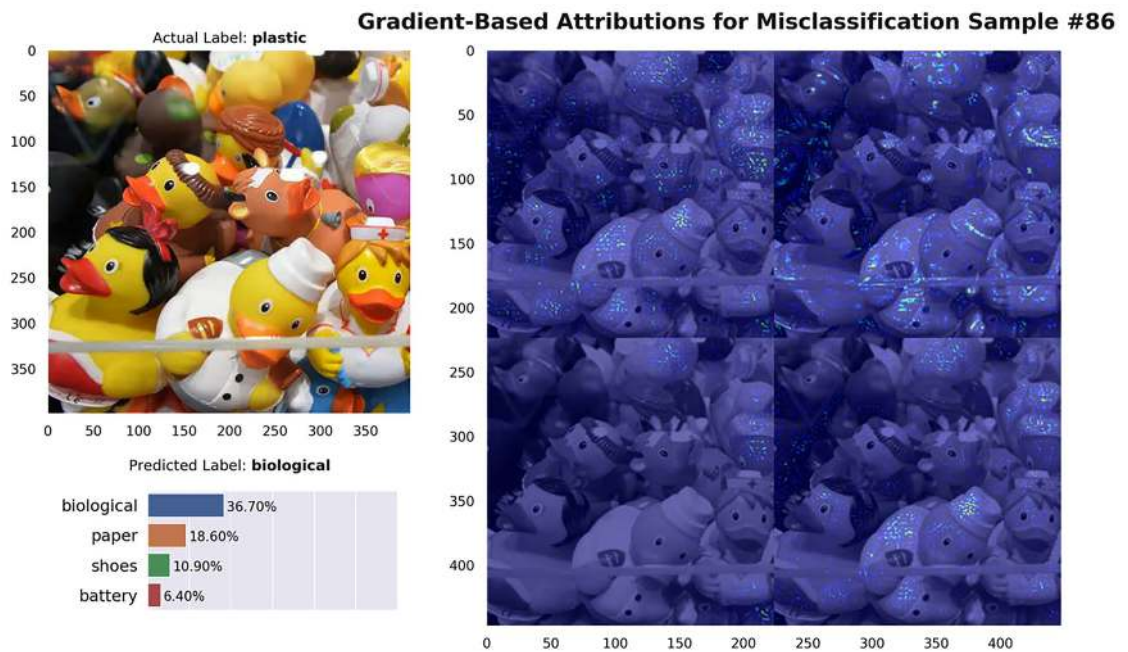
Figure 7.20: Gradient-based attributions for plastic misclassification #86

To continue with the recycling toys theme, how do LEGO bricks get misclassified as batteries? See *Figure 7.21* for an interpretation:
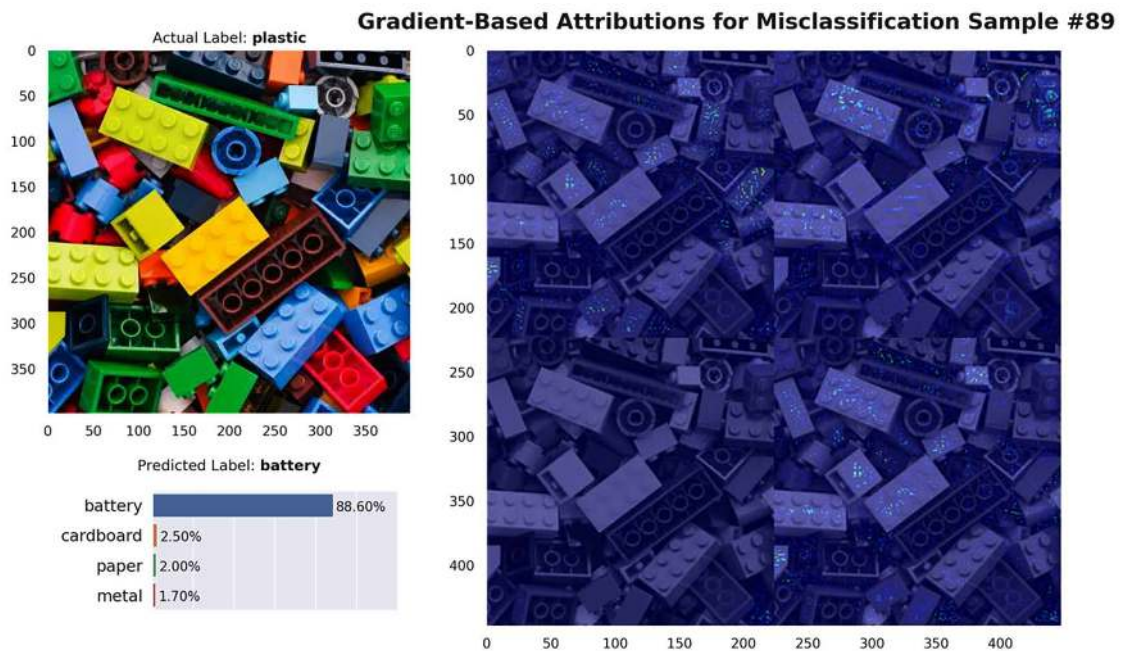
Figure 7.21: Gradient-based attributions for plastic misclassification #89

Figure 7.21 shows how, among all the attribution methods, it's mostly the yellow and green bricks (and to a lesser degree, light blue) that are to blame for the misclassification because these are popular colors among battery manufacturers, as attested by the training data. Also, the flat surface in between the studs got the most attributions since these resemble the contacts in batteries and, more specifically, 9-volt square batteries. As with the other examples, saliency was the most noisy method. However, this time, guided Grad-CAM was the least noisy. It was also more salient on the edges than on surfaces, unlike the others.

We will next try to discover what the model learned about batteries (in addition to white glass) through perturbation-based attribution methods performed on true positives.

Understanding classifications with perturbation-based attribution methods

Perturbation-based methods have already been covered to a great extent in this book so far. So many of the methods we have covered, including SHAP, LIME, anchors, and even permutation feature importance, employ perturbation-based strategies. The intuition behind them is that if you remove, alter, or mask features in your input data and then make predictions with them, you'll be able to attribute the difference between the new predictions and the original predictions to the changes you made in the input. These strategies can be leveraged in both global and local interpretation methods.

We will now do the same as we did with the misclassification samples, but to the chosen true positives, and gather four of each class in a single tensor (`X_correctcls`):

```
correctcls_idxs = wglass_TP_idxs[:4] + battery_TP_idxs[:4]
correctcls_data = torch.utils.data.Subset(test_data, correctcls_idxs)
correctcls_loader = torch.utils.data.DataLoader(correctcls_data,\
                                                batch_size = 32)
X_correctcls, y_correctcls = next(iter(correctcls_loader))
X_correctcls, y_correctcls = X_correctcls.to(device),\
                             y_correctcls.to(device)
```

One of the more complicated aspects of performing permutation methods on images is that there are not just a few dozen features but many thousands to permute. Picture this: 224 x 224 equals 50,176 pixels, and if we want to measure how a change in each pixel independently affects the outcome, we'll need to make at least 20 permuted samples for each pixel. So, over a million! For this reason, several permutation methods accept masks to determine which blocks of pixels to permute at once. If we group them in blocks of 32 x 32 pixels, this means we'll have only 49 blocks in total to permute. However, although it will speed up the attribution methods, we'll miss out on the effects on smaller sets of pixels the larger the block.

We can use many methods to create masks, such as using a segmentation algorithm to break up the images into intuitive blocks based on surfaces and edges. Segmentation is done per image, so the number and placement of segments will vary on an image-to-image basis. There are many methods with scikit-learn's image segmentation library (`skimage.segmentation`): `https://scikit-image.org/docs/stable/api/skimage.segmentation.html`. However, we are going to keep things simple and create one mask for all 224 x 224 images with the following code:

```python
feature_mask = torch.zeros(3, 224, 224).int().to(device)
counter = 0
strides = 16
for row in range(0, 224, strides):
    for col in range(0, 224, strides):
        feature_mask[:, row:row+strides, col:col+strides] = counter
        counter += 1
```

What the preceding code does is initialize a tensor of zeros the size of the model's input. It's easier to conceptualize this tensor as an empty image. Then it moves across strides that are 16 pixels wide and high, from the top-left corner of the image to the bottom right. As it moves across, it sets the values with consecutive numbers with the `counter`. What you end up with is a tensor with all values filled with numbers between 0 and 195, and, if you visualized it as an image, it would be a diagonal gradient from black at the top left to light gray at the bottom right. What's important to note is that each block with the same value is treated as if it were the same pixel by the attribution method.

Before we move forward, let's discuss baselines. In Captum attribution methods, as in other libraries for that matter, the default baseline is a tensor of zeros, which is usually equivalent to a black image when images are made up of floating-point numbers between 0 and 1. However, in our case, we are standardizing our input tensors so the model doesn't see tensors with a minimum of 0 but a mean of 0! Therefore, for our garbage model, a tensor of zeros corresponds to a medium gray image, not a black image. For gradient-based methods, there's nothing inherently wrong with a gray image baseline because there are likely a number of steps between it and the input image. However, perturbation-based methods can be particularly sensitive to having baselines that are too close to the input image because if you replace parts of the input image with the baseline, the model won't tell the difference!

For our garbage model's case, a black image is made up of tensors of `-2.1179` because one of the transformations performed to standardize the input tensors was `(x-0.485)/0.229`, which happens to equal approximately `-2.1179`, when `x=0`. You can also calculate the tensors when `x=1`; it converts to `2.64` for a white image. That being said, there's no harm in assuming that somewhere in our true positive samples, there's at least one pixel that has the lowest value and another with the highest, so we will just use `max()` and `min()` to create both light and dark baselines:

```python
baseline_light = float(X_correctcls.max().detach().cpu())
baseline_dark = float(X_correctcls.min().detach().cpu())
```

We will use only one baseline for all but one perturbation method but feel free to switch them around. Now, on to creating attribution maps for each method!
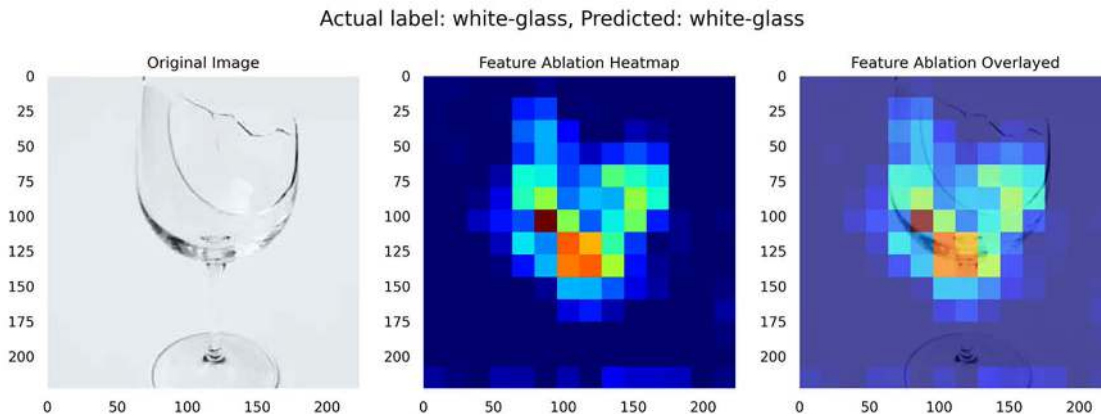
Feature ablation

Feature ablation is a relatively simple method. What it does is occlude portions of the sample input image by replacing it with the baseline, which is, by default, zero. The goal is to understand the importance of each input feature (or feature group) in making a prediction by observing the effect of altering it.

Here's how feature ablation works:

1. **Obtain the original prediction**: First, the model's prediction for the original input is obtained. This serves as a baseline for comparing the effect of perturbing the input features.
2. **Perturb the input feature**: Next, for each input feature (or feature group as set by the feature mask), it is replaced with the baselines value. This creates an "ablated" version of the input.
3. **Obtain the prediction for the perturbed input**: The model's prediction is calculated for the ablated input.
4. **Compute the attribution**: The difference in the model's predictions between the original input and the ablated input is calculated. This difference is attributed to the altered feature, indicating its importance in the prediction.

Feature ablation is a simple and intuitive approach to understanding the importance of input features in a model's prediction. However, it has some limitations. It assumes that features are independent and may not accurately capture the effects of interactions between features. Additionally, it can be computationally expensive for models with a large number of input features or complex input structures. Despite these limitations, feature ablation is a valuable tool for understanding and interpreting model behavior.

To generate the attribution maps, we will use the `get_attribution_maps` function as we have before, and enter the additional arguments for the `feature_mask` and `baselines`:

```
ablation_maps = get_attribution_maps(
    attr.FeatureAblation,garbage_mdl,\
    device,X_correctcls,y_correctcls,\
    feature_mask=feature_mask,\
    baselines=baseline_dark
)
```

To plot an example of the attribution map, you can copy the same code that we used for saliency, except `saliency_maps` is replaced by `ablation_maps`, and we are using the second image in the `occlusion_maps` array, like this:

```
pos = 2
orig_img = mldatasets.tensor_to_img(X_correctcls[pos], norm_std,\
                                    norm_mean, to_numpy=True)
attr_map = mldatasets.tensor_to_img(occlusion_maps[pos],to_numpy=True,\
                            cmap_norm='positive')
```

The modified snippet outputs *Figure 7.22*:

Figure 7.22: Feature ablation maps for a white glass true positive from the test dataset

In *Figure 7.22*, the feature groups in the bottom of the bowl of the wine glass appear to be most important because their absence makes the biggest difference in the outcome, but other portions of the glass are also salient to a lesser degree, except the stem of the wine glass. It makes sense because a wine glass without a stem is still a glass-like container.

Next, we will discuss a similar method that will be able to show us attributions with greater detail.

Occlusion sensitivity

Occlusion sensitivity is very similar to feature ablation because it also replaces portions of the image with a baseline. However, unlike feature ablation, it doesn't use a feature mask to group pixels together. Instead, it groups contiguous features automatically with a sliding window and strides. In this process, it creates many overlapping regions. When this happens, it averages the output differences to compute the attribution for each pixel.

In this scenario, besides overlapping regions and their corresponding averages, occlusion sensitivity and feature ablation are identical. In fact, if we used both sliding windows and strides of 3 x 16 x 16, there wouldn't be any overlapping areas and the feature grouping would be identical to those defined by our `feature_mask` made up of 16 x 16 blocks.

So, you may wonder, what's the point of being familiar with both methods? The point is occlusion sensitivity is only suitable for use when a fixed grouping of contiguous features matter, like with images and perhaps other spatial data. And because of its use of strides, it can capture local dependencies and spatial relationships between features. However, although we used contiguous blocks of features, feature ablation doesn't have to because the `feature_mask` can be arranged in whichever way it makes most sense for your inputs to be segmented. This small detail makes it very versatile to other data types. Therefore, feature ablation is a more general approach that can handle various input types and model architectures, while occlusion sensitivity is specifically tailored to image data and convolutional neural networks, with a focus on spatial relationships between features.

To generate the attribution maps for occlusion, we will do as before, and enter the additional arguments for the `baselines`, `sliding_window_shapes`, and `strides`:

```
occlusion_maps = get_attribution_maps(
    attr.Occlusion, garbage_mdl,\
    device,X_correctcls,y_correctcls,\
    baselines=baseline_dark,\
    sliding_window_shapes=(3,16,16),\
    strides=(3,8,8)
)
```

Please note that we are creating ample overlapping regions by setting the strides to be only 8 pixels while the sliding windows are 16 pixels. To plot an attribution, you can copy the same code that we used for feature ablation, except `ablation_maps` is replaced by `occlusion_maps`. The output is depicted in *Figure 7.23*:
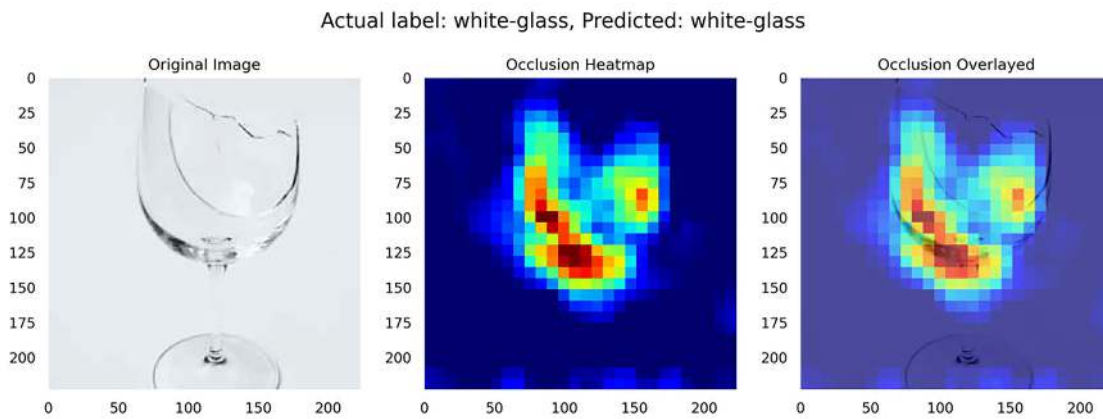
Figure 7.23: Occlusion sensitivity maps for a white glass true positive from the test dataset

With *Figure 7.23*, we can tell that occlusion's attributions are eerily similar to the ablation's attribution, except with more resolution. This resemblance shouldn't be surprising considering how the feature mask of the former aligns with the sliding window of the latter.

Whether we use blocks of non-overlapping 16 x 16 pixels or overlapping 8 x 8, the impact of their absence is measured independently to create the attributions. Therefore, both ablation and occlusion methods aren't equipped to measure interactions between non-contiguous feature groups. This can prove to be a problem when the absence of two non-contiguous feature groups is what causes a classification to change. For instance, can a wine glass without a stem or a base still be considered a wine glass? It can certainly be considered glass, one would hope, but perhaps the model has learned the wrong relationships.

Speaking of relationships, next, we will revisit an old friend: Shapley!

Shapley value sampling

If you recall from *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, Shapley has provided a method that is very good at measuring and attributing the impact of coalitions of features to the outcome. Shapley does this by permuting entire coalitions of features at a time rather than permuting one feature at a time, like the two previous methods. That way, it can tease out how more than one feature or feature group interacts with one another.

The code to create the attribution maps should be very familiar by now. This method uses the `feature_mask` and `baselines` but also the number of feature permutations tested (`n_samples`). This last attribute has a huge impact on the fidelity of the method. However, it can make it notoriously computationally expensive, so we are not going to run it with the default 25 samples per permutation. Instead, we will use 5 samples to make things more manageable. However, feel free to tweak it should your hardware be able to handle it:

```
svs_maps = get_attribution_maps(
    attr.ShapleyValueSampling,garbage_mdl,\
    device, X_correctcls, y_correctcls,\
    baselines=baseline_dark,\
    n_samples=5, feature_mask=feature_mask
)
```

To plot an attribution, you can copy the same snippet that we used for occlusion, except `occlusion_maps` is replaced by `svs_maps`. The output is shown in *Figure 7.24*:
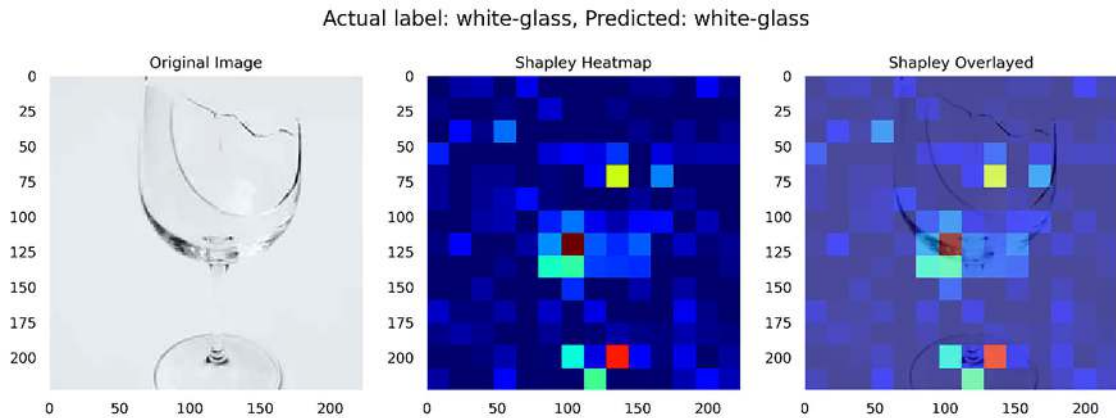
Figure 7.24: Shapley value sampling maps for a white glass true positive from the test dataset

Figure 7.24 shows some consistent attributions, such as the most salient area being in the bottom-left corner of the wine glass bowl. Also, the base seems to be more important than the occlusion and ablation methods suggested.

However, the attributions are a lot more noisy than the previous ones. This is partially because we didn't use a sufficient number of samples to cover all the combinations of features and interactions, and partially because of the messy nature of interactions. It makes sense that attributions for a single independent feature are concentrated in a few areas, such as the bowl of a wine glass. However, interactions can rely on several parts of the image, such as the base and the rim of the wine glass. They may become important only when they appear together. More interesting is the effect of a background. For instance, if you remove portions of the background, does the wine glass no longer look like a wine glass? Perhaps the background is more important than you think, especially when dealing with a translucent material.

KernelSHAP

Since we are on the topic of Shapley values, let's try `KernelSHAP` from *Chapter 4*, *Global Model-Agnostic Interpretation Methods*. It leverages LIME to compute Shapley values more efficiently. The Captum implementation is similar to the SHAP one except it uses linear regression and not Lasso, and it computes the kernel differently. Also, for the LIME image explainer, it is best to use meaningful feature groups (called superpixels) rather than the contiguous blocks we have used in the feature mask. The same advice persists for `KernelSHAP`. However, we will keep this simple for this exercise, and also consistent for comparing with the other three permutation-based methods.

We will now create the attribution maps but, this time, we will do one with light baselines and another with dark ones. Because `KernelSHAP` is an approximation to Shapley sampling values and not as computationally expensive, we can set `n_samples=300`. However, this won't necessarily guarantee high fidelity because it takes a high amount of samples in `KernelSHAP` to approximate what a relatively low amount of samples can do exhaustively with Shapley:

```python
kshap_light_maps = get_attribution_maps(attr.KernelShap, garbage_mdl,\
                                        device, X_correctcls, y_correctcls,\
                                        baselines=baseline_light,\
                                        n_samples=300,\
                                        feature_mask=feature_mask)
kshap_dark_maps = get_attribution_maps(attr.KernelShap, garbage_mdl,\
                                       device, X_correctcls, y_correctcls,\
                                       baselines=baseline_dark,\
                                       n_samples=300,\
                                       feature_mask=feature_mask)
```

To plot an attribution, you can copy the same snippet that we used for Shapley, except `svs_maps` is replaced by `kshap_light_maps`, and we modify the code to plot the attributions with the dark baselines in the third position, like this:

```python
axs[2].imshow(attr_dark_map)
axs[2].grid(None)
axs[2].set_title("Kernel Shap Dark Baseline Heatmap")
```

The preceding snippet outputs the plots in *Figure 7.25*:

Figure 7.25: KernelSHAP maps for a white glass true positive from the test dataset

The two attribution maps in *Figure 7.25* are mostly not consistent with each other, but more importantly, not with the previous attributions. Sometimes, some methods have a harder time than others, or it takes some tweaking to get them to work how we expect them to.

Tying it all together

Now, we will take everything that we have learned about perturbation-based attribution methods and use it to understand the reasons for all the chosen true positive classifications (for both white glass and batteries). As we did before, we can leverage the `compare_img_pred_viz` function to place the higher-resolution sample image side by side with the four attribution maps: feature ablation, occlusion sensitivity, Shapley, and `KernelSHAP`. First, we have to iterate all the classifications' positions and indexes and extract all the maps. Note that we are using `overlay_bg` to produce a new image overlaying the original image with the heatmap for every attribution, as we did for the gradient-based section. Lastly, we concatenate the four attribution outputs into a single image (`viz_img`). Just as we have done before, we extract the actual label (`y_true`), predicted label (`y_pred`), and `pandas` series with the probabilities (`probs_s`) to add some context to the plot we will produce. The `for` loop will produce six plots, but we will only discuss two of them:

```
for pos, idx in enumerate(correctcls_idxs):
    orig_img = mldatasets.tensor_to_img(test_400_data[idx][0],\
                                        norm_std, norm_mean,\
                                        to_numpy=True)
    bg_img = mldatasets.tensor_to_img(test_data[idx][0],\
                                      norm_std, norm_mean,\
                                      to_numpy=True)
    map1 = mldatasets.tensor_to_img(ablation_maps[pos],\
                                    to_numpy=True,\
                                    cmap_norm='positive',\
                                    overlay_bg=bg_img)
```

```
      map2 = mldatasets.tensor_to_img(svs_maps[pos], to_numpy=True,\
                                        cmap_norm='positive',\
                                        overlay_bg=bg_img)
      map3 = mldatasets.tensor_to_img(occlusion_maps[pos],\
                                        to_numpy=True,\
                                        cmap_norm='positive',\
                                        overlay_bg=bg_img)
      map4 = mldatasets.tensor_to_img(kshap_dark_maps[pos],\
                                        to_numpy=True,\
                                        cmap_norm='positive',\
                                        overlay_bg=bg_img)
  viz_img = cv2.vconcat([
          cv2.hconcat([map1, map2]),
          cv2.hconcat([map3, map4])
      ])
  label = int(y_test[idx])
  y_true = labels_l[label]
  y_pred = y_test_pred[idx]
  probs_s = probs_df.loc[idx]
  title = 'Pertubation-Based Attr for Correct classification #{}'.\
              format(idx)
  mldatasets.compare_img_pred_viz(orig_img, viz_img, y_true,\
                                    y_pred, probs_s, title=title)
```

The preceding code snippet generates several explanations, including *Figures 7.26* to *7.28*. For your reference, ablation is in the top-left corner and occlusion is at the bottom left. Then, Shapley is at the top right and `KernelSHAP` is at the bottom right.

Overall, you can tell that ablation and occlusion are very consistent, while much less so with Shapley and `KernelSHAP`. However, what Shapley and `KernelSHAP` have in common is that the attributions are more spread out.

In *Figure 7.26*, all attribution methods have text highlighted, as well as, at least, the left contact of the battery. This is similar to *Figure 7.28*, where the text is abundantly highlighted as well as the top contact. This suggests that, for batteries, the model has learned that text and a contact matter. As for white glass, it is less clear. All the attribution methods in *Figure 7.27* point to some of the edges of the broken vase, but not always the same edges (except for ablation and occlusion, which are consistent):
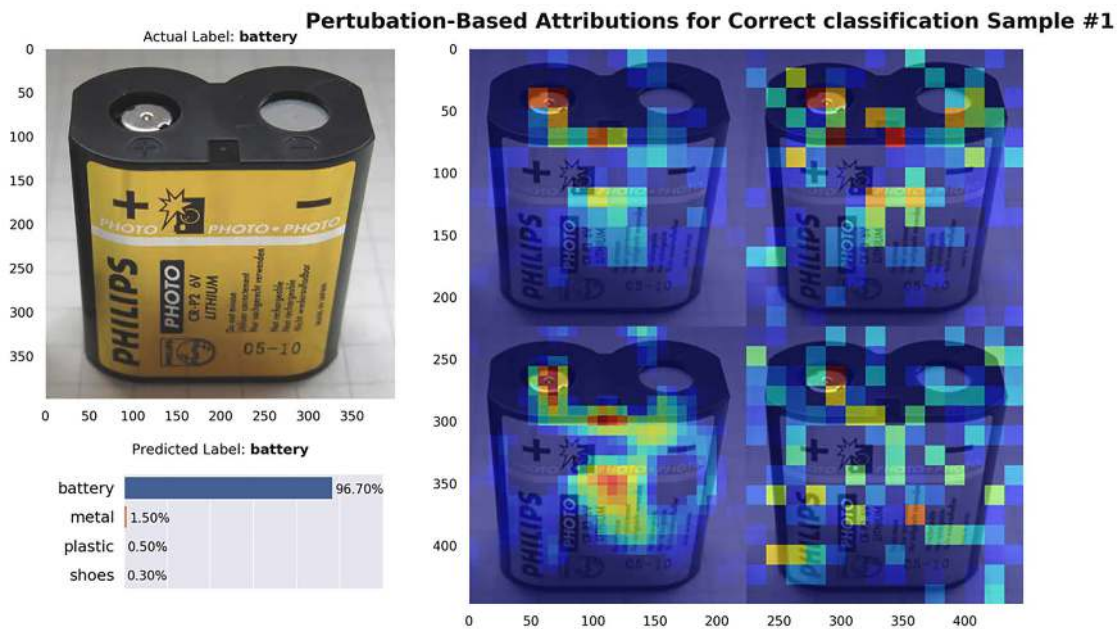
Figure 7.26: Perturbation-based attributions for battery classification #1

White glass is the hardest glass to classify of the three, and it's not hard to tell why:
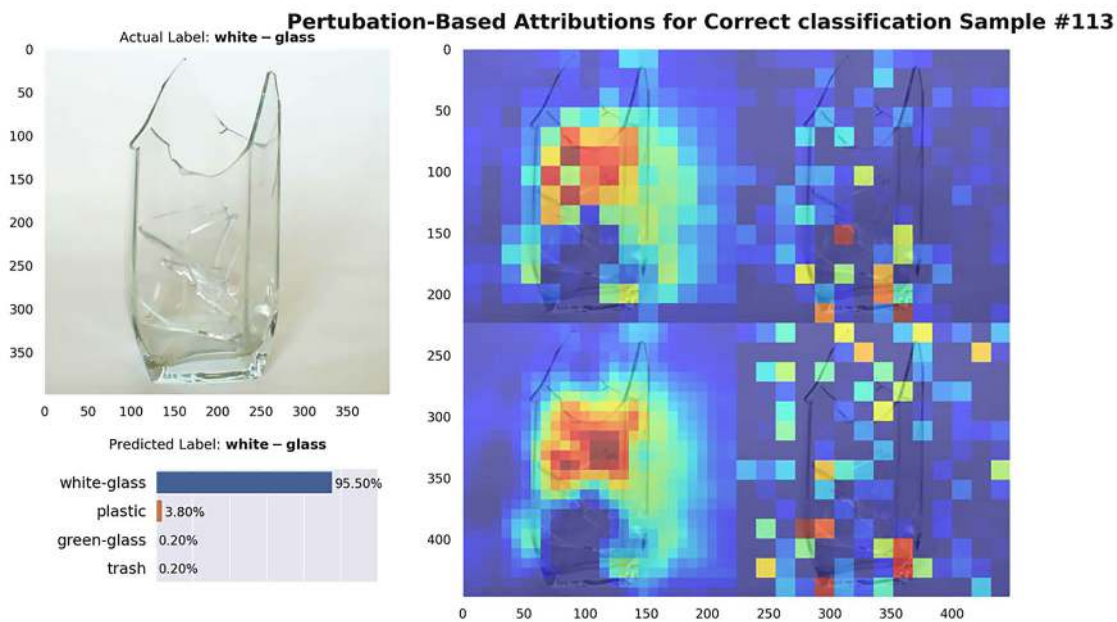
Figure 7.27: Perturbation-based attributions for white glass classification #113

As noted in *Figure 7.27*, and others in the test example, it's hard for the model to distinguish white glass from the light backgrounds. It manages to classify it correctly with these examples. However, this doesn't mean it will generalize well in other examples where glass is in shards and not as well-lit. As long as the attributions show significant influence from the background, it's hard to trust that it can recognize glass for its specular highlights, texture, and edges alone.
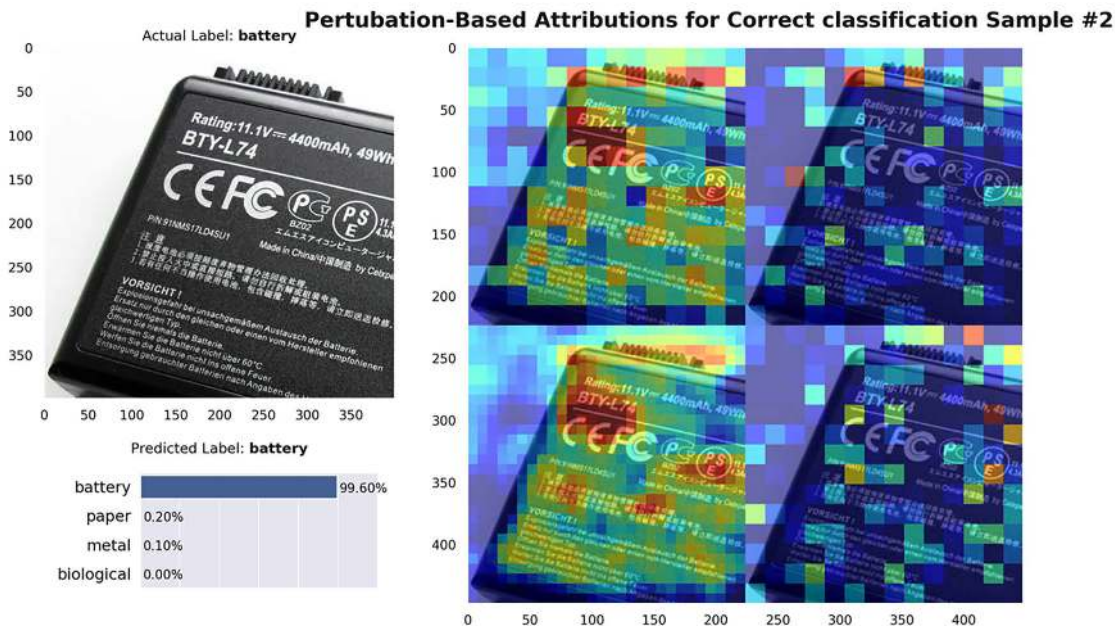
Figure 7.28: Perturbation-based attributions for battery classification #2

For *Figure 7.28*, the background is also highlighted significantly in all attribution maps. But perhaps this is because the baseline was dark and so is the object in its entirety. If you replace an area just outside the edge of the battery with a black square, it makes sense that the model would be confused. For this reason, with permutation-based methods, it's important to choose an appropriate baseline.

Mission accomplished

The mission was to provide an objective evaluation of the garbage classification model for the municipal recycling plant. The predictive performance on out-of-sample validation images was dismal! You could have stopped there, but then you would not have known how to make a better model.

However, the predictive performance evaluation was instrumental in deriving specific misclassifications, as well as correct classifications, to assess using other interpretation methods. To this end, you ran a comprehensive suite of interpretation methods, including activation, gradient, perturbation, and backpropagation-based methods. The consensus between all the methods was that the model was having the following issues:

- Differentiating between the background and the objects
- Understanding that different objects share similar color hues

- Confounding lighting conditions, such as specular highlights as specific material characteristics, like with the wine glasses
- An inability to separate unique features of each object, such as plastic studs in LEGO bricks from battery contacts
- Being confused by objects with multiple materials, such as batteries contained in plastic and even cardboard packaging

To address these problems, the model needed to be trained with a more varied dataset – hopefully, one that reflects the real-world conditions of the recycling plant; for instance, the expected background (on a conveyor belt), different lighting conditions, and even objects partially occluded by hands, gloves, bags, and so on. Also, they ought to add a category for miscellaneous objects that are made up of multiple materials.

Once this dataset has been compiled, it is essential to leverage data augmentation to make the model even more robust to all sorts of variations: angle, brightness, contrast, saturation, and hue variants. And they won't have to retrain the model from scratch! They can even fine-tune EfficientNet!

Summary

After reading this chapter, you should understand how to leverage traditional interpretation methods to more thoroughly assess predictive performance on a CNN classifier and visualize the learning process of CNNs with activation-based methods. You should also understand how to compare and contrast misclassifications and true positives with gradient-based and perturbation-based attribution methods. In the next chapter, we will study interpretation methods for NLP transformers.

Further reading

- Smilkov, D., Thorat, N., Kim, B., Viégas, F., and Wattenberg, M., 2017, *SmoothGrad: Removing noise by adding noise*. ArXiv, abs/1706.03825: `https://arxiv.org/abs/1706.03825`
- Sundararajan, M., Taly, A., and Yan, Q., 2017, *Axiomatic Attribution for Deep Networks*. Proceedings of Machine Learning Research, pp. 3319–3328, International Convention Centre, Sydney, Australia: `https://arxiv.org/abs/1703.01365`
- Zeiler, M.D., and Fergus, R., 2014, *Visualizing and Understanding Convolutional Networks*. In European conference on computer vision, pp. 818–833: `https://arxiv.org/abs/1311.2901`
- Shrikumar, A., Greenside, P., and Kundaje, A., 2017, *Learning Important Features Through Propagating Activation Differences*: `https://arxiv.org/abs/1704.02685`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

8

Interpreting NLP Transformers

In the last chapter, we learned about applying explanation methods to a specific type of deep learning model architecture, convolutional neural networks. In this chapter, we will provide some tools to do the same with the transformer model architecture. Transformer models are becoming increasingly popular, and their most common use case is **Natural Language Processing** (**NLP**). We broached the subject of NLP in *Chapter 5*, *Local Model-Agnostic Interpretation Methods*. In this chapter, we will do so too but with transformer-specific methods and tools. First, we will discuss how to visualize attention mechanisms, followed by interpreting integrated gradient attributions, and lastly, exploring the Swiss Army knife that is the **Learning Interpretability Tool** (**LIT**).

These are the main topics we will cover:

- Visualizing attention with BertViz
- Interpreting token attributions with integrated gradients
- LIME, counterfactuals, and other possibilities with the LIT

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy` , `torch`, `transformers` , `bertviz`, `captum`, and `lit-nlp` libraries. Instructions on how to install all these libraries are in the *Preface*.

> The code for this chapter is located here: `https://packt.link/Yzf2L`

The mission

You are a data scientist working for a yet-to-launch startup in New York City. This startup aims to establish itself as the go-to place to find the best, newest, and most exciting culinary destinations in the city!

The aim is to move beyond the typical structured data about restaurants and delve deep into the vast array of textual data available online, from social media sites to directory websites. The startup believes that while ratings might provide a simplistic quantification of experiences, reviews contain richer details and can offer multidimensional insights into what makes a restaurant special.

Reviews express detailed sentiments that capture diverse user experiences, unlike ratings, which provide a singular, non-comparative perspective. By harnessing the granularity present in reviews, the startup can tailor its recommendations to cater to various audience segments with greater precision.

Your team has been discussing how to leverage sentiment analysis on reviews to determine how to best look for the feelings that exemplify the experience users look for in the recommender system. Binary sentiment analysis (positive/negative) does not offer the nuances required to distinguish between usual and unique experiences, or those catering to specific groups such as travelers, families, or couples. Also, the startup founders believe that the dining experience is multifaceted. An experience that might be seen as "positive" could range from "comforting and nostalgic" to "thrilling and adventurous." Distinguishing these nuances will empower the recommender system to be more personalized and effective.

Your manager encountered a sentiment classification model that has 27 categories trained with a dataset called GoEmotions, published by Google. GoEmotions offers a more detailed classification of sentiments, capturing the richness of human emotions more effectively than binary classification models. However, the lead strategist decided there were too many classifications and decided to group them into a different emotion taxonomy called Ekman (see *Figure 8.1*):
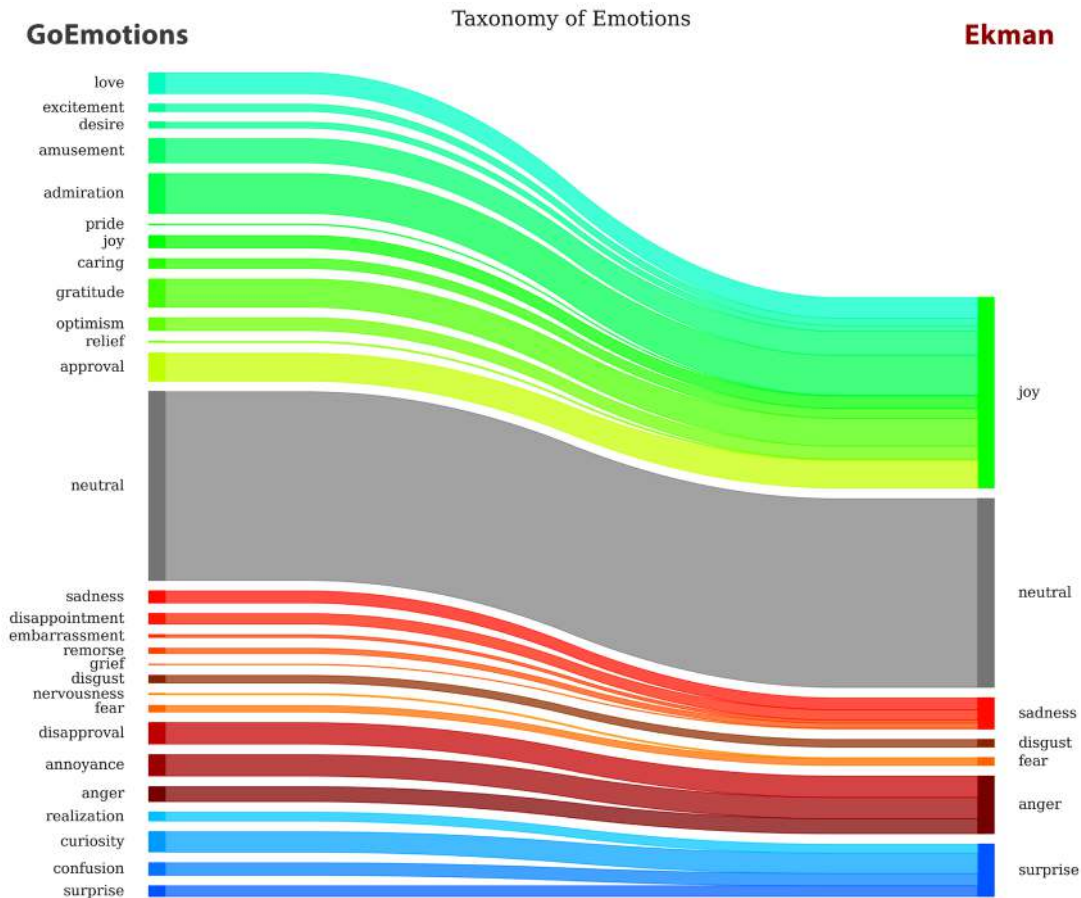
Figure 8.1: The Taxonomy of Emotions

Ekman's taxonomy of emotions is a classification system developed by psychologist Paul Ekman, which identifies six basic emotions that he believed to be universally experienced in all human cultures. These six emotions are joy, sadness, fear, anger, disgust, and surprise. Ekman suggested that these are fundamental emotions that are hard-wired into our brains and expressed in the same way by people all around the world, regardless of their culture. These emotions can be identified through specific facial expressions, and understanding them can help in fields like psychology, communication, and sociology. Ekman's taxonomy offers a more concise set of emotion categories while still preserving the nuances. This makes the interpretation more manageable and actionable for the development team and other stakeholders. However, we've had to keep neutral, which cannot be classified into any Ekman category.

Now, the next step is to interpret the GoEmotions Ekman classifier model with the Tripadvisor re-view dataset to understand what the model learned and uncover patterns that could be useful to the development of the recommender system. It's an open-ended task. The general goal is to understand the patterns the model has identified in the reviews and how these correlate with Ekman's categories. However, this path could lead to many findings or a dead-end. Leadership stressed that data scientists, like yourself, would have to use their judgment to find opportunities in data exploration and model interpretation.

By uncovering these patterns, the startup can fine-tune its algorithm to look for reviews that resonate with these emotions. The insights can also guide restaurant partnerships, marketing strategies, and feature enhancements to the platform.

The approach

You have decided to take a three-prong approach:

1. You will look under the hood of the transformer model to visualize attention weights with BertViz to find relevant patterns in those mechanisms.
2. Then, you'll produce saliency maps where attributions are color-coded for each token in re-views of interest, using the integrated gradients method.
3. Lastly, you'll examine counterfactuals with the LIT.

You hope that you can deliver some actionable insights to the leadership team with these steps.

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/tree/master/08/ReviewSentiment.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `torch` (PyTorch) and `transformers` to load and configure the model

- bertviz, captum, and lit-nlp to generate and visualize the model interpretations

You should load all of them first:

```
import math
import os, random, re, gc
import warnings
warnings.filterwarnings("ignore")
import mldatasets
import numpy as np
import pandas as pd

import torch
from transformers import AutoTokenizer,\
                          AutoModelForSequenceClassification, pipeline
from bertviz import head_view, model_view
from captum.attr import LayerIntegratedGradients,\
                        TokenReferenceBase, visualization
from lit_nlp import notebook
from lit_nlp.api import dataset as lit_dataset
from lit_nlp.api import model as lit_model
from lit_nlp.api import types as lit_types
```

Next, we work on data understanding and preparations.

Understanding and preparing the data

We load the data like this into a DataFrame we call `reviews_df`:

```
reviews_df = mldatasets.load("nyc-reviews", prepare=True)
```

There should be over 380,000 records and 12 columns. We can verify this is the case with `info()`:

```
reviews_df.info()
```

The output checks out. There are no missing values. However, there are only three numeric features, one date, and all the rest are object data types because they are mostly text. Given that this chapter focuses on NLP, this shouldn't come as a surprise. Let's examine the data dictionary to understand what we will use from this DataFrame.

The data dictionary

These are the 12 columns in the DataFrame, most of which are there for reference:

- `review_id`: ID – a unique identifier for the review (only for reference)
- `author_id`: ID – a unique identifier for the author (only for reference)
- `restaurant_name`: text – the name of the restaurant (only for reference)

- url_restaurant: **URL – Uniform Resource Identifier** to locate the web page where the restaurant review is located (only for reference)
- review_date: date – the date when the review was made (only for reference)
- review_title: text – the title the author wrote for the review
- review_preview: text – the preview generated for the review
- review_full: text – the full review written by the author
- rating: ordinal – a rating given by the author to the establishment (a 1–5 rating scale)
- positive_sentiment: binary – whether the review has a positive sentiment according to a binary sentiment model (positive/negative)
- label: categorical – the predicted emotion by the GoEmotions classifier (according to the Ekman seven-class classification: joy, neutral, sadness, disgust, fear, anger, and surprise)
- score: continuous – the predicted probability that the review belongs to the predicted class

It's a multi-class model, so it predicted scores for each class. However, we only stored the score of the most probable class (label). Therefore, the last two columns represent the output of the model. As for the input, let's examine the first three rows to illustrate it:

```
reviews_df[["review_title","review_full","label","score"]].head(3)
```

The preceding snippet will generate the output in *Figure 8.2*:

review_title	review_full	label	score
Good neighborhood spot!	Came with family for Labor Day weekend brunch as my daughter lives nearby and it's always been picked on previous visits. Had nice shaded and socially..	joy	0.987768
Disappointing	Food was mediocre at best. The lamb chops are an image they feature on the websites opening page. It wasn't even listed on the menu. When I asked I wa..	sadness	0.504617
What a find in Harlem	My co-workers were volunteering at a foodbank around the corner and we came here for lunch. What a find. Awesome Italian food with unique twists, not ..	joy	0.999603

Figure 8.2: The first three reviews of the dataset

It's the first two columns in *Figure 8.2*, review_title and review_full, that represent the input for the model. It does so as a single piece of text, so when we discuss *the review*, we are referring to a string that concatenates both with a colon and space, separating them like this: *Disappointing: Food was mediocre at best. The lamb chops are an image they feature on the websites opening page.*

But these are not the only columns that could matter in an analysis. We could, of course, analyze reviews by author, restaurant, date, and so on, and even connect restaurants to specific coordinates on a map to understand how sentiment varies geographically. This is all very interesting. However, we won't go into any detail here because although it might be relevant to the general mission of sentiment analysis, it will divert from the technical topic of this chapter, which is interpreting transformer models.

Nonetheless, we will explore some features that are highly correlated with the model outcomes, which are the `rating` provided by the author and the outcome of the binary sentiment analysis model (`positive_sentiment`). You would definitely expect these to match because reviews are generally consistent with the ratings — that is, a positive review will have a higher rating than a negative one. Likewise, some emotions are more positive than negative.

To understand these correlations a bit better, let's aggregate the reviews to get an average `rating` and `positive_sentiment` for each emotion like this:

```
sum_cols_l = ["score","positive_sentiment","rating"]
summary_df = reviews_df.groupby("label")[sum_cols_l].agg(
    {"score":["count","mean"], "positive_sentiment":"mean", "rating":"mean"}
)
summary_df.columns = ["count", "avg. score", "% positive", "avg. rating"]
summary_df.sort_values(by="avg. rating", ascending=False).style.format(
    {
        "count":"{:,}",
        "avg. score":"{:.1%}",
        "% positive":"{:.1%}" ,
        "avg. rating":"{:.2f}"
    }
).bar(subset=["avg. score", "% positive", "avg. rating"],\
            color="#4EF", width=60)
```

The above code will produce the output in *Figure 8.3*:

label	count	avg. score	% positive	avg. rating
joy	344,982	97.1%	91.3%	4.46
surprise	10,263	73.8%	65.1%	3.74
neutral	12,305	67.5%	36.5%	3.08
sadness	9,956	81.4%	12.4%	2.52
anger	1,398	56.6%	4.2%	1.99
fear	621	59.3%	15.5%	1.90
disgust	932	68.2%	0.3%	1.37

Figure 8.3: A summary table of the emotions predicted for the review dataset

As you can appreciate in *Figure 8.3*, the majority of the 380,000 reviews are placed in the joy label or class. Joy is a positive emotion, so it makes sense that our binary sentiment classifier classified over 90% of them as positive and that the average rating for joyous reviews is nearly 4.5. The DataFrame is sorted by the average rating because it's not a product of a model (which could be wrong), so it can perhaps give us the clearest indication of what predicted emotions the end users perceive to be the most positive. And as you go down in the list, you have positive emotions first, followed by neutral, and then the negative ones. Please note that the percentage of reviews that was deemed to be positive by the binary classifier is somewhat consistent with the same order provided by the average rating.

On the other hand, the average score for each label tells us how confident the predictions are on average that it belongs to said label. Joy, sadness, and surprise are the most confident. Since the multi-class predictions are seven numbers that add up to 1, an average score of 56.6% for anger is an indication that many predictions in which anger is the most probable emotion will have other emotions with a sizable probability – perhaps even emotions that may seem incompatible. We will put a pin on this because it would be interesting to explore this later.

Another fascinating interpretation you can make of *Figure 8.3* is that a large proportion of surprise is negative, despite it being supposedly perceived to be a positive emotion. Also, with an average rating lower than 4, there are probably quite a few negative ratings weighing them down. We won't explore the data in this chapter, but indeed, there are plenty of negative reviews that embody a sentiment of surprise. In light of this finding and for the sake of adapting to the mission, let's say you presented this to your bosses, and they decided it made sense to focus on surprise because market research shows that people love to find and be surprised by "hidden gems." Therefore, it's critical that a recommendation engine can help unearth any restaurants that are positively surprising, while suppressing any that are consistently negatively surprising.

Loading the model

Later on, we will be randomly selecting from our dataset, so in order to do that consistently, it's best to set a random seed. It's always good practice to initialize the seed in all the pertinent libraries, even though in this case it won't make a difference for PyTorch inference operations:

```
rand = 42
os.environ["PYTHONHASHSEED"]=str(rand)
random.seed(rand)
np.random.seed(rand)
torch.manual_seed(rand)
```

Next, let's define a `device` variable because if you have a CUDA-enabled GPU, model inference will perform quicker. Then, we will load the tokenizer (`goemotions_tok`) and model (`goemotions_mdl`) from Hugging Face using the `from_pretrained` function. Lastly, we will move all the weights and biases to your device with the `model.to(device)` function and set the model to evaluation mode with `model.eval()`:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
goemotions_mdl_path = "monologg/bert-base-cased-goemotions-ekman"
```

```
goemotions_tok = AutoTokenizer.from_pretrained(goemotions_mdl_path)
goemotions_mdl = AutoModelForSequenceClassification.from_pretrained(
    goemotions_mdl_path, output_attentions=True
)
goemotions_mdl.to(device)
goemotions_mdl.eval()
```

Once the model is loaded, we always inspect its architecture with `print(goemotions_mdl)`. However, architecturally, in broad terms, what may matter most when interpreting transformer models is how many layers and attention heads they have. We can inspect that easily with the following snippet:

```
num_layers = goemotions_mdl.config.num_hidden_layers
num_attention_heads = goemotions_mdl.config.num_attention_heads
print(f"The model has {num_layers} layers.")
print(f"Each layer has {num_attention_heads} attention heads.")
```

It should say that there are 12 layers and 12 attention heads. In the next section, we will dive into understanding how the attention mechanism works, and how to visualize the layers and heads with BertViz.

Visualizing attention with BertViz

What is attention? Let's imagine you're reading a book and come across a sentence that mentions a character you've read about earlier, but you've forgotten some details about them. Instead of going back and reading everything from the start, you'd likely skim through previous pages, focusing specifically on the parts that talk about this character. Your mind gives "attention" to the relevant information while filtering out the less relevant parts.

The attention mechanism in models like transformers works in a similar way. When processing information, it doesn't treat all pieces of data equally. Instead, it "pays attention" to the most relevant parts, giving them more importance in the context of the current task. This ability to selectively focus on specific parts helps the model understand complex patterns and relationships in the data.

Transformers are made up of two main components: the encoder and the decoder. Each component leverages attention mechanisms, but they do so differently:

- **Encoder:** The encoder's job is to understand the input data. It does this by using an attention mechanism to figure out how each part of the input (like a word in a sentence) relates to all other parts. This allows the encoder to create a rich representation of the input that captures the relationships and context within it. It's like reading a sentence and understanding what each word means in the context of that sentence.

- **Decoder:** Once the encoder has created this representation, the decoder uses it to produce the output. The decoder also uses an attention mechanism, but it uses it in two ways. First, it pays attention to the encoder's representation to understand what the input was. Second, it pays attention to its own previous outputs to make sure the current output is consistent with what it has produced so far. It's like writing a sentence that makes sense based on what you read and what you've already written.

However, not all transformer models have both components. In essence, the use case determines which parts of the transformer are needed:

- **Encoder models** (like BERT): These models only use the encoder component of the transformer. They are typically used for tasks that involve understanding the input data, like sentiment analysis (determining whether a text is positive or negative), named entity recognition (identifying people, organizations, and locations in text), or other classification tasks. This is because the encoder's job is to create a representation of the input data that captures the relationships and context within it.

- **Decoder models** (like GPT, LLaMa, etc.): These are used for tasks that involve generating new data, like text generation. The decoder part of the transformer ensures the generated output is consistent with what has been produced so far.

- **Encoder-decoder models** (like FLAN): These are used for tasks that involve transforming one piece of data into another, like translation. The encoder understands the input, and the decoder generates the output.

Now that we have covered attention models, let's dive into BERT.

BERT, which stands for **Bidirectional Encoder Representations from Transformers**, is a type of transformer model developed by Google. It's used to understand and analyze text data in a wide variety of languages. BERT is a transformer model that reads text bidirectionally to understand the context of words better. It only uses the encoder part of the transformer because its job is to understand text, not generate it. This makes BERT very effective for a wide range of tasks that involve understanding text.

So, our BERT transformer model has 12 layers and 12 attention heads. But what do these do, and how do they work?

- **BERT layers:** The 12 hidden layers are BERT layers, and these are a stack of other layers that make up this encoder transformer. Much like with convolutional layers in a CNN, like the one we examined in *Chapter 7, Visualizing Convolutional Neural Networks*, BERT layers represent layers of abstraction. As the input data progresses through layers, the model learns increasingly abstract representations of the data. In the context of text, the lower layers might capture basic syntactic information, like the role of a word in a sentence. As you move up through the layers, they tend to capture higher-level semantics, such as overall sentence meaning or themes. The number of layers, called the depth of the model, often correlates with its ability to understand context and represent complex relationships. However, more layers also require more computational resources and might be more prone to overfitting if not enough data is available.

- **Attention head:** The self-attention mechanism is the heart of any transformer model. The attention head has several self-attention mechanisms working in parallel. Inside the **multi-head self-attention mechanism**, there are multiple independent attention heads working in parallel. Each attention head learns to focus on different parts of the input data (like different relationships between tokens, which are usually the words). Having multiple attention heads allows the model to capture various types of relationships simultaneously. For example, one head might focus on the relationship between adjectives and nouns, while another might capture verb-subject relationships. After each head computes its own attention-weighted value representation, the outputs from all heads are concatenated and linearly transformed to produce the final value representation for the next layer.

Let's use some real reviews to examine the inner workings of the GoEmotions model. To that end, we will take four sample reviews and print out their details using the following code. While we are at it, we will save the sample reviews in a dictionary (`sample_reviews_dict`) so we can reference them later:

```python
surprise_sample_reviews_l = [174067, 284154, 480395, 47659]
line_pattern = r"(?<=[.!?])\s+"
sample_reviews_dict = {}
for i, review_idx in enumerate(surprise_sample_reviews_l):
    review_s = reviews_df.loc[review_idx, :]
    sentiment = "Positive" if review_s["positive_sentiment"]\
                          else "Negative"
    review_lines_l = re.split(
        line_pattern, review_s["review_full"], maxsplit=1
    )
    review_txt = "\r\n\t\t".join(review_lines_l)

    print(f"{review_s["restaurant_name"]}")
    print(f"\tSentiment:\t\t{sentiment}")
    print(f"\tRating:\t\t\t{review_s["rating"]}")
    print(f"\tGoEmotions Label:\t{review_s["label"]}")
    print(f"\tGoEmotions Score:\t{review_s["score"]:.1%}")
    print(f"\tTitle:\t{review_s["review_title"]}")
    print(f"\tReview:\t {review_txt}")
    sample_reviews_dict[i] = review_lines_l
```

The preceding snippet should output the text in *Figure 8.4*:

```
2nd_Avenue_Deli
        Sentiment:              Positive
        Rating:                 4
        GoEmotions Label:       surprise
        GoEmotions Score:       91.0%
        Title: Excellent salt beef sandwich
        Review: Great sandwich with gherkins and mustard albeit quite expensive.
               I was very surprised when I got the bill for 20 USD

Morning_Star_Cafe
        Sentiment:              Negative
        Rating:                 2
        GoEmotions Label:       surprise
        GoEmotions Score:       98.4%
        Title: Shocking when busy.
        Review: As soon as this place gets busy, the service gets shockingly bad.
               We asked for things over and over and the staff quite literally ignored us.

The_National_Bar_Dining_Rooms
        Sentiment:              Negative
        Rating:                 1
        GoEmotions Label:       surprise
        GoEmotions Score:       97.8%
        Title: Breakfast review
        Review: Poor service...poor omelette...poor croissant...will not try again!
               Lunch is much better but I was surprised at how bad their offering was...

Jacob_s_Pickles
        Sentiment:              Positive
        Rating:                 5
        GoEmotions Label:       surprise
        GoEmotions Score:       97.1%
        Title: All around great
        Review: I am wonderfully surprised at the menu, the music, the decor and the execution.
               Hats off to the chef and owner. Will be back
```

Figure 8.4: A few sample surprise reviews in the dataset

As you can see in *Figure 8.4*, the commonality between all the review samples is surprise, both negative and positive.

Next, we will leverage BertViz, which, despite the name, can visualize attention for encoder-only transformer models (like BERT and all variants), decoder-only transformers (like GPT and all variants), and encoder-decoder transformers (like T5). It's very flexible, but it's important to note that it's an interactive tool, so the print screens represented by the figures in this section won't do it justice.

Next, we will create a function that, with the tokenizer, model, and a tuple of sentences, can create two different kinds of BertViz visualizations:

```python
def view_attention(tokenizer, model, sentences, view="model"):
    sentence_a, sentence_b = sentences

    # Encode sentences with tokenizer
    inputs = tokenizer.encode_plus(
        sentence_a, sentence_b, return_tensors="pt"
    )

    # Extract components from inputs
    input_ids = inputs["input_ids"]
    token_type_ids = inputs["token_type_ids"]

    # Get attention weights from model given the inputs
    attention = model(input_ids, token_type_ids=token_type_ids)[-1]

    # Get 2nd sentence start and tokens
    sentence_b_start = token_type_ids[0].tolist().index(1)
    input_id_list = input_ids[0].tolist()
    tokens = tokenizer.convert_ids_to_tokens(input_id_list)

    # BertViz visualizers
    if view=="head":
        head_view(attention, tokens, sentence_b_start)
    elif view=="model":
        model_view(attention, tokens, sentence_b_start)
```

To visualize attention, we will need to take a pair of input `sentences` and encode them with our tokenizer (`inputs`). Then, we extract the token IDs for these inputs (`input_ids`) and values, which indicate what sentence each token belongs to (`token_type_ids`) – in other words, `0` for the first sentence and `1` for the second sentence. We then pass the inputs (`input_ids` and `token_type_ids`) to the model and extract the `attention` weights. Finally, there are two BertViz visualizers, `head_view` and `model_view`, and for them to work, all we need is the `attention` weights produced by our inputs, the token IDs converted to `tokens`, and the position for when the second sentence begins (`sentence_b_start`).

Next, we will visualize attention throughout the model with the *model view*.

Plotting all attention with the model view

The following snippet will produce a model view for the sentences in the 1st sample review:

```python
view_attention(
    goemotions_tok, goemotions_mdl, sample_reviews_dict[0], view="model"
)
```

The preceding lines of code create a large plot, like the one portrayed in *Figure 8.5*:
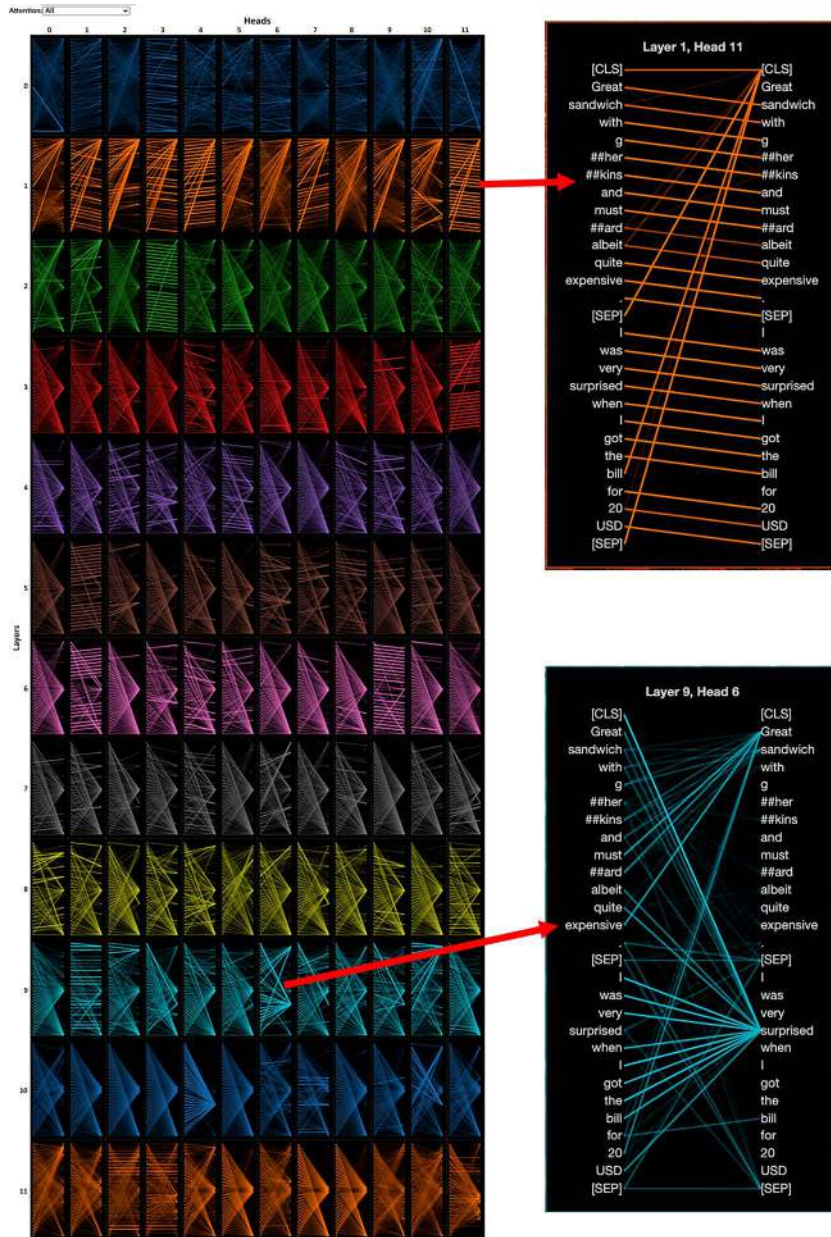
Figure 8.5: The model view for the 1st sample review for 2nd Avenue Deli

Figure 8.5 is a 12 x 12 grid with every attention head in the BERT model. We can click on any attention head to see both sentences in the BERT input, with lines drawn between them to represent the attention weights from one token (left) to another (right). We can select "Sentence A -> Sentence A", "Sentence A -> Sentence B," and every combination in between to view only a subset of all the attention weights. Lines for weights closest to one appear as very opaque, while weights close to zero show as transparent to the point of not being visible at all.

At a glance, we can tell that some attention heads have more lines, thicker lines, or lines that seem to go more in one direction than another. We can click on individual attention heads to examine them individually – for instance:

- **Layer 1 Head 11** is mostly attention, moving forward from one token to the following token in the same sentence. This is a very common pattern and makes complete logical sense because we read English left to right and understand it mostly in that order, although there are, of course, other patterns that are undoubtedly in the attention heads. We also see evidence of another common pattern, which is one or several tokens with attention weights toward the `[CLS]` token. The `[CLS]` token is a special token that is prepended to every input sequence when using BERT-like models for classification tasks. It's often used to get the aggregate representation of the entire sequence for classification. This means that for this particular attention head, the token serves a purpose in the classification decision. This can be especially insightful when trying to understand which words in a sequence the model deems critical for classification decisions. When attention goes from a separator token `[SEP]` to the classification token `[CLS]`, it could be seen as the model recognizing the end of a context or sentence and reflecting its semantic conclusion, to perhaps influence the classification decision.

- **Layer 9 Head** seems to perform a more complicated task, which is to relate words to "great" and "surprised," even across sentences. This is another common pattern where connecting words predict a word.

Take note of patterns in the attention heads, and notice a few others, like attention moving backward in a sentence, or connecting synonyms. Then, change the zero in `sample_reviews_dict[0]` to one, two, or three to see if the attention heads show the same pattern. The samples are quite different, but if the attention heads are not doing the same thing, chances are they are doing something remarkably similar. However, for the bigger picture, it's probably best to squint your eyes and see what kind of patterns are evident across different layers.

Next, we will make this task easier with the head view.

Diving into layer attention with the head view

We can start by choosing the first sample with the following code:

```
view_attention(
    goemotions_tok, goemotions_mdl, sample_reviews_dict[0], view="head"
)
```

We can select any of the 12 layers (0–11). Line transparency means the same thing it did with the model view. However, the difference is we can isolate individual token attention weights by clicking on them, so when we select any token on the left, we'll see lines connect with tokens on the right. Also, color-coded boxes will appear on the right, representing how much attention weight is in each of the 12 attention heads.

And if we happen to select any token on the right, we'll get all the attention that is directed to it from the left tokens. In *Figure 8.6*, we can see examples of how several of the sample sentences would appear:
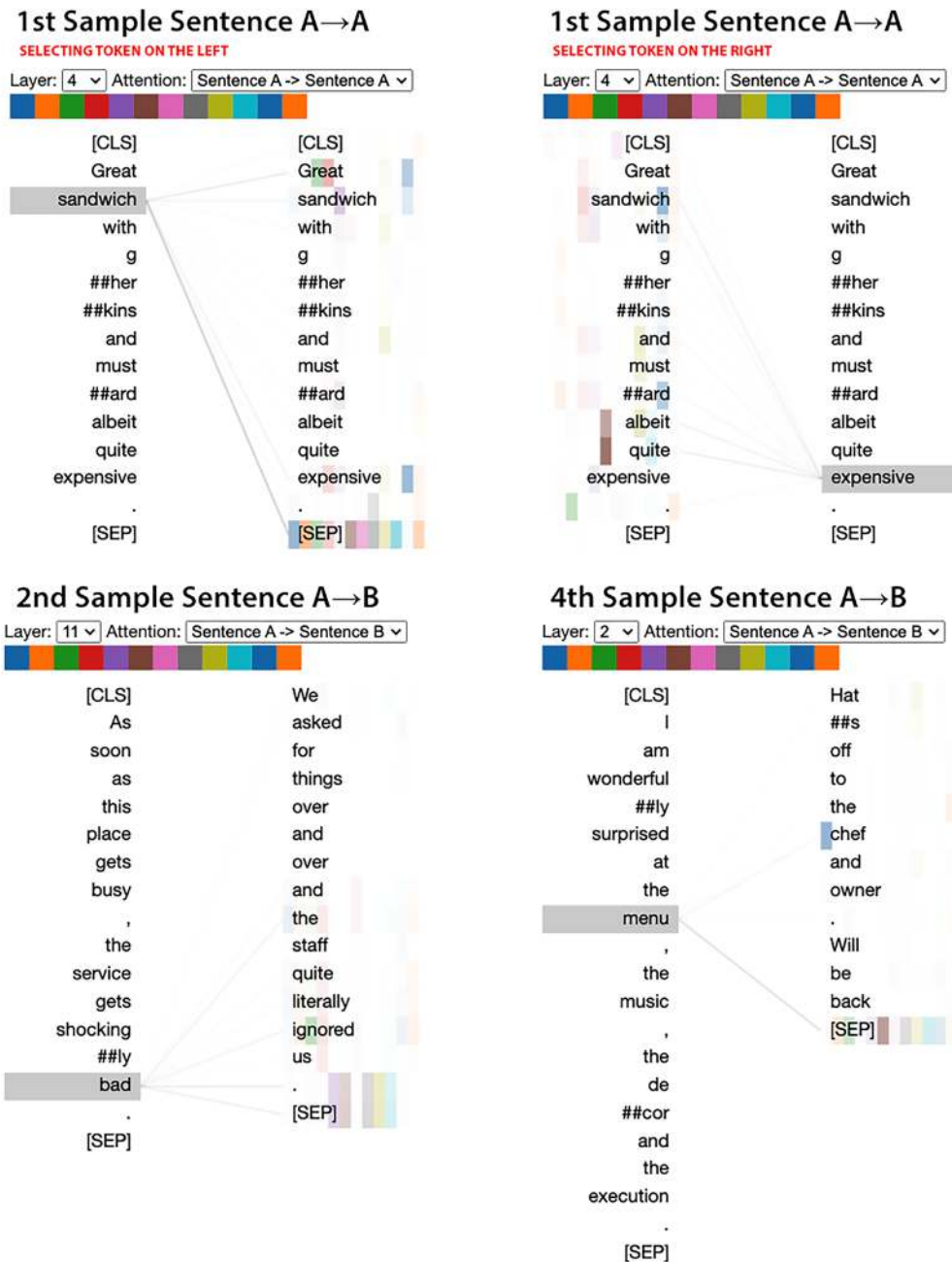
Figure 8.6: The head view for samples 1, 2, and 4

In *Figure 8.6*, see how we can drill down on patterns just as we did with model view. For instance, we can examine different patterns of attention, from related words that are next to each other ("great sandwich" and "albeit quite expensive") to those that have a relationship only in the context of the sentence ("sandwich" and "expensive"), or even across sentences ("bad" and "ignored," "chef," and "menu").

By doing this, we can realize that visualizing attention heads is not just for curiosity's sake but can also help us understand how a model connects the dots between words, thus accomplishing a downstream task like classification. Perhaps this can help us understand what to do next, whether it's to fine-tune the model further with underrepresented words and situations, or prepare the data differently to stop the model from getting confused by a particular word or set of words. However, given the complexity of attention heads finding model issues, this can be like looking for a needle in a haystack. We only started with layers and attention heads in this chapter because it provides an intuitive way of understanding how transformers encode relationships between tokens.

A better way to start would be with attributions. An attribution method is a method that will compute how much a part of an input contributed to a model's prediction for that input. In the case of images in *Chapter 7, Visualizing Convolutional Neural Networks*, the part of the input we compute attributions for is pixels. For text, the equivalent would be tokens, which in this case are made up of (mostly) words, so next, we will generate token attributions.

Interpreting token attributions with integrated gradients

Integrated gradients is a popular method, and in *Chapter 7*, we explained and leveraged it to produce attributions for each pixel in an image. The method has the same steps:

1. **Choose a baseline input**: The baseline represents no information. For images, it is usually a solid black image. For text, this could be a sentence with all words replaced by a placeholder like [PAD] or just an empty sentence.
2. **Gradually change this baseline input** into your actual input sentence (e.g., the review), step by step. At each step, you change a little bit of the baseline toward the actual input.
3. **Compute output changes**: For each step, calculate how much the model's prediction changes.
4. **Sum up all these changes** for each word in the sentence. This gives you a score for each word, indicating how much it contributed to the model's final prediction.

However, before we can use integrated gradients, it's best that we define a transformer pipeline that tokenizes the input and performs inference on the model in one step:

```
goemotions = pipeline(
    model=goemotions_mdl,
    tokenizer=goemotions_tok,
    task="text-classification",
    function_to_apply="softmax",
    device=device,
    top_k=None
)
```

You can test the goemotions pipeline like this:

```
goemotions(["this restaurant was unexpectedly disgusting!",
           "this restaurant was shockingly amazing!"])
```

It should output the following list of lists of dictionaries:

```
[[{"label": "disgust", "score": 0.961812436580658},
  {"label": "surprise", "score": 0.022211072966456413},
  {"label": "sadness", "score": 0.004870257806032896},
  {"label": "anger", "score": 0.0034139526542276144},
  {"label": "joy", "score": 0.003016095608472824},
  {"label": "fear", "score": 0.0027414397336542606},
  {"label": "neutral", "score": 0.0019347501220181584}],
 [{"label": "joy", "score": 0.6631762385368347},
  {"label": "surprise", "score": 0.3326122760772705},
  {"label": "neutral", "score": 0.001732577453367412},
  {"label": "anger", "score": 0.0011324150254949927},
  {"label": "sadness", "score": 0.0010101954967249184185},
  {"label": "fear", "score": 0.00021178492170292884},
  {"label": "disgust", "score": 0.00011514205834828317}]]
```

As you can see, in the first list, there are two predictions (one for each text), and each prediction has a list with seven dictionaries, with one that has the score for each class. Since the dictionaries are sorted from the highest score to the lowest, you can tell that the first restaurant review was mostly predicted as disgust and the second as joy at 66%, but there was also a sizable amount of surprise.

Next, we will create a function that can take any DataFrame row with our review and our transformer, and generate and output attributions for every prediction with over 10% probability:

```python
def visualize_ig_review(interpret_s:pd.Series,
                        pline:pipeline,
                        max_prob_thresh:float=0.1,
                        max_classes=np.PINF,
                        concat_title=True,
                        summary_df=None
) -> pd.DataFrame:
    print(f"{interpret_s.name}: {interpret_s['restaurant_name']}")

    # Init some variables
    if concat_title:
        text = interpret_s["review_title"] + ":" + interpret_s["review_full"]
    else:
        text = interpret_s["review_full"]
    true_label = "Positive" if interpret_s["positive_sentiment"]\
                            else "Negative"
    rating = interpret_s["rating"]
```

```python
# Get predictions
prediction = pline(text)[0]
prediction_df = pd.DataFrame(prediction)
if summary_df is not None:
    prediction_df["label_avg_rating"] = prediction_df.label.\
        replace(summary_df["avg. rating"].to_dict())
    prediction_df = prediction_df.sort_values("label_avg_rating",\
        ascending=False).reset_index(drop=True)

# Process predictions
prediction_tuples = [(p["label"], p["score"]) for p in prediction]
sorted_prediction_tuples = sorted(prediction_tuples,\
    key=lambda x: x[1], reverse=True)
pred_class, pred_prob = sorted_prediction_tuples[0]

# Initialize Integrated Gradients
forward_func = lambda inputs, position=0: pline.model(
    inputs, attention_mask=torch.ones_like(inputs)
)[position]
layer = getattr(pline.model, "bert").embeddings
lig = LayerIntegratedGradients(forward_func, layer)

# Prepare tokens and baseline
device = torch.device("cuda:0" if torch.cuda.is_available()\
                      else "cpu")
inputs = torch.tensor(pline.tokenizer.encode(text,\
    add_special_tokens=False), device = device).unsqueeze(0)
tokens = pline.tokenizer.convert_ids_to_tokens(
    inputs.detach().numpy()[0]
)
sequence_len = inputs.shape[1]
baseline = torch.tensor(
    [pline.tokenizer.cls_token_id]\
    + [pline.tokenizer.pad_token_id] * (sequence_len - 2)\
    + [pline.tokenizer.sep_token_id],\
    device=device
).unsqueeze(0)

# Iterate over every prediction
vis_record_l = []
for i, (attr_class, attr_score) in\
```

```
        enumerate(sorted_prediction_tuples):
        if (attr_score > max_prob_thresh) and (i < max_classes):
            # Sets the target class
            target = pline.model.config.label2id[attr_class]
            # Get attributions
            with torch.no_grad():
                attributes, delta = lig.attribute(
                    inputs=inputs,
                    baselines=baseline,
                    target=target,
                    return_convergence_delta = True
                )

            # Post-processing attributions
            attr = attributes.sum(dim=2).squeeze(0)
            attr = attr / torch.norm(attr)
            attr = attr.cpu().detach().numpy()

            # Generate & Append Visualization Data Record
            vis_record = visualization.VisualizationDataRecord(
                    word_attributions=attr,
                    pred_prob=pred_prob,
                    pred_class=pred_class,
                    true_class=f"{true_label} ({rating})",
                    attr_class=attr_class,
                    attr_score=attr_score,
                    raw_input_ids=tokens,
                    convergence_score=delta
            )
            vis_record_l.append(vis_record)

    # Display list of visualization data records
    _ = visualization.visualize_text(vis_record_l)
    return prediction_df
```

It may seem complicated by the amount of code, but there are plenty of steps that are relatively straightforward when explained individually. We will start with model inference and work our way down:

1. **Get predictions:** This is a very straightforward step. It just feeds the `text` into the pipeline (`pline`). It takes only the first item returned (`[0]`) because it only anticipates one prediction being inputted and, thus, returned by the pipeline. The next few lines show what the model does if the function **receives** a `sample_df`, which it only really needs to sort the predictions in the order of the best rating on average.

2. **Process predictions:** Here, the code makes sure predictions are sorted and in tuples for easier iteration in the `for` loop that follows later.

3. **Initialize integrated gradients:** A forward function is defined, which takes inputs and returns the model's output for a given position, as well as a layer for which the attributions will be calculated, which in this case is the embedding layer. Then, an instance of `LayerIntegratedGradients` (`lig`) is initialized with the forward function and the specified layer.

4. **Prepare tokens and baseline:** First, the text is tokenized and converted into a tensor, which is then moved to the specified device. Then, the token IDs are converted back to `tokens` for potential visualization or analysis. A `baseline` is created for the integrated gradients method. It consists of `[CLS]` token at the start, `[SEP]` token at the end, and `[PAD]` tokens in the middle matching the length of the input text.

5. **Iterate over every prediction:** Here's the `for` loop that iterates over every prediction as long as the probability is over 10%, as defined by `max_prob_threshold`. Within the `for` loop, we:

   a. **Set the target class:** Integrated gradients is a directed attribution method, so we need to know what `target` class to generate attributions for; therefore, we need the ID that was used internally by the model for the predicted class.

   b. **Get the attributions:** Using the very same Captum `attribute` method we used in *Chapter 7*, we generate the IG attributions for the tokenized version of our text (`inputs`), the `baselines`, the `target`, and decide whether to return a delta (a measure of approximation error) of the IG method (`return_convergence_delta`).

   c. **Post-process the attributions:** the attributions returned by the IG method are of the shape (`num_inputs`, `sequence_length`, `embedding_dim`), where `embedding_dim=768` for this model, `sequence_length` corresponds to the number of tokens in the input, and `num_inputs=1` because we only perform one attribution at a time. So each token's embedding has an attribution score, but what we need is one attribution per token. Therefore, these scores are summed across the embedding dimension to get a single attribution value for each token in the sequence. Then, the attributions are normalized, ensuring that the attributions have a magnitude between 0 and 1 and are in a comparable scale. Finally, the attributions are detached from the computation graph, moved to the CPU, and converted to a `numpy` array for further processing or visualization.

d. **Generate and append the Visualization Data Record:** Captum has a method called `VisualizationDataRecord`, which creates a record of each attribution for visualization purposes, so what we do in this step is create these records with the attributions, deltas, tokens, and metadata related to the prediction. It then appends this data record to a list.

6. **Display the list of Visualization Data Records:** leverage `visualize_text` to display the list of records.

Now, let's create some samples to perform integrated gradient attributions on:

```python
neg_surprise_df = reviews_df[
    (reviews_df["label"]=="surprise")
    & (reviews_df["score"]>0.9)
    & (reviews_df["positive_sentiment"]==0)
    & (reviews_df["rating"]<3)
] #43

neg_surprise_samp_df = neg_surprise_df.sample(
    n=10, random_state=rand
)
```

In the above snippet, we take all surprise reviews with a probability over 90%, but to ensure that they are negative, we will select a negative sentiment and a rating below three. Then, we will take a random sample of 10 reviews from these.

Next, we will iterate across every review in this list and generate some visualizations. A few others are shown in the screenshot in *Figure 8.7:*

```python
for i in range(10):
    sample_to_interpret = neg_surprise_samp_df.iloc[i]
    _ = visualize_ig_review(
        sample_to_interpret, goemotions, concat_title=True, summary_df=summary_
df
    )
```

The preceding snippet of code will produce the visualizations in *Figure 8.7*:

355891: Puglia_Restaurant

Legend: ■ Negative □ Neutral ■ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
Negative (1)	surprise (0.92)	surprise	0.92	Offensive Italian : Am simply shocked this restaurant tries to pass themselves off as authentic , traditional Italian – it is only appropriate for the university age " all you can drink " crowd that can ' t b ##oil past ##a themselves . I have been to countless Italian restaurants throughout NY and P ##ug ##lia may actually be the worst (even inferior to $ 1 pizza places) . Regarding the food , the best thing we ate was the garlic bread . Otherwise , simple , plain , b ##land , Italian staple ##s . If in a large group with pre fix , expect a heap ##ing amount of pen ##ne in various sauce ##s . The house wine is o ##bs ##cene ##ly und ##rin ##ka ##ble . They have music on the weekends - this actually is enjoyable momentarily , but they have a limited repertoire and will ha ##rass you for tips in between sets .

507090: Le_Pain_Quotidien

Legend: ■ Negative □ Neutral ■ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
Negative (1)	surprise (0.91)	surprise	0.91	5 $ stolen ? : Food is great : no complaints . But the service at the check out counter : un ##bel ##ie ##va ##bly bad . I purchased a c ##rois ##san ##ts , tea with milk , gave cash , expecting 5 $ change . While waiting for the change , the c ##rois ##san ##t was brought in a bag , which I checked : it was smashed . The tea had no milk . Then I realized that I had never received the 5 $ change , to the best of my knowledge . . . but the cash ##ier was now (within seconds after taking my payments) convenient ##ly " taking a break and no longer in the building " , according to another service person (the manager ?) , at 3 : 04 pm , April 21st , 2019 . Nobody else , of course , could now give me my change . I know : I should have checked .

193197: Almond

Legend: ■ Negative □ Neutral ■ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
Negative (2)	surprise (0.93)	surprise	0.93	Don ' t bother : There is so many great places to eat in Gram ##er ##cy I am not sure how we ended up here . The staff were more interested in gossip ##ing than looking after the 8 people that had made the same mistake we had . Lamb ch ##ops as a meal means more than 1 and a half cut ##lets , how you can make a b ##urger so dry and taste ##less is a mystery , the past ##a was ok . Had to beg for a beer even . . . . to hard don ' t bother .

174001: Katz_s_Deli

Legend: ■ Negative □ Neutral ■ Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
Negative (1)	surprise (0.96)	surprise	0.96	Shock ##ed : A long awaited trip to New York , and Katz diner was the place I wanted to visit , only a 15 ##min walk from our Hotel , perfect . It was 10 o ' clock on a sunny Sunday morning . En ##tering we were met by a very over ##weight man leaning back in a chair " here ' s you ' re ticket order there , pay on your way out " and do you pay , 3 past ##ram ##i sandwiches 1 cheese o ##mel ##ette and 3 regular coffee ##s just shy of $ 120 . People think London is expensive , considering Katz location it ' s extremely expensive , sandwiches not impressive and a g ##loom ##y and slightly g ##rea ##sy de ##cor (I know it ' s meant to be vintage but wipe the tables down and m ##op the floor) overall a very bad experience from the staff and the food . Don ' t waste your money going to Katz .

Figure 8.7: IG visualizations for negative surprises

As we can see in *Figure 8.7*, the surprise prediction is attributed to words like "shocked," "realized," and "mystery" and phrases like "not sure how." These all make sense because they indicate that something is unknown. Naturally, there are also a few cases where the word "surprise" or "surprised" is all it takes to get a surprise prediction. However, sometimes it's not that simple. In the last one, it's not one word that appears to indicate surprise but many words, saying to the effect that something doesn't add up. More specifically, these visitors from London were very surprised that a deli in New York City was so expensive. Please note that the color coding for "Negative" and "Positive" doesn't mean that a word is negative or positive but, rather, it's weighting against (negatively) or in favor (positively) of the attribution label.

Next, we are going to repeat code similar to what we ran to generate IG explanations for a sample of surprise negative reviews, but this time for positive reviews. To ensure that they are positive, we will use ratings above 4. This time, we will make sure to remove any reviews with the word "surprise" from the samples just to make things interesting:

```python
pos_surprise_df = reviews_df[
    (reviews_df["label"]=="surprise")
    & (reviews_df["score"]>0.97)
    & (reviews_df["positive_sentiment"]==1)
    & (reviews_df["rating"]>4)
]
pos_surprise_samp_df = pos_surprise_df[
    ~pos_surprise_df["review_full"].str.contains("surprise")
]
for i in range(10):
    sample_to_interpret = pos_surprise_samp_df.iloc[i]
    _ = visualize_ig_review(
        sample_to_interpret, goemotions,\
        concat_title=False, summary_df=summary_df
    )
```

The preceding code will produce the visualizations in *Figure 8.8*.

Figure 8.8. IG visualizations for positive surprises

Figure 8.8 shows how the words "perplexed" and "unbelievable," as well as the phrase "couldn't believe how" indicate surprise. There are also a few cases of tokens weighing negatively against the surprise prediction. For instance, for the last restaurant, having "something for everyone" doesn't make it very surprising. Also, you'll note the Japanese restaurant in the middle is predicted to embody both surprise and joy emotions. It's interesting how some words correlate with one emotion but not so much with another, and sometimes they indicate the opposite, like the "hard" in "it's very hard to find a place" indicating surprise but not joy.

Finding reviews with mixed sentiments like the Japanese restaurant might hold some answers to why some reviews are hard to classify entirely with one sentiment. So, now, we will produce some positive and negative mixed review samples. We can easily do so by making sure that the score for the predicted label is never over 50%:

```
pos_mixed_samp_df = reviews_df[
    (~reviews_df["label"].isin(["neutral","joy"]))
    & (reviews_df["score"] < 0.5)
    & (reviews_df["positive_sentiment"]==1)
    & (reviews_df["rating"]< 5)
].sample(n=10, random_state=rand)

neg_mixed_samp_df = reviews_df[
    (~reviews_df["label"].isin(["neutral","joy"]))
    & (reviews_df["score"] < 0.5)
    & (reviews_df["positive_sentiment"]==0)
```

```
      & (reviews_df["rating"]>2)
].sample(n=10, random_state=rand)
```

We can generate positive mixed sentiment reviews with the following snippet. Note that this time, we use a method called `mldatasets.plot_polar`, which plots a polar line chart for the predictions with `plotly`. You'll need both `plotly` and `kaleido` to make this work:

```
for i in range(10):
    sample_to_interpret = pos_mixed_samp_df.iloc[i]
    prediction_df = visualize_ig_review(
        sample_to_interpret,\
        goemotions, concat_title=False,\
        summary_df=summary_df
    )
    rest_name = sample_to_interpret["restaurant_name"]
    mldatasets.plot_polar(
    prediction_df, "score", "label", name=rest_name
)
```

The preceding code will produce the IG visualization and polar line plot in *Figure 8.9*:
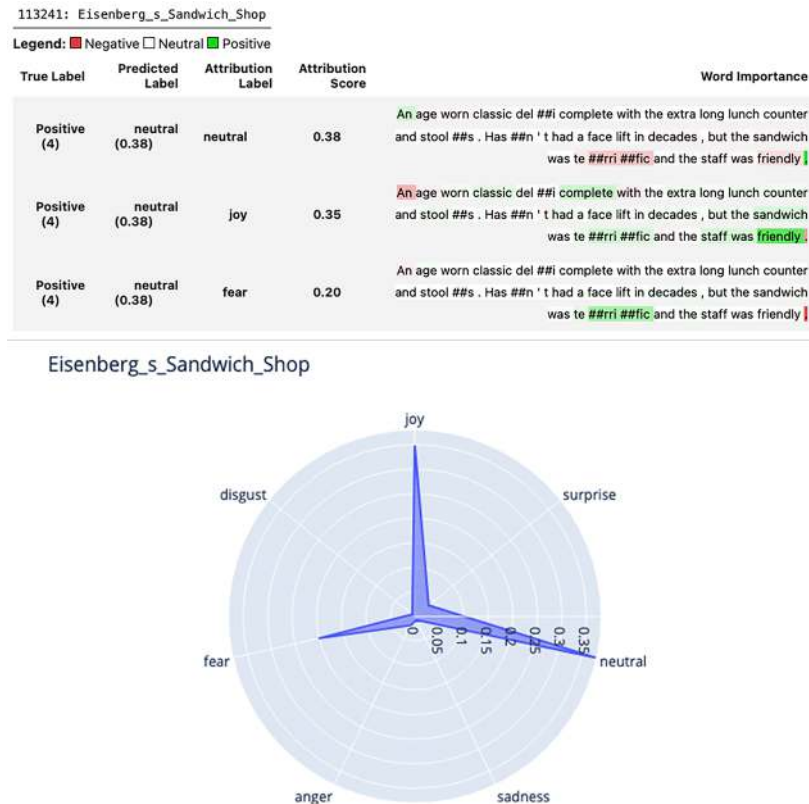
Figure 8.9: IG visualizations for mixed sentiment reviews

Figure 8.9 portrays how this review for a sandwich shop appears to have joy, fear, and neutral emotions. The words "terrific" and "friendly" connect with joy, but they don't with neutral. However, strangely, the word "terrific" also correlates with fear. Perhaps it has to do with the WordPiece tokenization performed on the word. Note that "terrific" appears as three subword tokens, te, ##rri, and ##fic. This probably happened because the original corpus used to train the model (the Reddit comments) didn't have enough frequency for "terrific" to include it as a standalone word, but these subwords did. The downside to this technique is that it's possible that the "te" and "rri" tokens are used often for words like "terrifying," and "fic" in other scary words like "horrific" and "mortific." On the other hand, "fic" appears in "magnificent" and "beneficial." So what happens is that despite the contextual embeddings, the subword tokens can cause some ambiguities.

We can now run the same code as before but for `neg_mixed_samp_df`, to examine other examples and make our own conclusions. Next, we can expand our XAI NLP-specific toolset with the LIT.

LIME, counterfactuals, and other possibilities with the LIT

The LIT is an open-source platform developed by the **People+AI Research** (**PAIR**) initiative to visualize and understand NLP models. PAIR developed the What-If Tool featured in *Chapter 6*, *Anchors and Counterfactual Explanations*.

LIT provides an interactive and visual interface to delve deep into NLP model behavior. With LIT, users can:

- Identify types of examples where a model underperforms.
- Determine reasons behind specific model predictions.
- Test the model's consistency under textual variations, like style, verb tense, or pronoun gender.

LIT offers various built-in capabilities, including salience maps, attention visualization, metrics calculations, and counterfactual generation. However, it also supports customization, allowing the addition of specialized interpretability techniques, visualizations, and more.

Although LIT's primary focus is textual language data, it also supports models that operate on image and tabular data. It's compatible with a range of machine learning frameworks, including TensorFlow and PyTorch. The tool can run both as a standalone server and within notebook environments like Colab, Jupyter, and Google Cloud Vertex AI notebooks.

In order to work with any custom dataset, LIT provides a `Dataset` subclass to create a LIT-compatible dataset loader. You must include an `__init__`, which loads the dataset, and a spec function, which specifies the data types returned in the dataset, while the `lit_nlp.api.types` provide a way to ensure that LIT recognizes each feature in your dataset. In this case, we provide the review (`TextSegment`), the label (`CategoryLabel`) with the seven labels, and two additional categories, which can be used for slicing and binning:

```
class GEDataset(lit_dataset.Dataset):
    GE_LABELS = ["anger", "disgust", "fear", "joy",\
```

```
                    "neutral", "sadness", "surprise"]
    def __init__(self, df: pd.DataFrame):
        self._examples = [{
            "review": row["review_title"] + ":" + row["review_full"],
            "label": row["label"],
            "rating": row["rating"],
            "positive": row["positive_sentiment"]
        } for _, row in df.iterrows()]
    def spec(self):
        return {
            "review": lit_types.TextSegment(),
            "label": lit_types.CategoryLabel(vocab=self.GE_LABELS),
            "rating": lit_types.CategoryLabel(),
            "positive": lit_types.CategoryLabel()
        }
```

To make LIT flexible to accommodate any model, there is a `Model` subclass to make a LIT-compatible model loader. It also needs the `__init__` function to initialize the model, as well as a `predict_minibatch` function to predict with it. To this end, we also need to create specs for both the inputs (`input_spec`) and outputs (`output_spec`) of the `predict` function. In this case, we enter a review (of type `TextSegment`) and return probabilities of type `MulticlassPreds`. Remember that the output of the model is not always consistent, since each prediction is arranged from the highest to the lowest score. Note that in order to make the output of `predict_minibatch` comply with the `MulticlassPreds`, we have to arrange the probabilities as a list corresponding to the labels (`GE_LABELS`), in the same order provided to `vocab`:

```
class GEModel(lit_model.Model):
    GE_LABELS = ["anger", "disgust", "fear", "joy",\
                 "neutral", "sadness", "surprise"]
    def __init__(self, model, tokenizer, **kw):
        self._model = pipeline(
            model=model,
            tokenizer=tokenizer,
            task="text-classification",
            function_to_apply="softmax",
            device=device,
            top_k=None
        )
    def input_spec(self):
        return {
            "review": lit_types.TextSegment()
        }
```

```python
    def output_spec(self):
        return {
            "probas": lit_types.MulticlassPreds(vocab=self.GE_LABELS,\
                                                parent="label")
        }
    def predict_minibatch(self, inputs):
        examples = [d["review"] for d in inputs]
        with torch.no_grad():
            preds = self._model(examples)
        preds = [{p["label"]:p["score"] for p in pred_dicts}\
                for pred_dicts in preds]
        preds = [dict(sorted(pred_dict.items())) for pred_dict in preds]
        preds = [{"probas": list(pred_dict.values())} for pred_dict in preds]
        return preds
```

OK, so now we have the two classes we need for LIT to function. The GoEmotions model initializer (`GEModel`) takes the model (`goemotions_mdl`) and tokenizer (`goemotions_tok`). We put these in a dictionary because LIT can take more than one model and more than one dataset to compare them. For the dataset, to make it load quickly, we will use 100 samples (`samples100_df`), made up of the four 10-sample DataFrames we have created so far, plus 60 additional random samples from the entire reviews dataset. Then, we just input our 100-sample DataFrame into the GoEmotions dataset initializer (`GEDataset`) and place it into our datasets dictionary as `NYCRestaurants`. Lastly, we create the widget (`notebook.LitWidget`) by inputting our model and datasets dictionaries and `render` it. Please note that if you want to run this outside of a notebook environment, you can use the `Server` command to have it run on a LIT server:

```python
models = {"GoEmotion":GEModel(goemotions_mdl, goemotions_tok)}

samples100_df = pd.concat(
    [
        neg_surprise_samp_df,
        pos_surprise_samp_df,
        neg_mixed_samp_df,
        pos_mixed_samp_df,
        reviews_df.sample(n=60, random_state=rand)
    ]
)
datasets = {"NYCRestaurants":GEDataset(samples100_df)}
widget = notebook.LitWidget(models, datasets)
widget.render(height=600)
# litserver = lit_nlp.dev_server.Server(models, datasets, port=4321)
# litserver.serve()
```

The above snippet will produce the interface in *Figure 8.10*:

Figure 8.10: Notebook view with the Predictions tab open

As you can appreciate in *Figure 8.10*, LIT has:

- A top bar with a dropdown to select the model and dataset (but you can't here because there's only one of each) and three different views (**simple**, **default**, and **notebook**). **notebook** is selected by default.
- A selection bar to select datapoints and see which ones are pinned.
- A tab bar with three tabs (**Predictions**, **Explanations**, and **Analysis**). By default, **Predictions** is selected, and this tab has **Data Table** to the left, where you can select and pin individual datapoints, and the **Classification Results** pane to the right.

Even though the **notebook** view has much more going on than the **simple** view, it lacks many features available in the **default** view. From now on, we will examine the **default** view depicted in *Figure 8.11*:
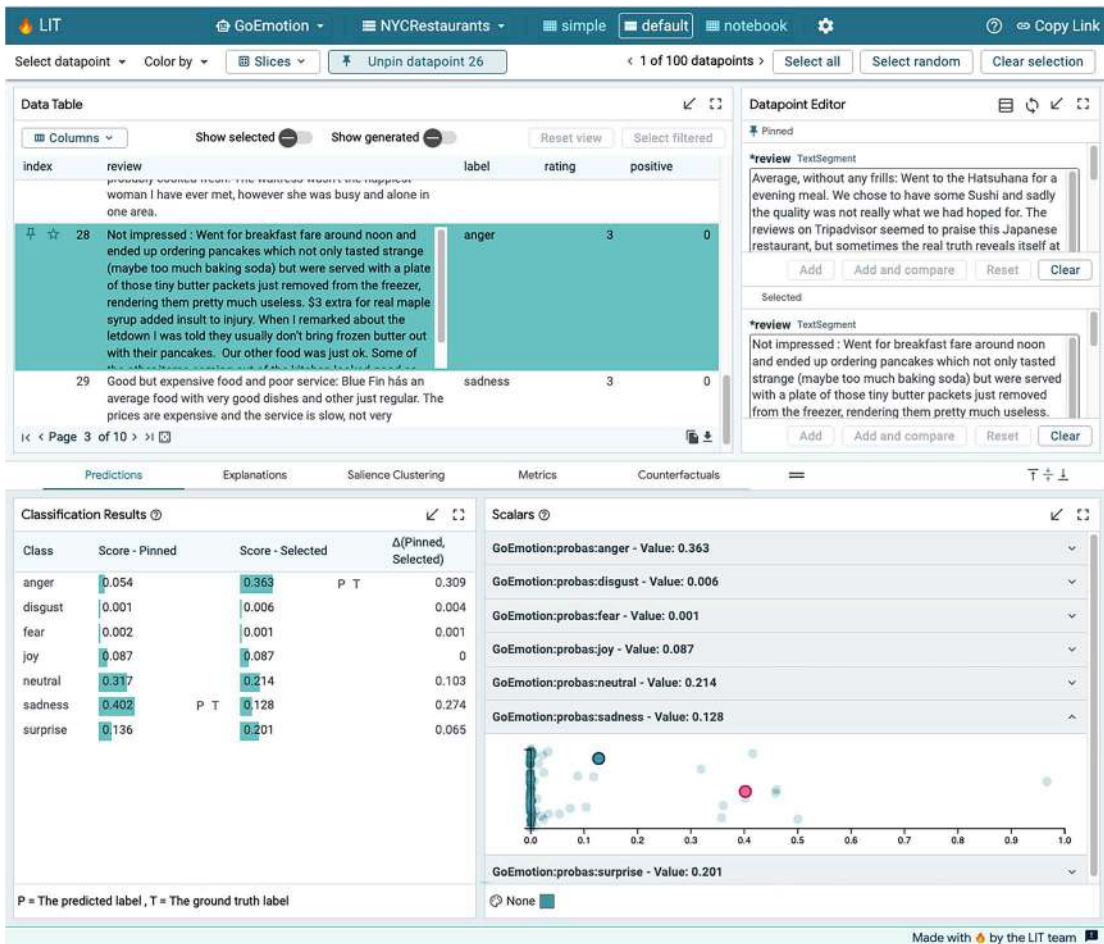
Figure 8.11: The default view with the Predictions tab open

As you can see in *Figure 8.11*, the default view has two permanent panes, with the **Data Table** and **Datapoint Editor** in the top pane and the tabs in the bottom pane. It's not a great layout for a small notebook cell, but it can let you easily pin, select, and edit datapoints while also performing tasks on them in the tabs below. Note also that there are more than three tabs. We will briefly explain each one:

- **Predictions:** This lets you see classification results for selected and pinned datapoints. Note that it denotes the predicted label with a "P" and the ground truth with a "T." However, since we didn't train the model with this dataset, the label provided is no different than the one predicted, but this can prove very useful for examining misclassifications. To the right of the **Classification Results**, we have **Scalars**, which allows us to compare scores for the pinned and selected datapoints with all others in the dataset.

- • **Explanations:** Here, we can use a number of explanation/attribution methods on our datapoints, such as LIME and integrated gradients.
- • **Salience Clustering:** We perform attributions on many datapoints and cluster the results to understand how tokens are clustered. We won't go into details here, given that we only are using a dataset of 100.
- • **Metrics:** Had we been using a training dataset with ground truth labels, this tab would be very useful because it can slice and bin performance metrics in many ways.
- • **Counterfactuals:** Much like with *Chapter 6*, the concept of counterfactuals is the same here, which is working out what feature (a token in this case) you can change in such a way that you modify the model outcome (the predicted label). There are several counterfactual finding methods provided.

So, we will work our way down this list, excluding **Salience Clustering** and **Predictions** (which we already explained in *Figure 8.11*), so next, we will take a look at **Explanations**, as shown in *Figure 8.12*:
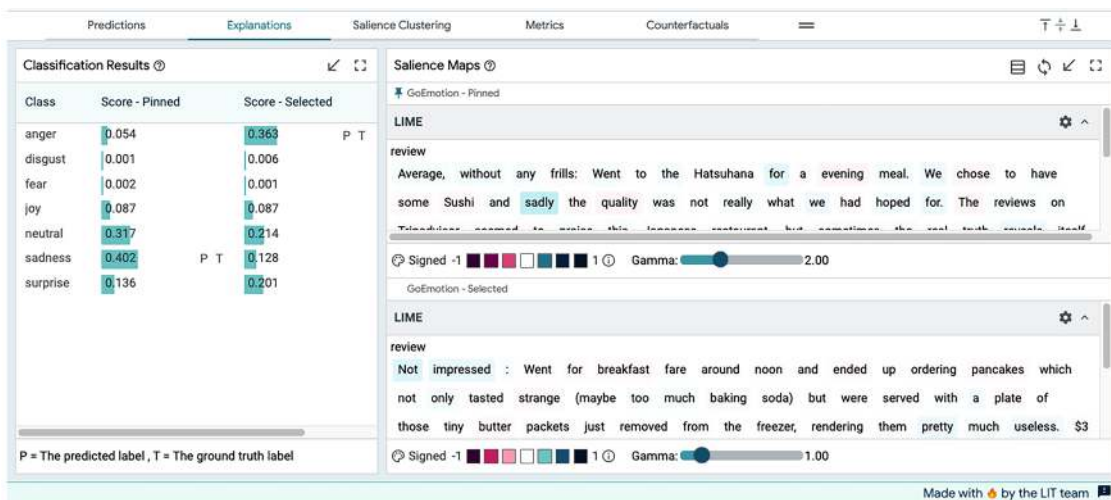
Figure 8.12: LIME explanation compared between pinned and selected reviews in the Explanations tab

Figure 8.12 shows how LIME explanations vary between the pinned and selected datapoint. LIME was previously covered in *Chapter 5*, *Local Model-Agnostic Interpretation Methods*, and in the context of NLP no less. It's the same here. Incidentally, although there are at least four methods available, including integrated gradients, only LIME will work with this model. The reason for this is that LIME is a model-agnostic permutation-based method that doesn't need to access any of the intrinsic parameters of the model, but the rest of the methods aren't model-agnostic. And if you recall, our `GEModel` doesn't expose any of the intrinsic parameters. If we wanted to leverage gradient-based methods like IG within LIT, we would need to not use the pipeline and then specify the input and output in such a way that token embeddings are exposed. There are some examples on the LIT website that can help you accomplish this.

Next, we will take a look at the **Metrics** tab, as seen in *Figure 8.13*:
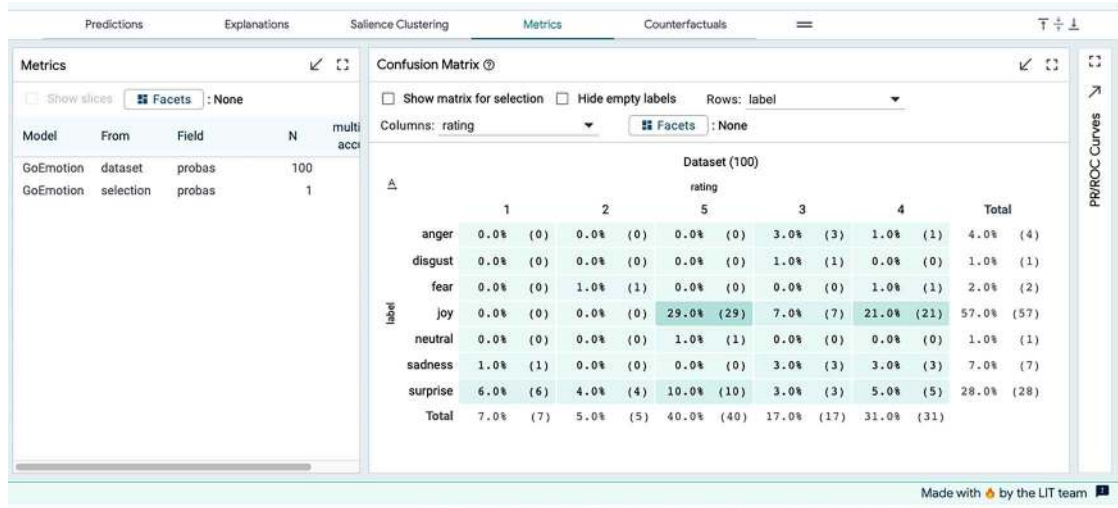
Figure 8.13: Confusion Matrix in the Metrics tab

In *Figure 8.13*, in the **Metrics** panel, there usually would be informative metrics for the entire data-set, the selection you have made, and any additional facets you may define. However, if you expand the tab, you'll always see 100% accuracy for this dataset because there's no ground truth. Perhaps the **Confusion Matrix** panel to the right is more informative in this case because we can see a cross tab between labels and rating, or labels and positive, since we defined both rating and positive as `CategoryLabels`. Please note that, technically, it's not a confusion matrix because it doesn't compare a predicted sentiment label against the corresponding true label, but you can see how much agreement there is between the predicted label and the rating.

Finally, let's examine the **Counterfactuals** tab, depicted in *Figure 8.14*:
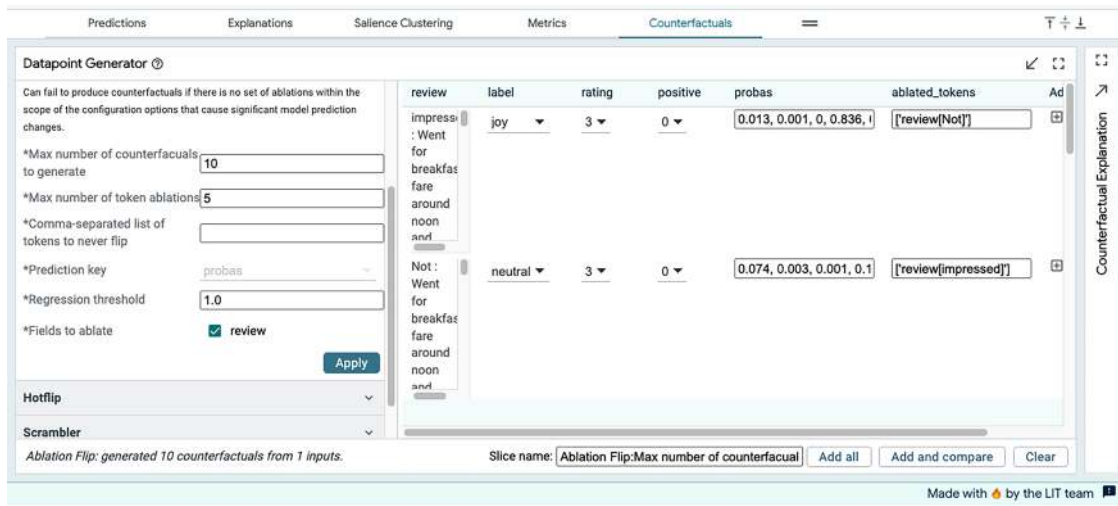
Figure 8.14: Generating ablation flip counterfactuals in the Counterfactuals tab

The **Counterfactuals** tab in *Figure 8.14* provides several methods to change the input in such a way that the predicted label is modified. Not all methods will work with `GEModel`, given how some counterfactual methods are model-agnostic and others require intrinsic parameters. Here, we use ablation flip, a model-agnostic method, to ablate (remove) tokens from the inputs. Ablation flip simply tries dropping tokens from the input to figure out which one changes the prediction. As you can see, the first one removed "Not" from "review," and the second removed "impressed."

With counterfactuals, you can test that the subword tokens in te ##rri ##fic (as depicted in *Figure 8.9*) indeed cause ambiguity when they are added and ablated in many different contexts. For instance, you can remove one token at a time from te ##rri ##fic from a review that is deemed to have a neutral or negative sentiment by the model, seeing if the prediction changes in a positive direction. You can also replace all three tokens with a synonym like "magnificent."

Mission accomplished

It was pretty evident from the summary table (*Figure 8.3*), and confirmed with the integrated gradients and, to some degree, the attention visualization exercise, that many of the Ekman emotions are hard to discern from the reviews, with fear, disgust, anger, and sadness producing many mixed sentiment reviews. And these hard ones are all negative emotions.

Also, many emotions in both the GoEmotions and Ekman taxonomy don't matter as much in the context of a recommendation engine, so it makes sense to consider consolidating some of the negative emotions and, given the ambiguity with the surprise category being sometimes positive and sometimes negative, splitting them to include curiosity and confusion.

Another important finding was that, given the many consistent patterns you found, surprise is not as hard to classify and yet a critical emotion to predict. However, there are good surprises and bad surprises. And given the right training data, a model can likely differentiate both with high precision. We can ensure that tokenization never separates words that convey emotions for the training corpus.

Summary

After reading this chapter, you should understand how to leverage BertViz to visualize transformer models, layers, and attention heads, and how to use Captum's attribution methods, more specifically integrated gradients, and the Visualization Data Record to see what tokens are responsible for a predicted label. Finally, you should have a solid grasp of how to get started with the LIT. In the next chapter, we will look at interpreting multi-variate time-series models.

Further reading

- Vig, J., 2019, *A Multiscale Visualization of Attention in the Transformer Model*. ArXiv: `https://arxiv.org/abs/1906.05714`
- Kokhlikyan, N., Miglani, V., Martin, M., Wang, E., Alsallakh, B., Reynolds, J., Melnikov, A., Kliushkina, N., Araya, C., Yan, S., & Reblitz-Richardson, O., 2020, *Captum: A unified and generic model interpretability library for PyTorch*. ArXiv: `https://arxiv.org/abs/2009.07896`

- Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., & Yuan, A., 2020, *The Language Interpretability Tool: Extensible, Interactive Visualizations and Analysis for NLP Models.* Conference on Empirical Methods in Natural Language Processing: `https://arxiv.org/abs/2008.05122`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

9

Interpretation Methods for Multivariate Forecasting and Sensitivity Analysis

Throughout this book, we have learned about various methods we can use to interpret supervised learning models. They can be quite effective at assessing models while also uncovering their most influential predictors and their hidden interactions. But as the term supervised learning suggests, these methods can only leverage known samples and permutations based on these known samples' distributions. However, when these samples represent the past, things can get tricky! As the Nobel laureate in physics Niels Bohr famously quipped, "Prediction is very difficult, especially if it's about the future."

Indeed, when you see data points fluctuating in a time series, they may appear to be rhythmically dancing in a predictable pattern – at least in the best-case scenarios. Like a dancer moving to a beat, every repetitive movement (or frequency) can be attributed to seasonal patterns, while a gradual change in volume (or amplitude) is attributed to an equally predictable trend. The dance is inevitably misleading because there are always missing pieces of the puzzle that slightly shift the data points, such as a delay in a supplier's supply chain causing an unexpected dent in today's sales figures. To make matters worse, there are also unforeseen catastrophic once-in-a-decade, once-in-a-generation, or simply once-ever events that can radically make the somewhat understood movement of a time series unrecognizable, similar to a ballroom dancer having a seizure. For instance, in 2020, sales forecasts everywhere, either for better or worse, were rendered useless by COVID-19!

We could call this an extreme outlier event, but we must recognize that models weren't built to predict these momentous events because they were trained on almost entirely likely occurrences. Not predicting these unlikely yet most consequential events is why we shouldn't place so much trust in forecasting models to begin with, especially without discussing certainty or confidence bounds.

This chapter will examine a multivariate forecasting problem with **Long Short-Term Memory** (**LSTM**) models. We will first assess the models with traditional interpretation methods, followed by the **Integrated Gradient** method we learned about in *Chapter 7, Visualizing Convolutional Neural Networks*, to generate our model's local attributions.

But more importantly, we will understand the LSTM's learning process and limitations better. We will then employ a prediction approximator method and SHAP's `KernelExplainer` for both global and local interpretation. Lastly, *forecasting and uncertainty are intrinsically linked*, and *sensitivity analysis* is a family of methods designed to measure the uncertainty of the model's output in relation to its input, so it's very useful in forecasting scenarios. We will also study two such methods: **Morris** for *factor prioritization* and **Sobol** for *factor fixing*, which involves cost sensitivity.

The following are the main topics we are going to cover:

- Assessing time series models with traditional interpretation methods
- Generating LSTM attributions with integrated gradients
- Computing global and local attributions with SHAP's `KernelExplainer`
- Identifying influential features with factor prioritization
- Quantifying uncertainty and cost sensitivity with factor fixing

Let's begin!

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `tensorflow`, `matplotlib`, `seaborn`, `alibi`, `distython`, `shap`, and `SALib` libraries. Instructions on how to install all these libraries can be found in this book's preface.

> The code for this chapter is located here: `https://packt.link/b6118`.

The mission

Highway traffic congestion is a problem that's affecting cities across the world. As the number of vehicles per capita steadily increases across the developing world with not enough road and parking infrastructure to keep up with it, congestion has been increasing at alarming levels. In the United States, the vehicle per capita statistic is among the highest in the world (838 per 1,000 people in 2019). For this reason, US cities represent 62 out of the 381 cities worldwide with at least a 15% congestion level.

Minneapolis is one such city (see *Figure 9.1*) where that threshold was recently surpassed and keeps rising. To put this metropolitan area into context, congestion levels are extremely severe at above 50%, but moderate-level congestion (15-25%) is already a warning sign of bad congestion to come. It's challenging to reverse congestion once it reaches 25% because any infrastructure improvement will be costly to implement without disrupting traffic even further. One of the worst congestion points is between the twin cities of Minneapolis and St. Paul throughout the Interstate 94 (I-94) highway, which congests alternate routes as commuters try to cut travel time. Knowing this, the mayors of both cities have obtained some federal funding to expand the highway:
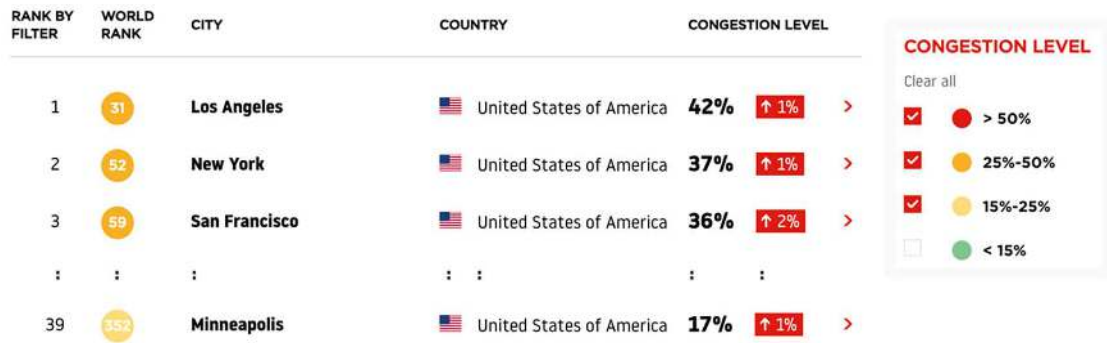
Figure 9.1: TomTom's 2019 traffic index for Minneapolis

The mayors want to be able to tout a completed expansion as a joint accomplishment to get reelected for a second term. However, they are well aware that a noisy, dirty, and obstructive expansion can be a big nuisance for commuters, so the construction project could backfire politically if it's not made nearly invisible. Therefore, they have stipulated that the construction company prefabricates as much as possible elsewhere and assembles only during low-volume hours. These hours have less than 1,500 vehicles per hour. They can also only work on one direction of the highway at a time and only block no more than half of its lanes when they are working on it. To ensure compliance with these stipulations, they will fine the company if they are blocking more than a quarter of the highway any time that volume is above this threshold, at a rate of $15 per vehicle.

In addition to that, if the construction crew are on-site blocking half the highway while traffic is over 1,500 vehicles per hour, it will cost them $5,000 a day. To put this into perspective, blocking during a typical peak hour could cost the construction company $67,000 per hour, plus the $5,000 daily fee! The local authorities will use **Automated Traffic Recorder** (**ATR**) stations along the route to monitor traffic volume, as well as local traffic police to register when lanes are getting blocked for construction.

The project has been planned as a 2-year construction project; the first year will expand the westbound lanes on the I-94 route, while the second will expand the eastbound lanes. The on-site portion of the construction will only occur from May through October because snow is less likely to delay construction during these months. Throughout the rest of the year, they will focus on pre-fabrication. They will attempt to work weekdays only because the workers union negotiated generous overtime pay for weekends. Therefore, weekend construction will happen only if there are significant delays. However, the union agreed to work holidays May through October for the same rate.

The construction company doesn't want to take any risks! Therefore, they need a model to predict traffic for the I-94 route and, more importantly, to understand what factors create uncertainty and possibly increase costs. They have hired a machine learning expert to do this: you!

The ATR data provided by the construction company includes hourly traffic volumes up to September 2018, as well as weather data at the same timescale. It only consists of the westbound lanes because that expansion will come first.

The approach

You have trained a stateful **Bidirectional LSTM** model with almost four years' worth of data (October 2012 – September 2016). You reserved the last year for testing (September 2017–2018) and the prior year to that for validation (September 2016 –2017). This made sense because the combined testing and validation datasets align well with the highway expansion project's expected conditions (March – November). You wondered about using other splitting schemes that leveraged only the data representative of these conditions, but you didn't want to reduce the training data so drastically, and maybe they might need it for winter predictions after all. A look-back window defines how much past data a time series model has access to. You chose 168 hours (1 week) as the look-back window size. Given the stateful nature of the model, as the model moves forward in the training data, it can learn daily and weekly seasonality, as well as some trends and patterns that can only be observed across several weeks. You also trained another two models. You have outlined the following steps to meet the client's expectations:

1. With *RMSE*, *regression plots*, *confusion matrices*, and much more, you will access the models' predictive performance and, more importantly, how the error is distributed.

2. With *integrated gradients*, you will understand if you took the best modeling strategy since it can help you visualize each of the model's pathways to a decision, and help you choose a model based on that.

3. With *SHAP's* `KernelExplainer` and a prediction approximation method, you will derive both a global and local understanding of what features matter to the chosen model.

4. With *Morris sensitivity analysis*, you will identify *factor prioritization*, which ranks factors (in other words, features) by how much they can drive output variability.

5. With *Sobol sensitivity analysis*, you will compute *factor fixing*, which helps determine what factors aren't influential. It does this by quantifying the input factors' contributions and interactions to the output's variability. With this, you can understand what factors may have the most effect on potential fines and costs, thus producing a variance-based cost-sensitivity analysis.

The preparation

You can find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/09/Traffic_compact1.ipynb`.

Loading the libraries

To run this example, you will need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate the dataset
- `tensorflow` to load the model
- `sklearn` (scikit-learn), `matplotlib`, `seaborn`, `alibi`, `distython`, `shap`, and `SALib` to create and visualize the interpretations

You should load all of them first:

```python
import math
import os
import mldatasets
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn import metrics

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.sequence import \ TimeseriesGenerator
from keras.utils import get_file

import matplotlib.pyplot as plt
from matplotlib.colors import TwoSlopeNorm
import seaborn as sns
from alibi.explainers import IntegratedGradients
from distython import HEOM
import shap
from SALib.sample import morris as ms
from SALib.analyze import morris as ma
from SALib.plotting import morris as mp
from SALib.sample.saltelli import sample as ss
from SALib.analyze.sobol import analyze as sa
from SALib.plotting.bar import plot as barplot
```

Let's check that TensorFlow has loaded the right version by using the `print(tf.__version__)` command. It should be 2.0 or above.

Understanding and preparing the data

In the following snippet, we are loading the data into a DataFrame called `traffic_df`. Please note that the `prepare=True` parameter is important because it performs necessary tasks such as subsetting the DataFrame to the required timeframe, since October 2015, some interpolation, correcting holidays, and performing one-hot encoding:

```python
traffic_df = mldatasets.load("traffic-volume-v2", prepare=True)
```

There should be over 52,000 records and 16 columns. We can verify this with `traffic_df.info()`. The output should check out. All the features are numerical and have no missing values, and the categorical features have already been one-hot encoded for us.

The data dictionary

There are only nine features, but they become 16 columns because of categorical encoding:

- `dow`: Ordinal; day of the week starting with Monday (between 0 and 6)
- `hr`: Ordinal; hour of the day (between 0 and 23)
- `temp`: Continuous; average temperature in Celsius (between-30 and 37)
- `rain_1h`: Continuous; mm of rainfall occurred in the hour (between 0 and 21)
- `snow_1h`: Continuous; cm of snow (when converted to liquid form) occurred in the hour (between 0 and 2.5)
- `cloud_coverage`: Continuous; percentage of cloud coverage (between 0 and 100)
- `is_holiday`: Binary; is the day a national or state holiday when it occurs Monday to Friday (1 for yes, 0 for no)?
- `traffic_volume`: Continuous; the target feature capturing traffic volume
- `weather`: Categorical; a short description of the weather during that hour (Clear | Clouds | Haze | Mist | Rain | Snow | Unknown | Other)

Understanding the data

The first step in understanding a time series problem is understanding the target variable. This is because it determines how you approach everything else, from data preparation to modeling. The target variable is likely to have a special relationship with time, such as a seasonal movement or a trend.

Understanding weeks

First, we can sample one 168-hour period from every season to understand the variance a bit better between days of the week, and then get an idea of how they could vary across seasons and holidays:

```
lb = 168
fig, (ax0,ax1,ax2,ax3) = plt.subplots(4,1, figsize=(15,8))
plt.subplots_adjust(top = 0.99, bottom=0.01, hspace=0.4)
traffic_df[(lb*160):(lb*161)].traffic_volume.plot(ax=ax0)
traffic_df[(lb*173):(lb*174)].traffic_volume.plot(ax=ax1)
traffic_df[(lb*186):(lb*187)].traffic_volume.plot(ax=ax2)
traffic_df[(lb*199):(lb*200)].traffic_volume.plot(ax=ax3)
```

The preceding code generates the plots shown in *Figure 9.2*. If you read them from left to right, you'll see that they all start with Wednesday and end with Tuesday of the following week. Every day of the week starts and ends at a low point, with a high point in between. Weekdays tend to have two peaks corresponding to morning and afternoon rush hour, while weekends only have one mid-afternoon bump:
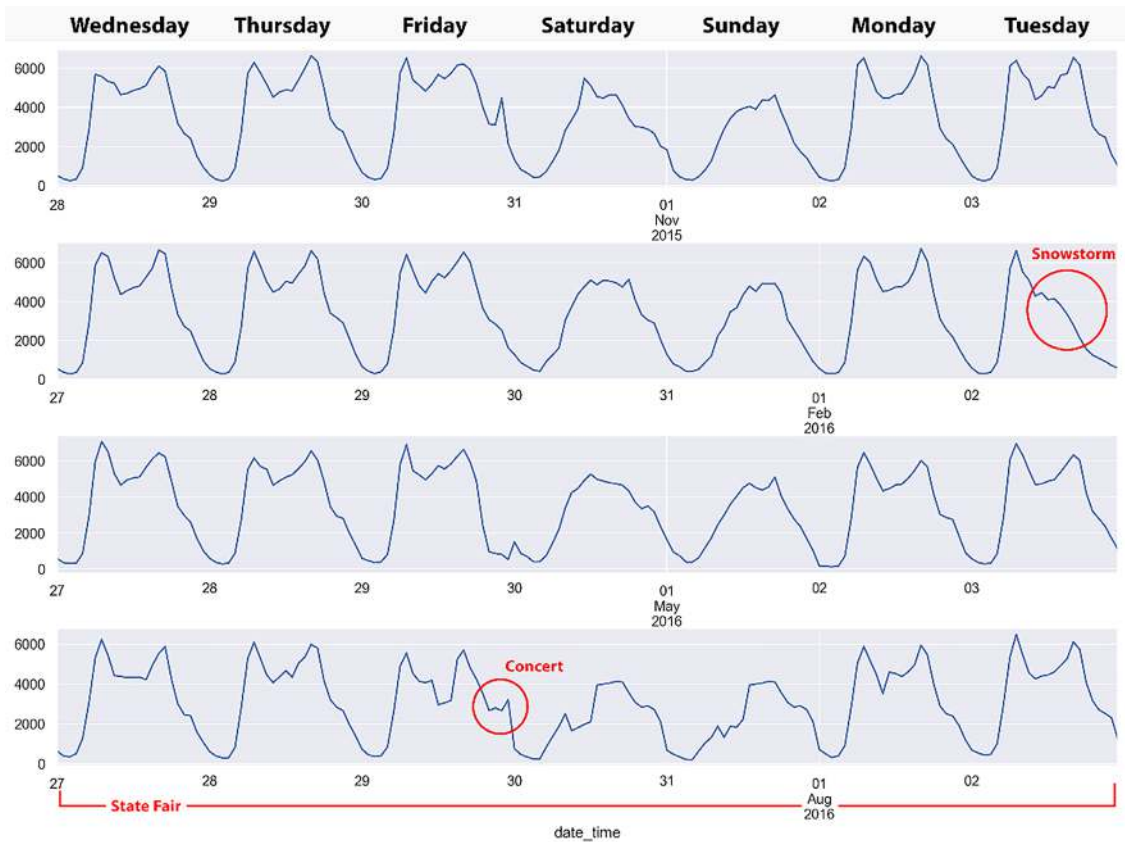
Figure 9.2: Several sample weekly periods for traffic_volume representing each season

There are some major outliers, such as Saturday October 31, which is basically Halloween and is not an official holiday. Also, February 2 (a Tuesday) was the beginning of a severe snowstorm, and the period in the late summer is much more chaotic than the other sample weeks. It turns out that in that year, the state fair occurred. Like Halloween, it's not a federal or a regional holiday, but it's important to note that the fairgrounds are located halfway between Minneapolis and St. Paul. You'll also notice that on Friday July 29, there's a midnight bump in traffic, which can be attributed to this being a big day for Minneapolis concerts.

Trying to explain these inconsistencies while comparing periods in your time series is a good exercise as it helps you figure out what variables to add to your model, or at least know what is missing. In our case, we know our `is_holiday` variable doesn't include days such as Halloween or the entire state fair week, nor do we have a variable for big music or sporting events. To produce a more robust model, it would be advisable to look for reliable external data sources and add more features that cover all these possibilities, not to mention validate the existing variables. For now, we will work with what we've got.

Understanding days

It is crucial for the highway expansion project to understand what traffic looks like for the average workday. The construction crew will be working on weekdays only (Monday to Friday) unless they experience delays, in which case they will also work weekends. We must also make a distinction between holidays and other weekdays because these are likely to be different.

To this end, we will create a DataFrame (`weekend_df`) and engineer a new column (`type_of_day`) that codes hours as being part of a "Holiday," "Weekday," or "Weekend." Then, we can group by this column and the `hr` column, and aggregate with `mean` and standard deviation (`std`). We can then `pivot` so that we have one column with the average and standard deviations traffic volumes for every `type_of_day` category, where the rows represent the hours of the day (`hr`). Then, we can plot the resulting DataFrame. We can create intervals with the standard deviations:

```python
weekend_df = traffic_df[
    ['hr', 'dow', 'is_holiday', 'traffic_volume']].copy()
weekend_df['type_of_day'] = np.where(
    weekend_df.is_holiday == 1,
    'Holiday',
    np.where(weekend_df.dow >= 5, 'Weekend', 'Weekday')
)
weekend_df = weekend_df.groupby(
['type_of_day','hr']) ['traffic_volume']
    .agg(['mean','std'])
    .reset_index()
    .pivot(index='hr', columns='type_of_day', values=['mean', 'std']
)
weekend_df.columns = [
    ''.join(col).strip().replace('mean','')\
    for col in weekend_df.columns.values
]
```

```
fig, ax = plt.subplots(figsize=(15,8))
weekend_df[['Holiday','Weekday','Weekend']].plot(ax=ax)
plt.fill_between(
    weekend_df.index,
    np.maximum(weekend_df.Weekday - 2 * weekend_df.std_Weekday, 0),
    weekend_df.Weekday + 2 * weekend_df.std_Weekday,
    color='darkorange',
    alpha=0.2
)
plt.fill_between(
    weekend_df.index,\
    np.maximum(weekend_df.Weekend - 2 * weekend_df.std_Weekend, 0),
    weekend_df.Weekend + 2 * weekend_df.std_Weekend,
    color='green',
    alpha=0.1
)
plt.fill_between(
    weekend_df.index,\
    np.maximum(weekend_df.Holiday - 2 * weekend_df.std_Holiday, 0),
    weekend_df.Holiday + 2 * weekend_df.std_Holiday,
    color='cornflowerblue',
    alpha=0.1
)
```

The preceding snippet results in the following plot. It represents the hourly average, but there's quite a bit of variation, which is why the construction company is proceeding with caution. There are horizontal lines that have been plotted representing each of the thresholds:

- 5,300 for full capacity.
- 2,650 for half-capacity, after which the construction company will get fined the daily amount specified.
- 1,500 is the no-construction threshold, after which the construction company will get fined the hourly amount specified.

They only want to work Monday to Friday during the hours that are typically below the 1,500 threshold. These five hours would be 11 p.m. (the day before) to 5 a.m. If they had to work weekends, this schedule would typically be delayed until 1 a.m. and end at 6 a.m. There's considerably less variance during weekdays, so it's understandable why the construction company is adamant about only working weekdays. During these hours, holidays appear to be similar to weekends, but holidays tend to vary even more than weekends, which is potentially even more problematic:
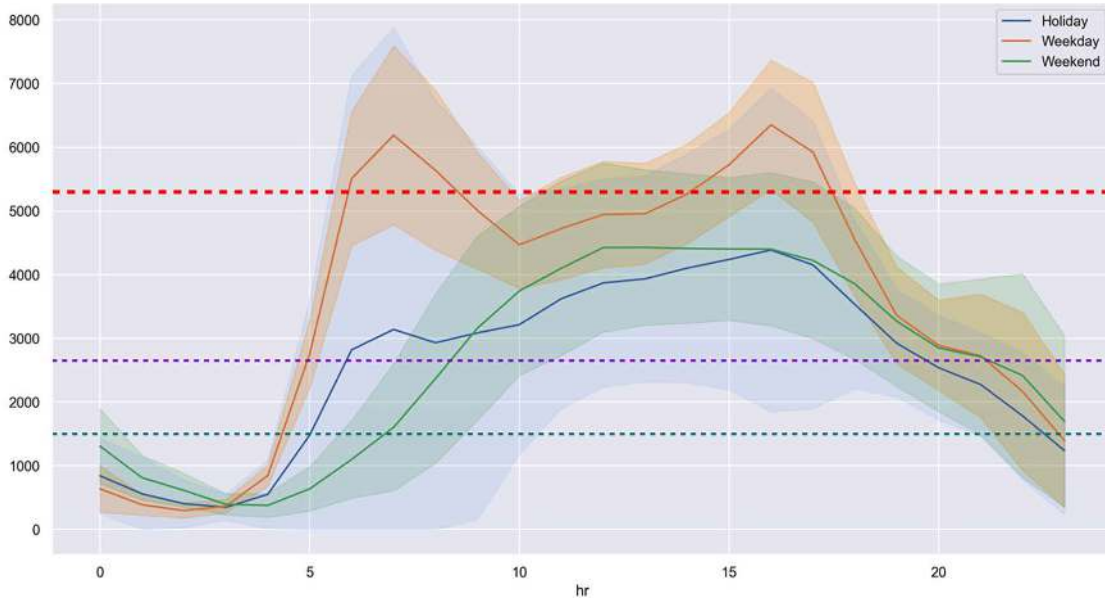
Figure 9.3: The average hourly traffic volume for holidays, weekdays, and weekends, with intervals

Usually, for a project like this, you would explore the predictor variables to the extent we have done with the target. This book is about model interpretation, so we will learn about the predictors by interpreting the models. But before we get to the models, we must prepare the data for them.

Data preparation

The first data preparation step is to split it into train, validation, and test sets. Please note that the test dataset comprises the last 52 weeks (2184 hours), while the validation dataset comprises the 52 weeks before that, so it starts at 4368 and ends 2184 hours before the last row of the DataFrame:

```
train = traffic_df[:-4368]
valid = traffic_df[-4368:-2184]
test = traffic_df[-2184:]
```

Now that the DataFrame has been split, we can plot it to ensure that its parts are split as intended. We can do so with the following code:

```
plt.plot(train.index.values, train.traffic_volume.values,
         label='train')
```

```
plt.plot(valid.index.values, valid.traffic_volume.values,
         label='validation')
plt.plot(test.index.values, test.traffic_volume.values,
         label='test')
plt.ylabel('Traffic Volume')
plt.legend()
```

The preceding code produces *Figure 9.4*. It shows that almost 4 years of data was allocated for the training dataset, and a year to validate and test each. We won't reference the validation dataset from this point on during this exercise because it was only instrumental during training to assess the model's predictive performance after every epoch.
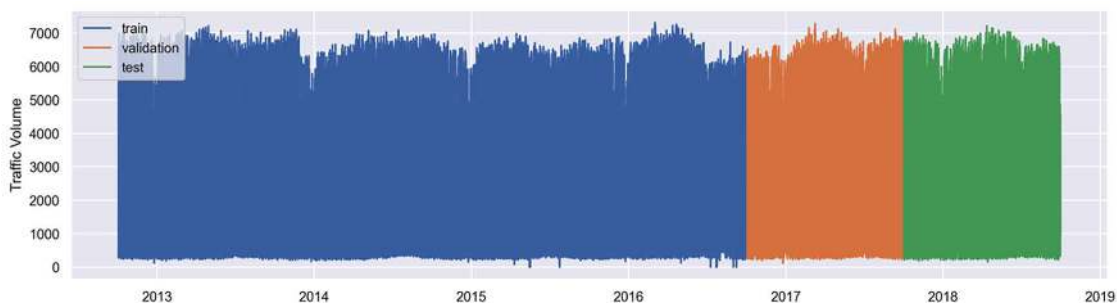
Figure 9.4: Time series split into train, validation, and test sets

The next step is to min-max normalize the data. We are doing this because larger values lead to slower learning for all neural networks in general and LSTMs are very prone to **exploding and vanishing gradients**. Relatively uniform and small numbers can help counter these problems. We will discuss this later in this chapter, but basically, the network becomes either numerically unstable or ineffective at reaching a global minimum.

We can min-max normalize with `MinMaxScaler` from the `scikit` package. For now, all we will do is `fit` the scaler so that we can use them whenever we need them. We will create a scaler for our target (`traffic_volume`) called `y_scaler` and another for the rest of the variables (`X_scaler`) with the entire dataset, so that transformations are consistent no matter what part you are using, be it `train`, `valid`, or `test`. All the `fit` process does is save the formula to make each variable fit between zero and one:

```
y_scaler = MinMaxScaler()
y_scaler.fit(traffic_df[['traffic_volume']])
X_scaler = MinMaxScaler()
X_scaler.fit(traffic_df.drop(['traffic_volume'], axis=1))
```

Now, we will `transform` both our train and test datasets with our scaler, creating *y* and *X* pairs for each:

```
y_train = y_scaler.transform(train[['traffic_volume']])
X_train = X_scaler.transform(train.drop(['traffic_volume'], axis=1))
y_test = y_scaler.transform(test[['traffic_volume']])
X_test = X_scaler.transform(test.drop(['traffic_volume'], axis=1))
```

However, for a time series model, the *y* and *X* pairs we created aren't useful because each observation is a timestep. And each timestep is more than the features that occur for that timestep, but to a certain extent what happens before it, called lags. For instance, say if we predict traffic based on 168 lagged observations, for every label, we will need the previous 168 hours of each feature. Therefore, you have to generate an array for every timestep, as well as its lags. Fortunately, `keras` has a function called `TimeseriesGenerator` that takes your *X* and *y* and produces a generator that feeds the data to your model. You must specify a certain `length`, which is the number of lagged observations (also known as the **lookback window**). The default `batch_size` is one, but we are using 24 because the client prefers to get forecasts 24 hours at a time, and also training and inference are much faster with a larger batch size.

Naturally, when you need to forecast tomorrow, you will need tomorrow's weather, but you can complete the timesteps with weather forecasts:

```
gen_train = TimeseriesGenerator(
    X_train,
    y_train,
    length=lb,
    batch_size=24
)
gen_test = TimeseriesGenerator(
    X_test,
    y_test,
    length=lb,
    batch_size=24
)
print(
    "gen_train:%sx%s→%s" % (len(gen_train),
    gen_train[0][0].shape, gen_train[0][1].shape)
)
print(
    "gen_test:%sx%s→%s" % (len(gen_test),
    gen_test[0][0].shape, gen_test[0][1].shape)
)
```

The preceding snippet outputs the dimensions of the training generator (`gen_train`) and the testing generator (`gen_test`), which use a length of 168 hours and a batch size of 24:

```
gen_train:  1454 ×    (24, 168, 15)    →    (24, 1)
gen_test:    357 ×    (24, 168, 15)    →    (24, 1)
```

Any model that was trained with a 1-week look-back window and 24-hour batch size will need this generator. Each generator is a list of tuples corresponding to each batch. Index 0 of this tuple is the *X* feature array, while index 1 is the *y* label array. Therefore, the first number output is the length of the list, which is the number of batches. The dimensions of the *X* and *y* array follow.

For instance, `gen_train` has 1,454 batches, and each batch has 24 timesteps, with a length of 168 and 15 features. The shape of the predicted labels expected from these 24 timesteps is (24,1).

Lastly, before moving forward with handling models and stochastic interpretation methods, let's attempt to make things more reproducible by initializing our random seeds:

```
rand = 9
os.environ['PYTHONHASHSEED']=str(rand)
tf.random.set_seed(rand)
np.random.seed(rand)
```

Loading the LSTM model

We can quickly load the model and output its summary like this:

```
model_name = 'LSTM_traffic_168_compact1.hdf5'
model_path = get_file(
    model_name,
    'https://github.com/PacktPublishing/Interpretable-\
    Machine-Learning-with-Python-2E/blob/main/models/{}?raw=true'
    .format(model_name)
)
lstm_traffic_mdl = keras.models.load_model(model_path)
lstm_traffic_mdl.summary()
```

As you can tell by the summary that's produced by the preceding snippet, the model starts with a **bidirectional LSTM** layer with an output of (24, 168). 24 corresponds to the batch size, while 168 means that there's not one but two 84-unit LSTMs going in opposite directions and meeting in the middle. It has a `dropout` of 10%, and then a `dense` layer with a single ReLu-activated unit. The ReLu ensures that all the predictions are over zero since negative traffic volume makes no sense:

```
Model: "LSTM_traffic_168_compact1"
_____
Layer (type)                 Output Shape              Param #
=================================================================
Bidir_LSTM (Bidirectional)   (24, 168)                 67200
_____
Dropout (Dropout)            (24, 168)                 0
_____
Dense (Dense)                (24, 1)                   169
=================================================================
Total params: 67,369
Trainable params: 67,369
Non-trainable params: 0
_____
```

Now, let's assess the `LSTM_traffic_168_compact1` model using traditional interpretation methods.

Assessing time series models with traditional interpretation methods

A time series regressor model can be evaluated as you would evaluate any regression model; that is, using metrics derived from the **mean squared error** or the **R-squared** score. There are, of course, cases in which you will need to use a metric with medians, logs, deviances, or absolute values. These models don't require any of this.

Using standard regression metrics

The `evaluate_reg_mdl` function can evaluate the model, output some standard regression metrics, and plot them. The parameters for this model are the fitted model (`lstm_traffic_mdl`), `X_train` (`gen_train`), `X_test` (`gen_test`), `y_train`, and `y_test`.

Optionally, we can specify a `y_scaler` so that the model is evaluated with the labels' inverse trans-formed, which makes the plot and **root mean square error** (**RMSE**) much easier to interpret. Another optional parameter that is very much necessary, in this case, is `y_truncate=True` because our `y_train` and `y_test` are of larger dimensions than the predicted labels. This discrepancy happens because the first prediction occurs several timesteps after the first timestep in the dataset due to the look-back window. Therefore, we would need to deduct these timesteps from `y_train` in order to match the length of `gen_train`.

We will now evaluate both models with the following code. To observe the prediction's progress as it happens, we will use `predopts={"verbose":1}`.

```
y_train_pred, y_test_pred, y_train, y_test =\
    mldatasets.evaluate_reg_mdl(lstm_traffic_mdl,
        gen_train,
        gen_test,
        y_train,
        y_test,
        scaler=y_scaler,
        y_truncate=True,
        predopts={"verbose":1}
)
```

The preceding snippet produced the plot and metrics shown in *Figure 9.5*. The *regression plot* is, essentially, a scatter plot of the observed versus predicted traffic volumes, fitted to a linear regression model to show how well they match. These plots show that the model tends to predict zero traffic when it's substantially higher. Besides that, there are a number of extreme outliers, but it fits relatively well with a test RMSE of 430 and only a slightly better train RMSE:

RMSE_train: 422.76   RMSE_test: 430.26   r2: 0.9528
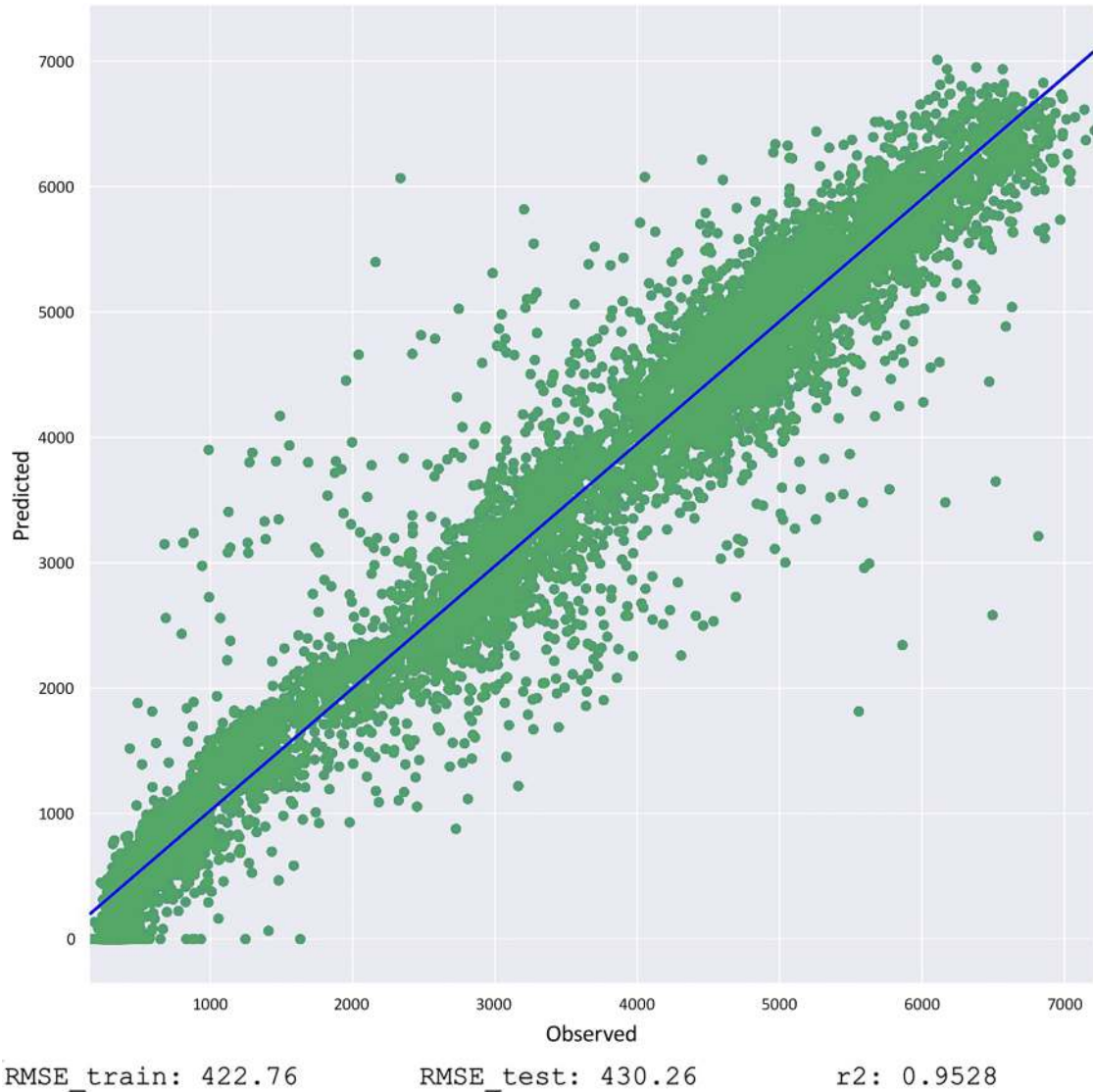
Figure 9.5: Predictive performance evaluations for the "LSTM_traffic_168_compact1" model

We can also evaluate the model by comparing observed versus predicted traffic. It would be helpful to break down the error by the hour and type of day too. To this end, we can create DataFrames with these values – one for each model. But first, we must truncate the DataFrame (`-y_test_pred.shape[0]`) so that it matches the length of the predictions array, and we won't need all the columns, so we are providing indexes for only those we are interested in: `traffic_volume` is #7 but we also will want `dow` (#0), `hr` (#1), and `is_holiday` (#6). We will rename `traffic_volume` to `actual_traffic` and create a new column called `predicted_traffic` with our predictions. Then, we will engineer a `type_of_day` column, as we did previously, which tells us if it's a holiday, weekday, or weekend. Finally, we can drop the `dow` and `is_holiday` columns since we won't need them:

```python
evaluate_df = test.iloc[-y_test_pred.shape[0]:,[0,1,6,7]]
    .rename(columns={'traffic_volume':'actual_traffic'}
)
evaluate_df['predicted_traffic'] = y_test_pred
evaluate_df['type_of_day'] = np.where(
    evaluate_df.is_holiday == 1,
    'Holiday',
    np.where(evaluate_df.dow >= 5,
    'Weekend', 'Weekday')
)
evaluate_df.drop(['dow','is_holiday'], axis=1, inplace=True)
```

You can quickly review the contents of the DataFrames by simply running a cell with `evaluate_df`. It should have 4 columns.

Predictive error aggregations

It may be that some days and times of day are more prone to predictive errors. To get a better sense of how these errors are distributed across time, we can plot RMSE on an hourly basis segmented by `type_of_day`. To do this, we must first define an `rmse` function and then group each of the models' evaluated DataFrames by `type_of_day` and `hr` and use the `apply` function to aggregate using the `rmse` function. We can then pivot to ensure that each `type_of_day` has a column with the RMSEs on an hourly basis. We can then average these columns and store them in a series:

```python
def rmse(g):
    rmse = np.sqrt(
    metrics.mean_squared_error(g['actual_traffic'],
                               g['predicted_traffic'])
    )
    return pd.Series({'rmse': rmse})
```

```
evaluate_by_hr_df = evaluate_df.groupby(['type_of_day', 'hr'])
    .apply(rmse).reset_index()
    .pivot(index='hr', columns='type_of_day', values='rmse')

mean_by_daytype_s = evaluate_by_hr_df.mean(axis=0)
```

Now that we have DataFrames with the hourly RMSEs for holidays, weekdays, and weekends, as well as the average for these "types" of day, we can plot them using the `evaluate_by_hr` DataFrame. We will also create dotted horizontal lines with the averages for each `type_of_day` from the `mean_by_daytype` pandas series:

```
evaluate_by_hr_df.plot()
ax = plt.gca()
ax.set_title('Hourly RMSE distribution', fontsize=16)
ax.set_ylim([0,2500])
ax.axhline(
    y=mean_by_daytype_s.Holiday,
    linewidth=2,
    color='cornflowerblue',
    dashes=(2,2)
)
ax.axhline(
    y=mean_by_daytype_s.Weekday,
    linewidth=2,
    color='darkorange',
    dashes=(2,2)
)
ax.axhline(
    y=mean_by_daytype_s.Weekend,
    linewidth=2,
    color='green',
    dashes=(2,2)
)
```

The preceding code generated the plot shown in *Figure 9.6*. As we can see, the model has a high RMSE for holidays. However, the model could be overestimating the traffic volume, and overestimating is not as bad as underestimating in this particular use case because underestimating can lead to annoying commuters with traffic delays and additional costs from fines:
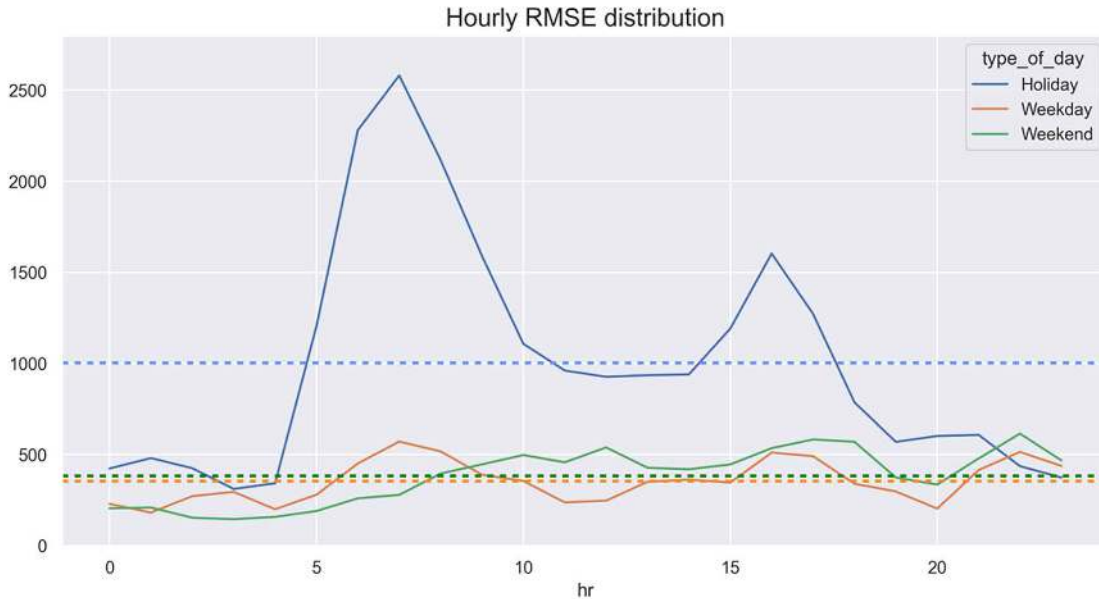
Figure 9.6: Hourly RMSE segmented by type_of_day for the "LSTM_traffic_168_compact1" model

Evaluating the model like a classification problem

Indeed, just like classification problems can have false positives and false negatives and one is more costly than the other, you can frame any regression problem with concepts such as underestimation and overestimation. This framing is especially useful when one is more costly than the other. If you have clearly defined thresholds, as we have for this project, you can evaluate any regression problem as you would a classification one. We will assess it with a confusion matrix with half-capacity and no-construction thresholds. To accomplish this, we can use `np.where` to get binary arrays for when the actuals and predictions surpass each threshold. We can then use the `compare_confusion_matrices` function to compare the confusion matrices for the model:

```
actual_over_half_cap = np.where(evaluate_df['actual_traffic'] >\
                                2650, 1, 0)
pred_over_half_cap = np.where(evaluate_df['predicted_traffic'] >\
                              2650, 1, 0)
actual_over_nc_thresh = np.where(evaluate_df['actual_traffic'] >\
                                 1500, 1, 0)
pred_over_nc_thresh = np.where(evaluate_df['predicted_traffic'] >\
                               1500, 1, 0)
```

```
mldatasets.compare_confusion_matrices(
    actual_over_half_cap,
    pred_over_half_cap,
    actual_over_nc_thresh,
    pred_over_nc_thresh,
    'Over Half-Capacity',
    'Over No-Construction Threshold'
)
```

The preceding snippet produced the confusion matrices shown in *Figure 9.7*.
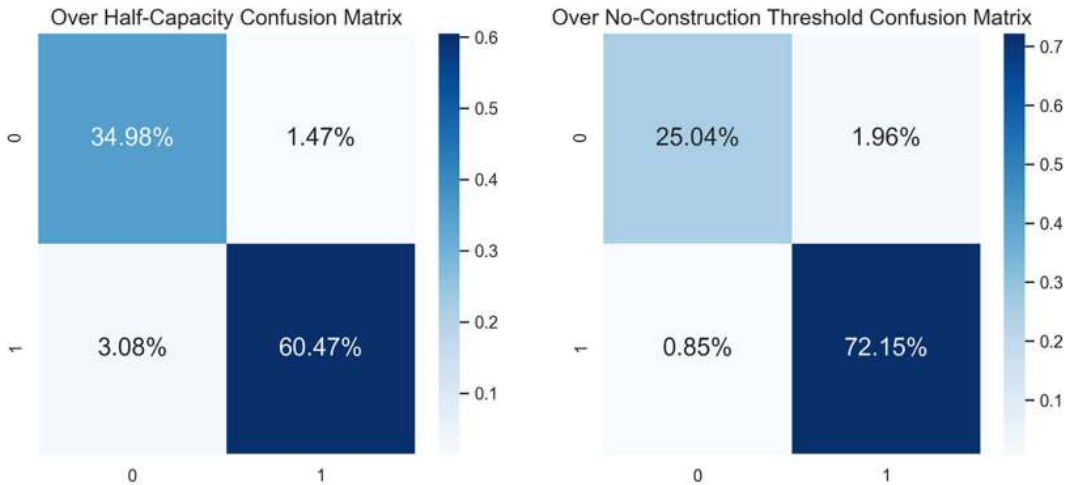
Figure 9.7: Confusion matrices for going over half and the no-construction threshold for the "LSTM_ traffic_168_compact1" model

We are most interested in the percentage of false negatives (bottom-left quadrant) because predicting no traffic beyond the threshold when, in fact, it did rise above it, will lead to a steep fine. On the other hand, the cost of false positives is in preemptively leaving the construction site when traffic didn't rise above the threshold after all. It's better to be safe than sorry, though! If you compare false negatives for the "no-construction" threshold (0.85%), it's less than a third of that of the half-capacity threshold (3.08%). Ultimately, what matters most is the no-construction threshold because the idea is to stop construction before it gets close to half-capacity.

Now that we have leveraged traditional methods to understand the model's decisions, let's move on to some more advanced model-agnostic methods.

Generating LSTM attributions with integrated gradients

We first learned about **integrated gradients** (**IG**) in *Chapter 7, Visualizing Convolutional Neural Networks*. Unlike the other gradient-based attribution methods studied in that chapter, path-integrated gradients is not contingent on convolutional layers, nor is it limited to classification problems.

In fact, since it computes the gradients of the output concerning the inputs averaged along the path, the input and output could be anything! It is common to use integrated gradients with **Convolutional Neural Networks** (**CNNs**) and **Recurrent Neural Networks** (**RNNs**), like the one we are interpreting in this chapter. Frankly, when you see an IG LSTM example online, it has an embedding layer and is an NLP classifier, but IG could be used very effectively for LSTMs that even process sounds or genetic data!

The integrated gradient explainer and the explainers that we will use moving forward can access any part of the traffic dataset. First, let's create a generator for all of it:

```
y_all = y_scaler.transform(traffic_df[['traffic_volume']])
X_all = X_scaler.transform(
    traffic_df.drop(['traffic_volume'], axis=1)
)
gen_all = TimeseriesGenerator(
    X_all, y_all, length=lb, batch_size=24
)
```

Integrated gradients is a local interpretation method. So, let's get a few sample "instances of interest" we can interpret. We know holidays may require specialized logic, so let's see if our model picks up on the importance of `is_holiday` for one example (`holiday_afternoon_s`). Also, mornings are a concern, especially mornings with a larger than average rush hour because of weather conditions, so we have one example for that (`peak_morning_s`). Lastly, a hot day might have more traffic, especially on a weekend (`hot_Saturday_s`):

```
X_df = traffic_df.drop(['traffic_volume'], axis=1 \
    ).reset_index(drop=True)
holiday_afternoon_s = X_df[
    (X_df.index >= 43800) & (X_df.dow==0) &\
    (X_df.hr==16) &(X_df.is_holiday==1)
].tail(1)

peak_morning_s = X_df[
    (X_df.index >= 43800) & (X_df.dow==2) &\
    (X_df.hr==8) & (X_df.weather_Clouds==1) & (X_df.temp<20)
].tail(1)

hot_Saturday_s = X_df[
    (X_df.index >= 43800) & (X_df.dow==5) &\
    (X_df.hr==12) & (X_df.temp>29) & (X_df.weather_Clear==1)
].tail(1)
```

Now that we have created some instances, let's instantiate our explainers. `IntegratedGradients` from the `alibi` package only requires a deep learning model, but it is recommended to set a number of steps (`n_steps`) for the integral approximation and `internal_batch_size`. We will instantiate an explainer for our model:

```
ig = IntegratedGradients(
    lstm_traffic_mdl, n_steps=25, internal_batch_size=24
)
```

Before we iterate our samples and the explainers, it is important to realize how we need to input the sample to the explainer because it will need a batch of 24. To this end, we will have to get the index of the sample once we've deducted the lookback window (`nidx`). Then, you can obtain the batch for this sample from the generator (`gen_all`). Each batch includes 24 timesteps, so you floor `nidx` by 24 (`nidx//24`) to get the batch's position for that sample. Once you've got the batch for the sample (`batch_X`) and printed the shape (`24, 168, 15`), it shouldn't surprise you that the first number is 24. Of course, we will need to get the index of the sample within the batch (`nidx%24`) to obtain the data for that sample:

```
nidx = holiday_afternoon_s.index.tolist()[0] - lb
batch_X = gen_all[nidx//24][0]
print(batch_X.shape)
```

The `for` loop will use the previously explained method to locate the batch for the sample (`batch_X`). This `batch_X` is inputted into the `explain` function. This is because this is a regression problem and there's no target class; that is, `target=None`. Once the explanation is produced, the `attributions` property will have the attributions for the entire batch. We can only obtain this for the sample and `transpose` it to produce an image that has this shape: (`15, lb`). The rest of the code in the `for` loop simply obtains the labels to use in the tick marks and then plots an image stretched out to fit the dimensions of our `figure`, along with its labels:

```
samples = [holiday_afternoon_s, peak_morning_s, hot_saturday_s]
sample_names = ['Holiday Afternoon', 'Peak Morning' , 'Hot Saturday']
for s in range(len(samples)):
    nidx = samples[s].index.tolist()[0] - lb
    batch_X = gen_all[nidx//24][0]

    explanation = ig.explain(batch_X, target=None)
    attributions = explanation.attributions[0]
    attribution_img = np.transpose(attributions[nidx%24,:,:])

    end_date = traffic_df.iloc[samples[s].index
        ].index.to_pydatetime()[0]
    date_range = pd.date_range(
        end=end_date, periods=8, freq='1D').to_pydatetime().tolist()
    columns = samples[s].columns.tolist()

    plt.title(
        'Integrated Gradient Attribution Map for "{}"'.\
        format(sample_names[s], lb), fontsize=16
    )
```

```
    divnorm = TwoSlopeNorm(
        vmin=attribution_img.min(),
        vcenter=0,
        vmax=attribution_img.max()
    )
    plt.imshow(
        attribution_img,
        interpolation='nearest' ,
        aspect='auto',
        cmap='coolwarm_r',
        norm=divnorm
)
    plt.xticks(np.linspace(0,lb,8).astype(int), labels=date_range)
    plt.yticks([*range(15)], labels=columns)
    plt.colorbar(pad=0.01,fraction=0.02,anchor=(1.0,0.0))
    plt.show()
```

The preceding code will generate the plots shown in *Figure 9.8*. On the *y*-axis, you can see the variable names, while on the *x*-axis, you can see the dates corresponding to the lookback window for the sample in question. The rightmost part of the *x*-axis is the sample's date, and as you move left, you go backward in time. For instance, the holiday afternoon sample was 4 p.m. September 3 and there is one week's worth of lookback, so each tick mark backward is a day before that date.

Figure 9.8: Annotated integrated gradients attribution map for all samples for the "LSTM_traffic_168_compact1" model

You can tell by the intensity in the attribution maps in *Figure 9.8* which hour/variables mattered for the prediction. The color bar to the right of each attribution map can serve as a key. Negative numbers in red correspond to a negative correlation, while positive numbers in blue correspond to a positive correlation. However, something that is pretty evident is the tendency for intensities to fade as each map goes backward in time. Since it's bidirectional, this happens from both ends. What is surprising is how fast this happens.

Let's start from the bottom. For "Hot Saturday," day of the week, hour, temperature, and clear weather play an important role in this prediction increasingly as you get closer to the predicted time (midday Saturday). The day started cooler, which explains how there's a patch of red before the blue in the temperature feature.

For "Peak Morning," attributions make sense since it was clear after it had been previously rainy and cloudy, which caused the rush hour to peak quickly rather than increase slowly. To a certain degree, the LSTM has learned that only recent weather matters – no more than two or three days' worth. However, that is not the only reason the integrated gradients fade. They also fade because of the **vanishing gradient problem**. This problem occurs during backpropagation because the gradient values are multiplied by the weight matrices in each step, so gradients can exponentially decrease to zero.

LSTMs are organized in a very long sequence, making the network ever more ineffective at capturing dependencies in the long term. Fortunately, these LSTMs are **stateful**, which means they string batches in a sequence by leveraging states from the previous batch. **Statefulness** ensures learning from a long sequence, despite vanishing gradients. This is why when we observe the attribution map for "Holiday Afternoon," there are negative attributions for is_holiday, which makes sense to anticipate no rush hour. It turns out September 3 (Labor Day) is nearly two months after the previous holiday (Independence Day), which is a more festive holiday. Is it possible that the model is picking up on these patterns?

We could try subcategorizing holidays by their traffic patterns to see if that helps the model identify them. We could also make rolling aggregations of previous weather conditions to make it easier for the model to pick up on recent weather patterns. Weather patterns span hours, so it is intuitive to aggregate, not to mention easier to interpret. Interpretation methods can point us in the right direction as to how to improve models, and there's certainly a lot of room for improvement.

Next, we will take a stab at a permutation-based method!

Computing global and local attributions with SHAP's KernelExplainer

Permutation methods make changes to the input to assess how much difference they will make to a model's output. We first discussed this in *Chapter 4*, *Global Model-Agnostic interpretation methods*, but if you recall, there's a coalitional framework to perform these permutations that will produce the average marginal contribution for each feature across different coalitions of features. This process's outcome is **Shapley values**, which have essential mathematical properties such as additivity and symmetry. Unfortunately, Shapley values are costly to compute for datasets that aren't small, so the SHAP library has approximation methods. One of these methods is `KernelExplainer`, which we also explained in *Chapter 4* and used in *Chapter 5*, *Local Model-Agnostic Interpretation Methods*. It approximates the Shapley values with a weighted local linear regression, just like LIME does.

Why use KernelExplainer?

We have a deep learning model, so why aren't we using SHAP's `DeepExplainer` as we did with the CNN in *Chapter 7*, *Visualizing Convolutional Neural Networks*? DeepExplainer adapted the DeepLIFT algorithm to approximate the Shapley values. It works very well with any feedforward network that's used for tabular data, CNNs, and RNNs with an embedding layer, such as those used for an NLP classifier, or even to detect genomic sequences. It gets trickier for multivariate time series because DeepExplainer doesn't know what to do with the input's three-dimensional array. Even if it did, it includes data for previous timesteps, so you cannot permute one timestep without considering the previous ones. For instance, if the permutation dictates that the temperature is five degrees lower, shouldn't that affect all the previous timestep's temperatures up to a certain number of hours? And what if it's 20 degrees lower? Doesn't that mean it's likely in a different season with entirely different weather – perhaps more clouds and snow as well?

SHAP's `KernelExplainer` can receive any arbitrary black box `predict` function. It also makes assumptions about the input dimensions. Fortunately, we can change the input data before it permutes it, making it seem to the `KernelExplainer` like it's dealing with a tabular dataset. The arbitrary `predict` function doesn't have to simply call the model's `predict` function – it can change data both on the way in and on the way out!

Defining a strategy to get it to work with a multivariate time series model

To mimic likely past weather patterns based on the permutated input data, we could create a generative model or something to that effect. This strategy will help us to generate a variety of past timesteps that fit the permutated timestep, as well as to generate images for a specific class. Although this would likely lead to more accurate predictions, we won't use this strategy because it's incredibly time-consuming.

Instead, we will find the time series data that best suits the permutated input with existing examples from our `gen_all` generator. There are distance metrics we can use to find the one that is closest to the permutated input. However, we must place some guardrails because if the permutation is for a Saturday at 5 a.m. with a temperature of 27 degrees Celsius and 90 percent cloud coverage, the closest observation to this one could be on a Friday at 7 a.m., but regardless of the weather traffic, it would be completely different. Therefore, we can implement a filter function that ensures that it only finds the closest observations for the same `dow`, `is_holiday`, and `hr`. The filter function can also clean up the permutated sample to remove or modify anything nonsensical for the model, such as a continuous value for a categorical feature:
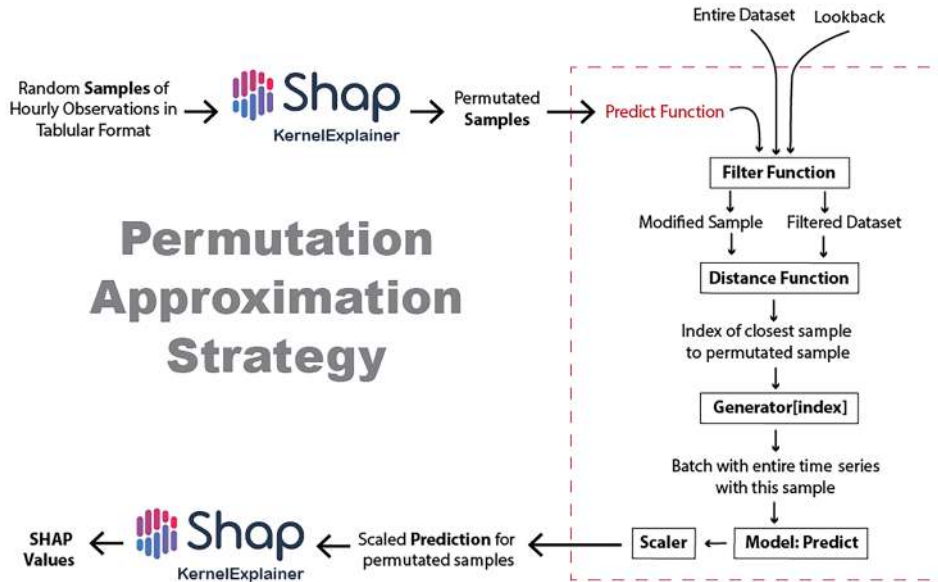
Figure 9.9: Permutation approximation strategy

Figure 9.9 depicts the rest of the process where it uses a distance function to find the closest observation to the modified permutated sample. This function returns the closest observation index, but the model can't predict on singular observations (or timesteps), so it requires its past hourly history up to the lookback window. For this reason, it retrieves the right batch from the generator and makes a prediction on that, but the predictions will be on a different scale, so they need to be inverse transformed with `y_scaler`. Once the `predict` function has iterated through all the samples and made predictions for them and rescaled them, it sends them back to the `KernelExplainer`, which outputs their SHAP values.

Laying the groundwork for the permutation approximation strategy

You can define a custom filter function (`filt_fn`). It takes a `pandas` DataFrame with the entire dataset (`X_df`) you want to filter from, as well as the permutated sample (`x`) for filtering and the length of the `lookback` window.

The function can also modify the permutated sample. In this case, we have to do this because so many features of the model are discrete, but the permutation process makes them continuous. As we mentioned previously, all the filtering does is protect the distance function from finding a nonsensical closest sample to the permutated sample by limiting the options:

```python
def filt_fn(X_df, x, lookback):
    x_ = x.copy()
    x_[0] = round(x_[0]) #round dow
    x_[1] = round(x_[1]) #round hr
    x_[6] = round(x_[6]) #round is_holiday
    if x_[1] < 0:#if hr < 0
        x_[1] = 24 + x_[1]
        x_[0] = x_[0] - 1   #make it previous day
    if x_[0] < 0:#if dow < 0
        x_[0] = 7 + x_[0] #make it previous week
        X_filt_df = X_df[
            (X_df.index >= lookback) & (X_df.dow==x_[0]) &\
            (X_df.hr==x_[1]) & (X_df.is_holiday==x_[6]) &\
            (X_df.temp-5<=x_[2]) & (X_df.temp+5>=x_[2])
        ]
    return X_filt_df, x_
```

If you refer to *Figure 9.9*, after the filter function, the next thing we ought to define is the distance function. We could use any standard distance function accepted by `scipy.spatial.distance.cdist`, such as "Euclidean," "cosine," or "Hamming." The problem with these standard distance functions is that they either work well with continuous or discrete variables but not both. We have both in this dataset!

Fortunately, some alternatives exist that can handle both, such as **Heterogeneous Euclidean-Overlap Metric** (**HEOM**) and **Heterogeneous Value Difference Metric** (**HVDM**). Both methods apply different distance metrics, depending on the nature of the variable. HEOM uses a normalized Euclidean ($\sqrt{(a-b)^2}$) for continuous and , for discrete, "overlap" distance; that is, a distance of zero if the same and one otherwise.

HVDM is more complicated because, for continuous variables, it's the absolute distance between both values, divided by the standard deviation of the feature in question times four ($|a - b|$ / 4*Standard Deviation of the Feature), which is a great distance metric for handling outliers. For discrete variables, it uses a normalized **value difference metric**, which is based on the difference between the conditional probability of both values.

Even though HVDM is better than HEOM for datasets with many continuous values, it is overkill in this case. Once the dataset has been filtered by day of the week (`dow`) and hour (`hr`), the remaining discrete features are all binary, so "overlap" distance is ideal, and for the three remaining continuous features (`temp`, `rain_1h`, `snow_1h`, and `cloud_coverage`), Euclidean distance should suffice. `distython` has an `HEOM` distance method, and all it requires is a background dataset (`X_df.values`) and the indexes of the categorical features (`cat_idxs`). We can programmatically identify these features with an `np.where` command.

If you want to verify that these are the right ones, run `print(cat_idxs)` in a cell. Only indexes 2, 3, 4, and 5 should be omitted:

```
cat_idxs = np.where(traffic_df.drop(['traffic_volume'],\
                                    axis=1).dtypes != np.float64)[0]
heom_dist = HEOM(X_df.values, cat_idxs)
print(cat_idxs)
```

Now, we can create a `lambda` function that puts everything depicted in *Figure 9.9* together. It leverages a function called `approx_predict_ts` that takes care of the entire pipeline. It takes our filter function (`filt_fn`), distance function (`heom_dist.heom`), generator (`gen_all`), and fitted model (`lstm_traffic_mdl`), and chains them together, as described in *Figure 9.9*. It also scales the data with our scalers (`X_scaler` and `y_scaler`). Distance is computed on transformed features for higher accuracy, and the predictions are reverse transformed on the way out:

```
predict_fn = lambda X: mldatasets.approx_predict_ts(
    X, X_df,
    gen_all,
    lstm_traffic_mdl,
    dist_metric=heom_dist.heom,
    lookback=lookback,
    filt_fn=filt_fn,
    X_scaler=X_scaler,
    y_scaler=y_scaler
)
```

We can now use the prediction function with `KernelExplainer`, but it should be done on samples that are most representative of the construction crew's expected working conditions; that is, they plan to work March through November only, preferably on weekdays and in low-traffic hours. To this end, let's create a DataFrame (`working_season_df`) that only includes these months and initializes a `KernelExplainer` with `predict_fn` and the k-means of the DataFrame as background data:

```
working_season_df =\
    traffic_df[lookback:].drop(['traffic_volume'], axis=1).copy()
working_season_df =\
    working_season_df[(working_season_df.index.month >= 3) &\
                      (working_season_df.index.month <= 11)]
explainer = shap.KernelExplainer(
    predict_fn, shap.kmeans(working_season_df.values, 24)
)
```

We can now produce SHAP values for a random set of observations of the `working_season_df` Data-Frame.

Computing the SHAP values

We will sample 48 observations from it. `KernelExplainer` is rather slow, especially when it's using our approximation method. To get an optimal global interpretation, it is best to use a high number of observations but also a high `nsamples`, which is the number of times we need to reevaluate the model when explaining each prediction. Unfortunately, having 50 of each would cause the explainer to take many hours to run, depending on your available compute, so we will use `nsamples=10`. You can look at SHAP's progress bar and adjust it accordingly. Once it's done, it will produce a feature importance `summary_plot` containing the SHAP values:

```
X_samp_df = working_season_df.sample(80, random_state=rand)
shap_values = explainer.shap_values(X_samp_df, nsamples=10)
shap.summary_plot(shap_values, X_samp_df)
```

The preceding code plots the summary shown in the following graph. Not surprisingly, `hr` and `dow` are the most important features, followed by some weather features. Strangely enough, temperature and rain don't seem to weigh in on the predictions, but late spring through fall may not be a significant factor. Or maybe more observations and a higher `nsample` will yield a better global interpretation:
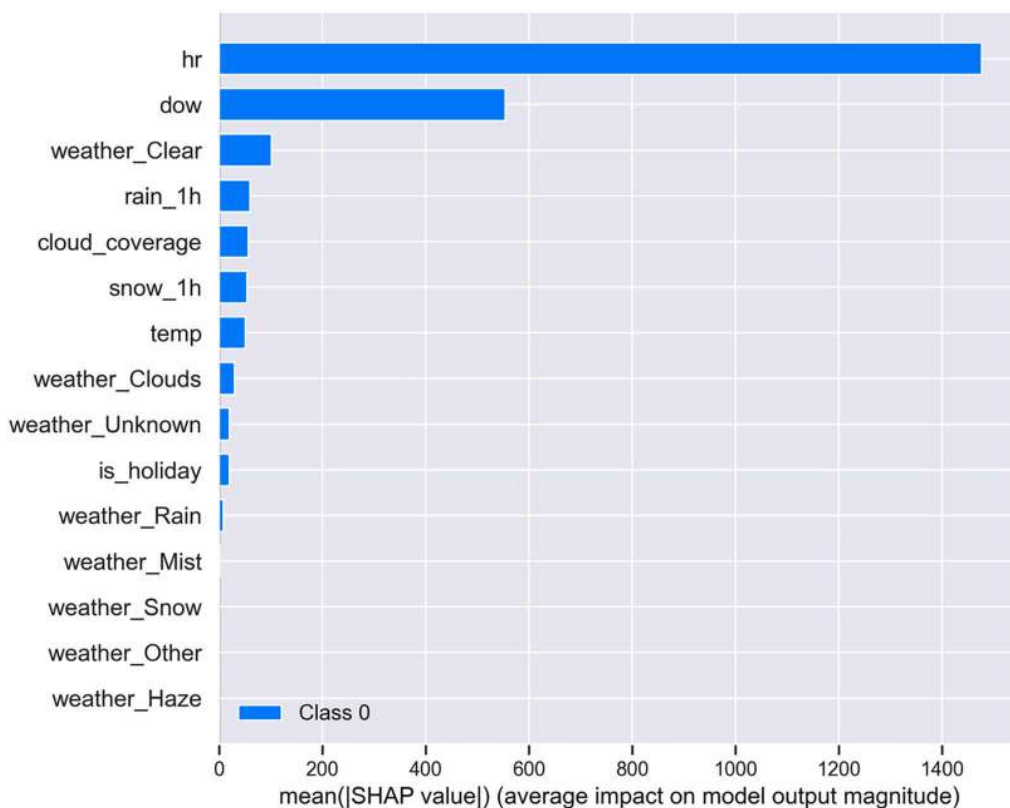
Figure 9.10: SHAP summary plot based on the SHAP values produced by 48 sampled observations

We can do the same with the instances of interest we chose in the previous section for local interpretations. Let's iterate through all these data points. Then, we can produce a single `shap_values`, but this time with `nsamples=80`, and then generate a `force_plot` for each one:

```
for s in range(len(samples)):
    print('Local Force Plot for "{}"'.format(sample_names[s]))
    shap_values_single = explainer.shap_values(
        datapoints[i], nsamples=80)
    shap.force_plot(
    explainer.expected_value,
    shap_values_single[0],
    samples[s],
    matplotlib=True
)
    plt.show()
```

The preceding code generates the plots shown in *Figure 9.11*. "Holiday afternoon" has the hour (`hr=16`) pushing toward a higher prediction, while the fact that it's a Monday (`dow=0`) and a holiday (`is_holiday=1`) is a driving force in the opposite direction. On the other hand, "Peak Morning" is mostly peak due to the hour (`hr=8.0`), but it has a high `cloud_coverage`, affirmative `weather_Clouds`, and yet no rain (`rain_1h=0.0`). Lastly, "Hot Saturday" has the day of the week (`dow=5`) pushing for a lower value, but the abnormally high value is mostly due to it being midday with no rain and clouds. Strangely, higher than normal temperature is not one of the factors:

Figure 9.11: Force plots generated with SHAP values using nsamples=80 for a Holiday Afternoon, Peak Morning, and Hot Saturday

With SHAP's game theory-based approach, we can gauge how many permutations for the existing observations marginally vary the predicted outcome across many possible coalitions of features. However, this approach can be very limiting because our background data's existing variance shapes our understanding of outcome variance.

In the real world, *variability is often determined by what is NOT represented in your data – but infinitesimally plausible*. For instance, reaching 25°C (77°F) before 5 a.m. in a Minneapolis summer is not a common occurrence, but with global warming, it could become frequent, so we would want to simulate how it could impact traffic patterns. Forecasting models are particularly prone to risk, so simulating is a crucial interpretation component to assess this uncertainty. A better understanding of uncertainty can yield more robust models and directly inform decisions. Next, we will discuss how we can produce simulations with sensitivity analysis methods.

Identifying influential features with factor prioritization

The **Morris method** is one of several global sensitivity analysis methods that range from simple **Fractional factorial** to complicated **Monte Carlo filtering**. Morris is somewhere on this spectrum, falling into two categories. It uses **one-at-a-time sampling**, which means that only one value changes between consecutive simulations. It's also an **Elementary Effects** (**EE**) method, which means that it doesn't quantify the exact effect of a factor in a model but rather gauges its importance and relationship with other factors. By the way, **factor** is just another word for a feature or variable that's commonly used in applied statistics. To be consistent with the related theory, we will use this word in this and the next section.

Another property of Morris is that it's less computationally expensive than the variance-based methods we will study next. It can provide more insights than simpler and less costly methods such as regression-, derivative-, or factorial-based ones. It can't quantify effects precisely but can identify those with negligible or interaction effects, making it an ideal method for screening factors when the number of factors is low. Screening is also known as **factor prioritization** because it can prioritize your factors by how they are classified.

Computing Morris sensitivity indices

The Morris method derives a distribution of elementary effects that it associates with an individual factor. Each EE distribution has a mean (μ) and a standard deviation (σ). These two statistics are what help map the factors into different classifications. The mean could be negative when the model is non-monotonic, so a Morris method variation adjusts for this with absolute values (μ^*) so that it is more manageable to interpret. We will use this variation here.

Now, let's limit the scope of this problem to make it more manageable. The traffic uncertainties the construction crew will face will be ongoing from May to October, Monday to Friday, from 11 p.m. to 5 a.m. Therefore, we can take the `working_season_df` DataFrame and subset it further to produce a working hours one (`working_hrs_df`) that we can `describe`. We will include the 1%, 50%, and 99% percentiles to understand where the median and outliers lie:

```
working_hrs_df = working_season_df[
    (working_season_df.dow < 5)
    & ((working_season_df.hr < 5) | (working_season_df.hr > 22))
]
working_hrs_df.describe(percentiles=[.01,.5,.99]).transpose()
```

The preceding code produced the table in *Figure 9.12*. We can use this table to extract the ranges we will use for our features in the simulation. Typically, we would use plausible values that have exceeded the existing maximums or minimums. For most models, any feature value can be increased or decreased beyond its known limits, and since the model learned a monotonic relationship, it can infer a realistic outcome. For instance, it might learn that rain beyond a certain point will increasingly diminish traffic. Then, say you want to simulate a severe flood with, say, 30 mm of rain per hour; it can accurately predict no traffic:

	count	mean	std	min	1%	50%	99%	max
dow	7026.00	2.01	1.41	0.00	0.00	2.00	4.00	4.00
hr	7026.00	5.50	7.93	0.00	0.00	2.50	23.00	23.00
temp	7026.00	10.83	9.14	-24.19	-12.31	12.60	25.29	30.25
rain_1h	7026.00	0.12	0.67	0.00	0.00	0.00	3.10	20.40
snow_1h	7026.00	0.01	0.06	0.00	0.00	0.00	0.28	1.54
cloud_coverage	7026.00	38.54	37.77	0.00	0.00	27.60	100.00	100.00
is_holiday	7026.00	0.04	0.18	0.00	0.00	0.00	1.00	1.00
weather_Clear	7026.00	0.32	0.46	0.00	0.00	0.00	1.00	1.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
weather_Snow	7026.00	0.02	0.13	0.00	0.00	0.00	0.00	1.00
weather_Unknown	7026.00	0.21	0.41	0.00	0.00	0.00	1.00	1.00

Figure 9.12: Summary statistics for the period that the construction crew plans to work through

However, because we are using a prediction approximation method that samples from historical values, we are limited to how far we can push the boundaries outside of the known. For this reason, we will use the 1% and 99% percentile values as our limits. We should note that this is an important caveat for any findings, especially for features that could plausibly extend beyond these limits, such as `temp`, `rain_1h`, and `snow_1h`.

Another thing to note from the summary of *Figure 9.12* is that many weather-related binary features are very sparse. You can tell by their extremely low mean. Each factor that's added to the sensitivity analysis simulation slows it down, so we will only take the top three; that is, `weather_Clear`, `weather_Clouds`, and `weather_Rain`. These factors are specified along with the other six factors in a "problem" dictionary (`morris_problem`), which has their corresponding `names`, `bounds`, and `groups`. Now, `bounds` is critical because it denotes what ranges of values will be simulated for each factor. We will use [0,4] (Monday to Friday) for `dow` and [-1,4] (11 p.m. to 4 a.m.) for `hr`. The filter function automatically translates negative hours into hours from the day before so that -1 on a Tuesday is equivalent to 23 on a Monday. The rest of the bounds were informed by the percentiles. Note that `groups` all have factors in the same group, except for the three weather ones:

```
morris_problem = {
    # There are nine variables
    'num_vars': 10,
    # These are their names
    'names': ['dow', 'hr', 'temp', 'rain_1h', 'snow_1h',\
              'cloud_coverage', 'is_holiday', 'weather_Clear',\
              'weather_Clouds', 'weather_Rain'],
    # Plausible ranges over which we'll move the variables
    'bounds': [
        [0, 4], # dow Monday - Firday
        [-1, 4], # hr
        [-12, 25.], # temp (C)
        [0., 3.1], # rain_1h
        [0., .3], # snow_1h
        [0., 100.], # cloud_coverage
        [0, 1], # is_holiday
        [0, 1], # weather_Clear
        [0, 1], # weather_Clouds
        [0, 1] # weather_Rain
    ],
    # Only weather is grouped together
    'groups': ['dow', 'hr', 'temp', 'rain_1h', 'snow_1h',\
               'cloud_coverage', 'is_holiday', 'weather', 'weather',\
               'weather']
}
```

Once the dictionary has been defined, we can generate Morris method samples with `SALib's sample` method. In addition to the dictionary, it takes a number of trajectories (256) and levels (`num_levels=4`). The method uses a grid with factors and levels to construct the trajectories for which inputs are randomly moved **one at a time** (**OAT**). What is important to heed here is that more levels add more resolution to this grid, potentially making for a better analysis. However, this can be very time-consuming. It's better to start with a ratio between the number of trajectories and levels of 25:1 or higher.

Then, you can decrease this ratio progressively. In other words, if you have enough compute, you can make `num_levels` match the number of trajectories, but if you have this much compute available, you could try `optimal_trajectories=True`. However, given that we have groups, `local_optimization` would have to be `False`. The output of `sample` is an array that is one column for each factor and $(G + 1) \times T$ rows (where G is the number of groups and T is the number of trajectories). We have eight groups and 256 trajectories, so `print` should output a shape of 2,304 rows and 10 columns:

```
morris_sample = ms.sample(morris_problem, 256,\
                          num_levels=4, seed=rand)
print(morris_sample.shape)
```

Given that the `predict` function will only work with 15 factors, we should modify the samples to fill the remaining five factors with zeroes. We use zeroes because that is the median value for these features. Medians are least likely to increase traffic, but you ought to tailor your default values on a case-by-case basis. If you recall our **Cardiovascular Disease** (**CVD**) example from *Chapter 2*, *Key Concepts of Interpretability*, the feature value that would increase CVD risk was sometimes the minimum or maximum.

The `np.hstack` function can concatenate the array horizontally so that three zero factors follow the samples for the first eight factors. Then, there's a lonely ninth sample factor corresponding to `weather_Rain`, followed by two zero factors. The resulting array should have the same number of rows as before but 15 columns:

```
morris_sample_mod = np.hstack(
    (
        morris_sample[:,0:9],
        np.zeros((morris_sample.shape[0],3)),
        morris_sample[:,9:10],
        np.zeros((morris_sample.shape[0],2))
    )
)
print(morris_sample_mod.shape)
```

The `numpy` array known as `morris_sample_mod` now has the Morris samples in a shape that can be understood by our `predict` function. If this was a model that had been trained on a tabular dataset, we could just leverage the model's `predict` function. However, just as we did with SHAP, we have to use the approximation method. This time, we won't use `predict_fn` because we want to set one additional option, `progress_bar=True`, in `approx_predict_ts`. Everything else will remain the same. The progress bar will come in handy because this should take a while. Run the cell and take a coffee break:

```
morris_preds = mldatasets.approx_predict_ts(
    morris_sample_mod,
    X_df,
    gen_all,
```

```
    lstm_traffic_mdl,
    filt_fn=filt_fn,
    dist_metric=heom_dist.heom,
    lookback=lookback,
    X_scaler=X_scaler,
    y_scaler=y_scaler,
    progress_bar=True
)
```

To produce a sensitivity analysis with `SALib's analyze` function, all you need is your problem dictionary (`morris_problem`), the original Morris samples (`morris_sample`), and the predictions we just produced with those samples (`morris_preds`). There's an optional confidence interval level argument (`conf_level`), but the default of 0.95 is good. It uses resamples to compute this confidence level, which is 1,000 by default. This setting can also be changed with an optional `num_resamples` argument:

```
morris_sensitivities = ma.analyze(
    morris_problem, morris_sample, morris_preds,\
    print_to_console=False
)
```

Analyzing the elementary effects

`analyze` will return a dictionary with the Morris sensitivity indices, including the mean (μ) and standard deviation (σ) elementary effect, as well as the absolute value of the mean (μ^*). It's easier to appreciate these values in a tabular format so that we can place them into a DataFrame and sort and color-code them according to μ^*, which can be interpreted as the overall importance of the factor. σ, on the other hand, is how much the factor interacts with other ones:

```
morris_df = pd.DataFrame(
    {
        'features':morris_sensitivities['names'],
        'μ':morris_sensitivities['mu'],
        'μ*':morris_sensitivities['mu_star'],
        'σ':morris_sensitivities['sigma']
    }
)
morris_df.sort_values('μ*', ascending=False).style\
    .background_gradient(cmap='plasma', subset=['μ*'])
```

The preceding code outputs the DataFrame depicted in *Figure 9.13*. You can tell that is_holiday is one of the most important factors, at least during the bounds specified in the problem definition (morris_problem). Another thing to note is that weather does have an absolute mean elementary effect but inconclusive interaction effects. Groups are challenging to assess, especially when they are sparse binary factors:

features	μ	μ*	σ
hr	-560.18	1316.23	1393.25
dow	100.72	350.63	460.59
temp	263.15	311.29	344.00
weather	nan	154.45	nan
is_holiday	-85.24	151.30	299.68
cloud_coverage	-14.05	97.22	203.60
snow_1h	-29.57	49.24	156.02
rain_1h	0.66	45.81	134.17

Figure 9.13: The elementary effects decomposition of the factors

The DataFrame in the preceding figure is not the best way to visualize the elementary effects. When there are not too many factors, it's easier to plot them. SALib comes with two plotting methods. The horizontal bar plot (horizontal_bar_plot) and covariance plot (covariance_plot) can be placed side by side. The covariance plot is excellent, but it doesn't annotate the areas it delineates. We will learn about these next. So, solely for instructional purposes, we will use text to place the annotations:

```
fig, (ax0, ax1) = plt.subplots(1,2, figsize=(12,8))
mp.horizontal_bar_plot(ax0, morris_sensitivities, {})
mp.covariance_plot(ax1, morris_sensitivities, {})
ax1.text(
    ax1.get_xlim()[1] * 0.45, ax1.get_ylim()[1] * 0.75,\
    'Non-linear and/or-monotonic', color='gray',\
    horizontalalignment='center'
)
ax1.text(ax1.get_xlim()[1] * 0.75, ax1.get_ylim()[1] * 0.5,\
    'Almost Monotonic', color='gray', horizontalalignment='center')
ax1.text(ax1.get_xlim()[1] * 0.83, ax1.get_ylim()[1] * 0.2,\
    'Monotonic', color='gray', horizontalalignment='center')
ax1.text(ax1.get_xlim()[1] * 0.9, ax1.get_ylim()[1] * 0.025,
    'Linear', color='gray', horizontalalignment='center')
```

The preceding code produces the plots shown in *Figure 9.14*. The bar plot on the left ranks the factors by μ^*, while the lines sticking out of each bar signify their corresponding confidence bands. The covariance plot to the right is a scatter plot with μ^* on the *x*-axis and σ on the *y*-axis. Therefore, the farther right the point is, the more important it is, while the further up it is in the plot, the more it interacts with other factors and becomes increasingly less monotonic. Naturally, this means that factors that don't interact much and are mostly monotonic ones comply with linear regression assumptions, such as linearity and multicollinearity. However, the spectrum between linear and non-linear or non-monotonic is determined diagonally by the ratio of σ and μ^*:
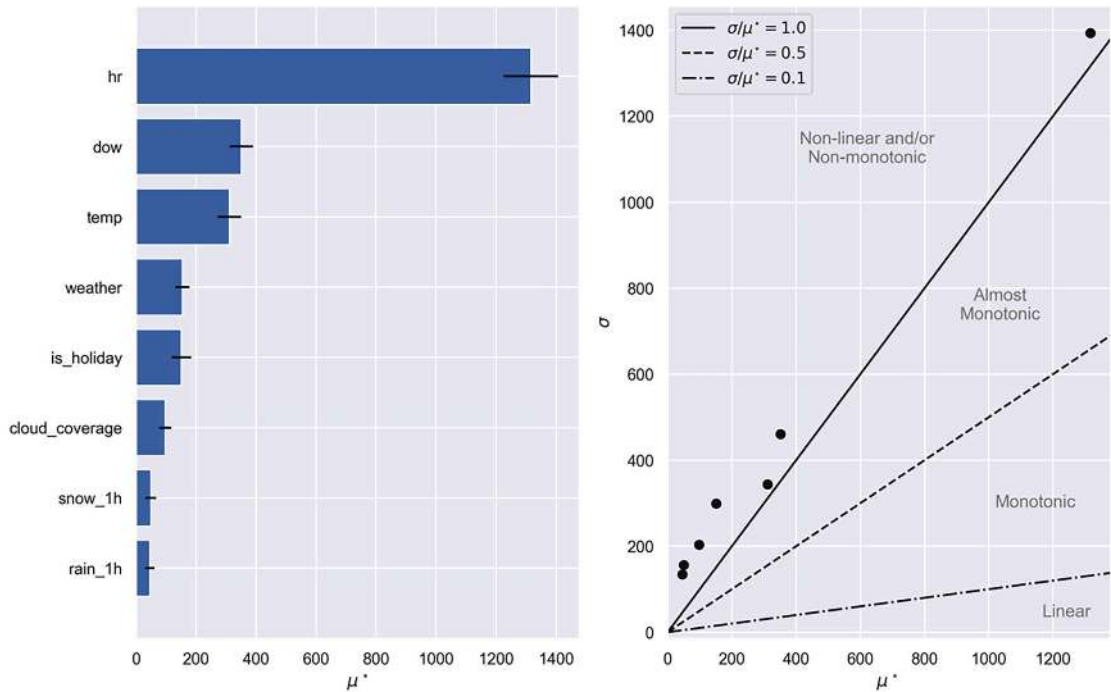
Figure 9.14: A bar and covariance plot depicting the elementary effects

You can tell by the preceding covariance plot that all the factors are non-linear or non-monotonic. `hr` is by far the most important, with the following two (`dow` and `temp`) clustered relatively nearby, followed by `weather` and `is_holiday`. The `weather` group is not on the plot because interactivity was inconclusive, yet `cloud_coverage`, `rain_1h`, and `snow_1h` are considerably more interactive than important on their own.

Elementary effects help us understand how to classify our factors in accordance with their effects on model outcomes. However, it's not a robust method to properly quantify their effects or those derived from factor interactions. For that, we would have to turn to a variance-based global method that uses a probabilistic framework to decompose the output's variance and trace it back to the inputs. Those methods include **Fourier Amplitude Sensitivity Test** (**FAST**) and **Sobol**. We will study the latter approach next.

Quantifying uncertainty and cost sensitivity with factor fixing

With the Morris indices, it became evident that all the factors are non-linear or non-monotonic. There's a high degree of interactivity between them – as expected! It should be no surprise that climate factors (`temp`, `rain_1h`, `snow_1h`, and `cloud_coverage`) are likely multicollinear with `hr`. There are also patterns to be found between `hr`, `is_holiday`, and `dow` and the target. Many of these factors most definitely don't have a monotonic relationship with the target. We know this already. For instance, traffic doesn't consistently increase as hours increase throughout the day. That's not the case for days of the week either!

However, we didn't know to what degree `is_holiday` and `temp` impacted the model, particularly during the crew's working hours, which was an important insight. That being said, factor prioritization with Morris indices is usually to be taken as a starting point or "first setting" because once you ascertain that there are interaction effects, it's best if you disentangle them. To this end, there's a "second setting" called **factor fixing**. We can quantify the variance and, by doing so, the uncertainty brought on by all the factors.

Only **variance-based methods** can quantify these effects in a statistically rigorous fashion. **Sobol sensitivity analysis** is one of these methods, which means that it decomposes the model's output variance into percentages and attributes it to the model's inputs and interactions. Like Morris, it has a sampling step, as well as a sensitivity index estimation step.

Unlike Morris, the sampling doesn't follow a series of levels but the input data's distribution. It uses a **quasi-Monte Carlo method**, where it samples points in hyperspace that follow the inputs' probability distributions. **Monte Carlo** methods are a family of algorithms that perform random sampling, often for optimization or simulation. They seek shortcuts on problems that would be impossible to solve with brute force or entirely deterministic approaches. Monte Carlo methods are common in sensitivity analysis precisely for this reason. Quasi-Monte Carlo methods have the same goal. However, they converge faster because they use a deterministic low-discrepancy sequence instead of using a pseudorandom one. The Sobol method uses the **Sobol sequence**, devised by the same mathematician. We will use another sampling scheme derived from Sobol's, called Saltelli's.

Once the samples have been produced, Monte Carlo estimators compute the variance-based sensitivity indices. These indices are capable of quantifying non-linear non-additive effects and second-order indices, which relate to the interaction between two factors. Morris can reveal interactivity in your model, but not precisely how it is manifested. Sobol can tell you what factors are interacting and to what degree.

Generating and predicting on Saltelli samples

To begin a Sobol sensitivity analysis with `SALib`, we must first define a problem. We'll do the same as we did with Morris. This time, we will reduce the factors because we realized that the `weather` grouping led to inconclusive results. We should include the least sparse of all the weather factors; that is, `weather_Clear`. And since Sobol uses a probabilistic framework, there's no harm in expanding the bounds to their minimum and maximum values for `temp`, `rain_1h`, and `cloud_coverage`, as seen in *Figure 9.12*:

```
sobol_problem = {
    'num_vars': 8,
    'names': ['dow', 'hr', 'temp', 'rain_1h', 'snow_1h',
              'cloud_coverage', 'is_holiday', 'weather_Clear'],
    'bounds': [
        [0, 4], # dow Monday through Friday
        [-1, 4], # hr
        [-3., 31.], # temp (C)
        [0., 21.], # rain_1h
        [0., 1.6], # snow_1h
        [0., 100.], # cloud_coverage
        [0, 1], # is_holiday
        [0, 1] # weather_Clear
      ],
    'groups': None
}
```

Generating the samples should look familiar too. The Saltelli `sample` function requires the following:

- A problem statement (`sobol_problem`)
- A number of samples to produce per factor (`300`)
- Second-order indices to compute (`calc_second_order=True`)

Given that we want the interactions, the output of `sample` is an array that has one column for each factor and $N \times (2F + 2)$ rows (where N is the number of samples and F is the number of factors). We have eight factors and 256 samples per factor, so `print` should output a shape of 4,608 rows and 8 columns. First, we will modify it, as we did previously, with `hstack` to add the 7 empty factors needed to make the predictions, resulting in 15 columns instead:

```
saltelli_sample = ss.sample(
    sobol_problem, 256, calc_second_order=True, seed=rand
)
saltelli_sample_mod = np.hstack(
    (saltelli_sample, np.zeros((saltelli_sample.shape[0],7)))
)
print(saltelli_sample_mod.shape)
```

Now, let's predict on these samples. This should take a while, so it's coffee time once more:

```
saltelli_preds = mldatasets.pprox._predict_ts(
    saltelli_sample_mod,
    X_df,
    gen_all,
    lstm_traffic_mdl,
    filt_fn=filt_fn,
    dist_metric=heom_dist.heom,
    lookback=lookback,
    X_scaler=X_scaler,
    y_scaler=y_scaler,
    progress_bar=True
)
```

Performing Sobol sensitivity analysis

For Sobol sensitivity analysis (`analyze`), all you need is a problem statement (`sobol_problem`) and the model outputs (`saltelli_preds`). But the predictions don't tell the story of uncertainty. Sure, there's variance in the predicted traffic, but that traffic is only a problem once it exceeds 1,500. Uncertainty is something you want to relate to risk or reward, costs or revenue, loss or profit – something tangible you can connect to your problem.

First, we must assess if there's any risk at all. To get an idea of whether the predicted traffic in the samples exceeded the no-construction threshold during working hours, we can use `print(max(saltelli_preds[:,0]))`. The maximum traffic level should be somewhere in the neighborhood of 1,800-1,900, which means that there's at least some risk that the construction company will pay a fine. Instead of using the predictions (`saltelli_preds`) as the model's output, we can create a simple binary array with ones when it exceeded 1,500 and zero otherwise. We will call this `costs`, and then run the `analyze` function with it. Note that `calc_second_order=True` is also set here. It will throw an error if `sample` and `analyze` don't have a consistent setting. Like with Morris, there's an optional confidence interval level argument (`conf_level`), but the default of 0.95 is good:

```
costs = np.where(saltelli_preds > 1500, 1,0)[:,0]
factor_fixing_sa = sa.analyze(
    sobol_problem,
    costs,
    calc_second_order=True,
    print_to_console=False
)
```

`analyze` will return a dictionary with the Sobol sensitivity indices, including the first-order (`S1`), second-order (`S2`), and total-order (`ST`) indices, as well as the total confidence bounds (`ST_conf`). The indices correspond to percentages, but the totals won't necessarily add up unless the model is additive. It's easier to appreciate these values in a tabular format so that we can place them into a DataFrame and sort and color-code them according to the total, which can be interpreted as the overall importance of the factor. However, we will leave the second-order indices out because they are two-dimensional and akin to a correlation plot:

```python
sobol_df = pd.DataFrame(
    {
        'features':sobol_problem['names'],
        '1st':factor_fixing_sa['S1'],
        'Total':factor_fixing_sa['ST'],
        'Total Conf':factor_fixing_sa['ST_conf'],
        'Mean of Input':saltelli_sample.mean(axis=0)[:8]
    }
)
sobol_df.sort_values('Total', ascending=False).style
    .background_gradient(cmap='plasma', subset=['Total'])
```

The preceding code outputs the DataFrame depicted in *Figure 9.15*. You can tell that `temp` and `is_holiday` are in the top four, at least during the bounds specified in the problem definition (`sobol_problem`). Another thing to note is that `weather_Clear` does have more of an effect on its own, but `rain_1h` and `cloud_coverage` seem to have no effect on the potential cost because they have zero total first-order indices:

features	1st	Total	Total Conf	Mean of Input
hr	0.27	0.69	0.54	1.50
dow	-0.14	0.62	0.60	1.99
temp	-0.00	0.21	0.31	14.08
snow_1h	-0.01	0.14	0.26	0.80
is_holiday	0.13	0.14	0.24	0.50
weather_Clear	-0.00	0.07	0.17	0.50
rain_1h	0.00	0.00	0.00	10.51
cloud_coverage	0.00	0.00	0.00	49.69

Figure 9.15: Sobol global sensitivity indices for the eight factors

Something interesting about the first-order values is how low they are, suggesting that interactions account for most of the model output variance. We can easily produce a heatmap with second-order indices to corroborate this. It's the combination of these indices and the first-order ones that add up to the totals:

```
S2 = factor_fixing_sa['S2']
divnorm = TwoSlopeNorm(vmin=S2.min(), vcenter=0, vmax=S2.max())
sns.heatmap(S2, center=0.00, norm=divnorm, cmap='coolwarm_r',\
            annot=True, fmt ='.2f',\
            xticklabels=sobol_problem['names'],\
            yticklabels=sobol_problem['names'])
```

The preceding code outputs the heatmap in *Figure 9.16*:
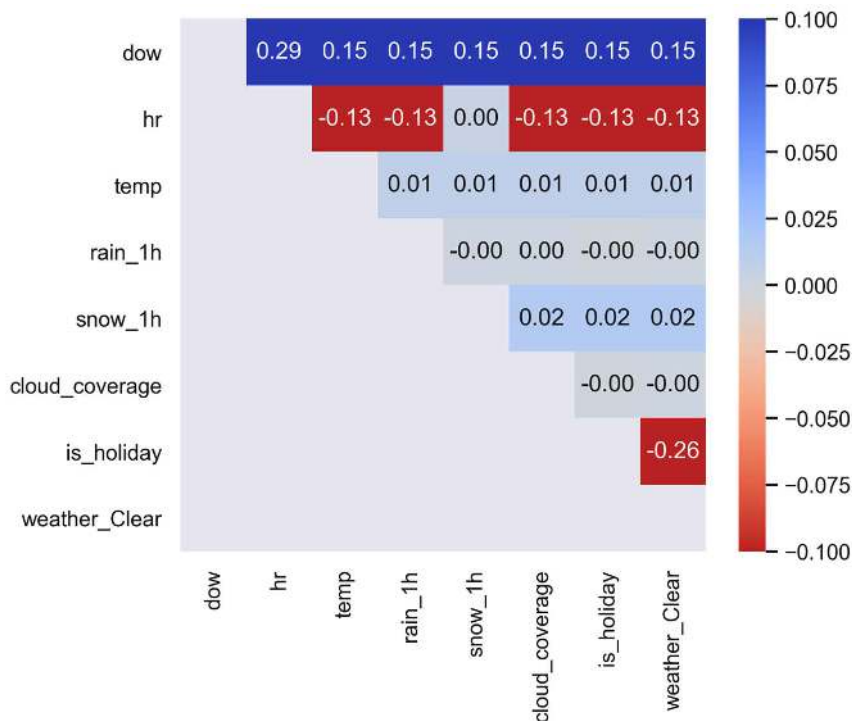
Figure 9.16: Sobol second-order indices for the eight factors

Here, you can tell that `is_holiday` and `weather_Clear` are the two factors that contribute the most to the output variance with the highest absolute value of 0.26. `dow` and `hr` have sizable interactions with all the factors.

Incorporating a realistic cost function

Now, we can create a cost function that takes our inputs (`saltelli_sample`) and outputs (`saltelli_preds`) and computes how much the twin cities would fine the construction company, plus any additional costs the additional traffic could produce.

It is better to do this if both the input and outputs are in the same array because we will need details from both to calculate the costs. We can use `hstack` to join the samples and their corresponding predictions, producing an array with eight columns (`saltelli_sample_preds`). We can then define a cost function that can compute the costs (`cost_fn`), given an array with these nine columns:

```
#Join input and outputs into a sample+prediction array
saltelli_sample_preds = np.hstack((saltelli_sample, saltelli_preds))
```

We know that the half-capacity threshold wasn't exceeded for any sample predictions, so we won't even bother to include the daily penalty in the function. Besides that, the fines are $15 per vehicle that exceeds the hourly no-construction threshold. In addition to these fines, to be able to leave on time, the construction company estimates additional costs: $1,500 in extra wages if the threshold is exceeded at 4 a.m. and $4,500 more on Fridays to speed up the moving of their equipment because it can't stay on the highway shoulder during weekends. Once we have the cost function, we can iterate through the combined array (`saltelli_sample_preds`), calculating costs for each sample. List comprehension can do this efficiently:

```python
#Define cost function
def cost_fn(x):
    cost = 0
    if x[8] > 1500:
        cost = (x[8] - 1500) * 15
    if round(x[1]) == 4:
        cost = cost + 1500
        if round(x[0]) == 4:
            cost = cost + 4500
    return cost
#Use list comprehension to compute costs for sample+prediction array
costs2 = np.array([cost_fn(xi) for xi in saltelli_sample_preds])
#Print total fines for entire sample predictions
print('Total Fines: $%s' % '{:,.2f}'.format(sum(costs2)))
```

The `print` statement should output a cost somewhere between $170,000 and $200,000. But not to worry! The construction crew only plans to work about 195 days on-site per year and 5 hours each day, for a total of 975 hours. However, there are 4,608 samples, which means that there are almost 5 years' worth of predicted costs due to excess traffic. In any case, the point of calculating these costs is to figure out how they relate to the model's inputs. More years' worth of samples means tighter confidence intervals:

```python
factor_fixing2_sa = sa.analyze(
    sobol_problem, costs2, calc_second_order=True,
    print_to_console=False
)
```

We can now perform the analysis again but with `costs2`, and we can save the analysis into a `factor_fixing2_sa` dictionary. Lastly, we can produce a new sorted and color-coded DataFrame with this dictionary's values, as we did previously for *Figure 9.15*, which generates the output shown in *Figure 9.17*.

As you can tell by *Figure 9.17* once the actual costs have been factored in, `dow`, `hr`, and `is_holiday` become riskier factors, while `snow_1h` and `temp` become less relevant when compared to *Figure 9.15*:

features	1st	Total	Total Conf	Mean of Input
dow	-0.03	1.31	2.47	1.99
hr	0.39	0.79	0.65	1.50
is_holiday	0.31	0.16	0.28	0.50
temp	-0.00	0.12	0.13	14.08
snow_1h	-0.01	0.11	0.20	0.80
weather_Clear	-0.01	0.09	0.14	0.50
cloud_coverage	0.00	0.00	0.00	49.69
rain_1h	0.00	0.00	0.00	10.51

Figure 9.17: Sobol global sensitivity indices for the eight factors using the realistic cost function

One thing that is hard to appreciate with a table is the confidence intervals of the sensitivity indices. For that, we can use a bar plot, but first, we must convert the entire dictionary into a DataFrame so that `SALib`'s plotting function can plot it:

```
factor_fixing2_df = factor_fixing2_sa.to_df()
fig, (ax) = plt.subplots(1,1, figsize=(15, 7))
sp.plot(factor_fixing2_df[0], ax=ax)
```

The preceding code generates the bar plot in *Figure 9.18*. The 95% confidence interval for `dow` is much larger than for other important factors, which shouldn't be surprising considering how much variance there is between days of the week. Another interesting insight is how `weather_Clear` has negative first-order effects, so the positive total-order indices are entirely attributed to second-order ones, which expand the confidence interval:
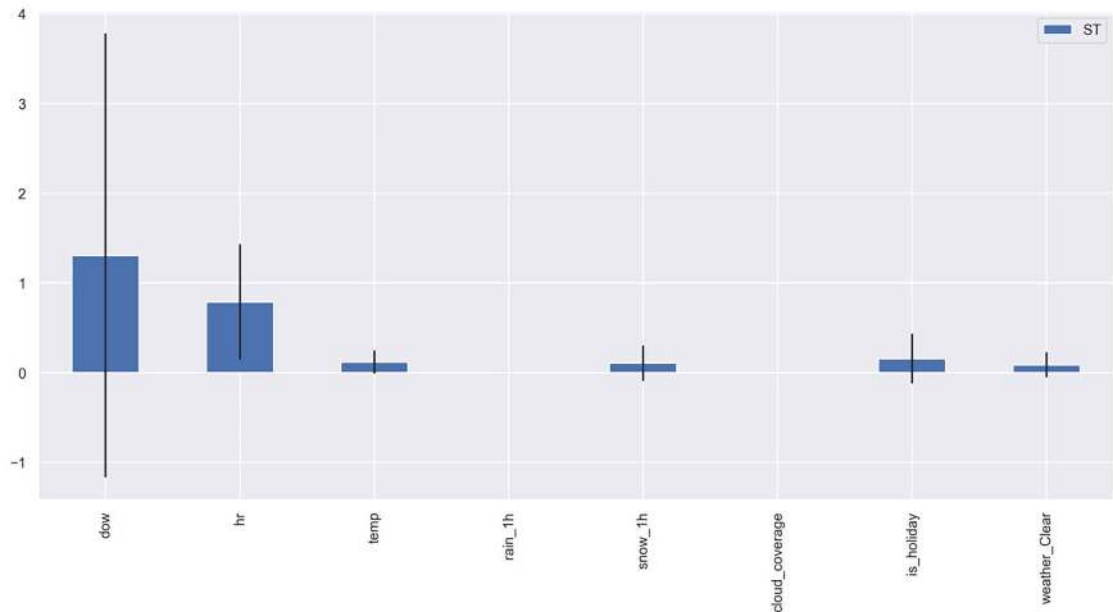
Figure 9.18: Bar plot with the Sobol sensitivity total-order indices and their confidence intervals using a realistic cost function

To understand how, let's plot the heatmap shown in *Figure 9.16* again but this time using `factor_fixing2_sa` instead of `factor_fixing_sa`. The heatmap in *Figure 9.19* should depict how the realistic costs reflect the interactions in the model:
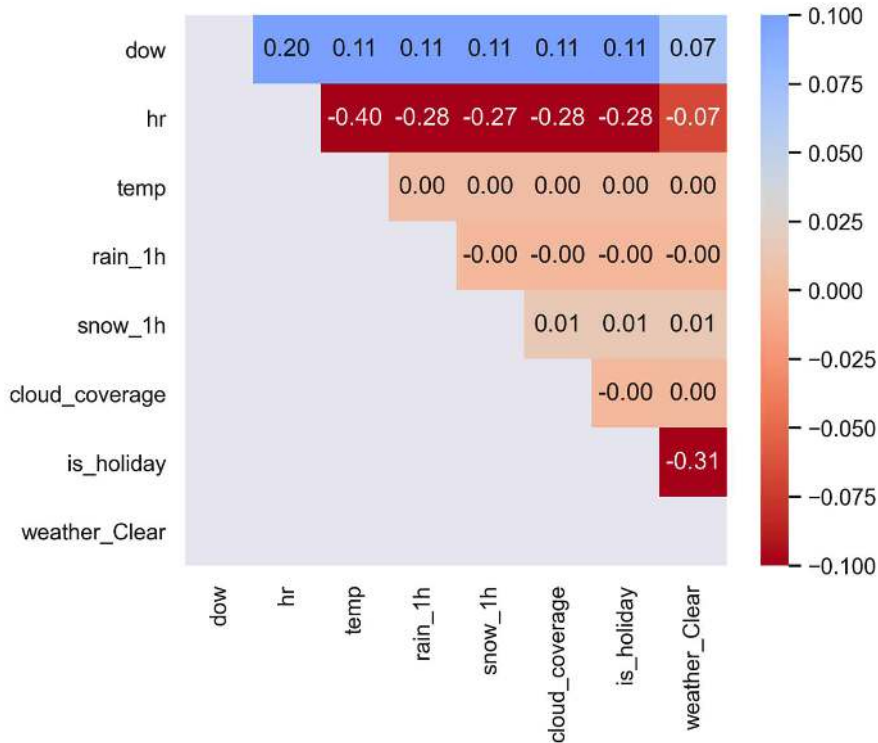
Figure 9.19: Sobol second-order indices for seven factors while factoring a more realistic cost function

The preceding heatmap shows similar salient interactions to those in *Figure 9.16* but they're much more nuanced since there are more shades. It becomes evident that `weather_Clear` has a magnifying effect when combined with `is_holiday`, and a tempering effect for `dow` and `hr`.

Mission accomplished

The mission was to train a traffic prediction model and understand what factors create uncertainty and possibly increase costs for the construction company. We can conclude a significant portion of the potential $35,000/year in fines can be attributed to the `is_holiday` factor. Therefore, the construction company should rethink working holidays. There are only seven or eight holidays between March and November, and they could cost more because of the fines than working on a few Sundays instead. With this caveat, the mission was successful, but there's still a lot of room for improvement.

Of course, these conclusions are for the `LSTM_traffic_168_compact1` model – which we can compare with other models. Try replacing the `model_name` at the beginning of the notebook with `LSTM_traffic_168_compact2`, an equally small but significantly more robust model, or `LSTM_traffic_168_optimal`, a larger slightly better-performing model, and re-running the notebook. Or glance at the notebooks named `Traffic_compact2` and `Traffic_optimal`, which already have been re-run with these corresponding models. You will find that it is possible to train and select models that manage uncertain inputs much better. That being said, improvement doesn't always come by simply selecting a better model.

For instance, one thing that could be covered in further depth is the true impact of `temp`, `rain_1h`, and `snow_1h`. Our prediction approximation method precluded Sobol from testing the effect of extreme weather events. If we modified the model to train on aggregated weather features at single timesteps and built in some guardrails, we could simulate weather extremes with Sobol. And the "third setting" of sensitivity analysis, known as factor mapping, could help pinpoint how exactly some factor values affect the predicted outcome, leading to a sturdier cost-benefit analysis, but we won't cover that in this chapter.

Throughout *Part Two* of this book, we explored an ecosystem of interpretation methods: global and local; model-specific and model-agnostic; permutation-based and sensitivity-based. There's no shortage of interpretation methods to choose from for any machine learning use case. However, it cannot be stressed enough that *NO method is perfect*. Still, they can complement each other to approximate a better understanding of your machine learning solution and the problem it aims to solve.

This chapter's focus on certainty in forecasting was designed to shed light on a particular problem in the machine learning community: overconfidence. *Chapter 1*, *Interpretation, Interpretability, Explainability; and Why Does It All Matter?*, in the *A business case of interpretability* section, described the many biases that infest human decision-making. These biases are often fueled by overconfidence in domain knowledge or our models' impressive results. And these impressive results cloud us from grasping the limitations of our models as the public distrust of AI increases.

As we discussed in *Chapter 1*, *Interpretation, Interpretability, Explainability; and Why Does It All Matter?*, machine learning is only meant to tackle *incomplete problems*. Otherwise, we might as well use deterministic and procedural programming like those found in closed-loop systems. The best we can do to solve an incomplete problem is an incomplete solution, which should be optimized to solve as much of it as possible. Whether through gradient descent, least-squares estimation, or splitting and pruning a decision tree, machine learning doesn't produce a model that generalizes perfectly. That lack of completeness in machine learning is precisely why we need interpretation methods. In a nutshell: models learn from our data, and we can learn a lot from our models, but only if we interpret them!

Interpretability doesn't stop there, though. Model interpretations can drive decisions and help us understand model strengths and weaknesses. However, often, there are problems in the data or models themselves that can make them less interpretable. In *Part Three* of this book, we'll learn how to tune models and the training data for interpretability by reducing complexity, mitigating bias, placing guardrails, and enhancing reliability.

Statistician George E.P. Box famously quipped that "*all models are wrong, but some are useful.*" Perhaps they aren't always wrong, but humility is required from machine learning practitioners to accept that even high-performance models should be subject to scrutiny and our assumptions about them. Uncertainty with machine learning models is expected and shouldn't be a source of shame or embarrassment. This leads us to another takeaway from this chapter: that uncertainty comes with ramifications, be it costs or profit lift, and that we can gauge these with sensitivity analysis.

Summary

After reading this chapter, you should understand how to assess a time series model's predictive performance, know how to perform local interpretations for them with integrated gradients, and know how to produce both local and global attributions with SHAP. You should also know how to leverage sensitivity analysis factor prioritization and factor fixing for any model.

In the next chapter, we will learn how to reduce the complexity of a model and make it more interpretable with feature selection and engineering.

Dataset and image sources

- TomTom, 2019, Traffic Index: `https://nonews.co/wp-content/uploads/2020/02/TomTom2019.pdf`
- UCI Machine Learning Repository, 2019, Metro Interstate Traffic Volume Data Set: `https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume`

Further reading

- Wilson, D.R., and Martinez, T., 1997, *Improved Heterogeneous Distance Functions*. J. Artif. Int. Res. 6-1. pp.1-34: `https://arxiv.org/abs/cs/9701101`
- Morris, M., 1991, *Factorial sampling plans for preliminary computational experiments*. Quality Engineering, 37, 307-310: `https://doi.org/10.2307%2F1269043`
- Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M., 2007, *Sensitivity analysis in practice: A guide to assessing scientific models*. Chichester: John Wiley & Sons.
- Sobol, I.M., 2001, *Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates*. MATH COMPUT SIMULAT,55(1–3),271-280: `https://doi.org/10.1016/S0378-4754(00)00270-6`
- Saltelli, A., P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola, 2010, *Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index*. Computer Physics Communications, 181(2):259-270: `https://doi.org/10.1016/j.cpc.2009.09.018`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

10

Feature Selection and Engineering for Interpretability

In the first three chapters, we discussed how complexity hinders **Machine Learning** (**ML**) interpretability. There's a trade-off because you may need some complexity to maximize predictive performance, yet not to the extent that you cannot rely on the model to satisfy the tenets of interpretability: fairness, accountability, and transparency. This chapter is the first of four focused on how to tune for interpretability. One of the easiest ways to improve interpretability is through feature selection. It has many benefits, such as faster training and making the model easier to interpret. But if these two reasons don't convince you, perhaps another one will.

A common misunderstanding is that complex models can self-select features and perform well nonetheless, so why even bother to select features? Yes, many model classes have mechanisms that can take care of useless features, but they aren't perfect. And the potential for overfitting increases with each one that remains. Overfitted models aren't reliable, even if they are more accurate. So, while employing model mechanisms such as regularization is still highly recommended to avoid overfitting, feature selection is still useful.

In this chapter, we will comprehend how irrelevant features adversely weigh on the outcome of a model and thus, the importance of feature selection for model interpretability. Then, we will review filter-based feature selection methods such as **Spearman's correlation** and learn about embedded methods such as **LASSO and ridge regression**. Then, we will discover wrapper methods such as **sequential feature selection**, and hybrid ones such as **Recursive Feature Elimination** (**RFE**). Lastly, even though feature engineering is typically conducted before selection, there's value in exploring feature engineering for many reasons after the dust has settled and features have been selected.

These are the main topics we are going to cover in this chapter:

- Understanding the effect of irrelevant features
- Reviewing filter-based feature selection methods
- Exploring embedded feature selection methods
- Discovering wrapper, hybrid, and advanced feature selection methods

•   Considering feature engineering

Let's begin!

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `scipy`, `mlxtend`, `sklearn-genetic-opt`, `xgboost`, `sklearn`, `matplotlib`, and `seaborn` libraries. Instructions on how to install all these libraries are in the *Preface*.

> The GitHub code for this chapter is located here: `https://packt.link/1qP4P`.

The mission

It has been estimated that there are over 10 million non-profits worldwide, and while a large portion of them have public funding, most of them depend mostly on private donors, both corporate and individual, to continue operations. As such, fundraising is mission-critical and carried out throughout the year.

Year over year, donation revenue has grown, but there are several problems non-profits face: donor interests evolve, so a charity popular one year might be forgotten the next; competition is fierce between non-profits, and demographics are shifting. In the United States, the average donor only gives two charitable gifts per year and is over 64 years old. Identifying potential donors is challenging, and campaigns to reach them can be expensive.

A National Veterans Organization non-profit arm has a large mailing list of about 190,000 past donors and would like to send a special mailer to ask for donations. However, even with a special bulk discount rate, it costs them $0.68 per address. This adds up to over $130,000. They only have a marketing budget of $35,000. Given that they have made this a high priority, they are willing to extend the budget but only if the **Return On Investment** (**ROI**) is high enough to justify the additional cost.

To minimize the use of their limited budget, instead of mass mailing, they'd like to try direct mailing, which aims to identify potential donors using what is already known, such as past donations, geographic location, and demographic data. They will reach other donors via email instead, which is much cheaper, costing no more than $1,000 per month for their entire list. They hope this hybrid marketing plan will yield better results. They also recognize that high-value donors respond better to personalized paper mailers, while smaller donors respond better to email anyway.

No more than six percent of the mailing list donates to any given campaign. Using ML to predict human behavior is by no means an easy task, especially when the data categories are imbalanced. Nevertheless, success is not measured by the highest predictive accuracy but by profit lift. In other words, the direct mailing model evaluated on the test dataset should produce more profit than if they mass-mailed the entire dataset.

They have sought your assistance to use ML to produce a model that identifies the most probable donors, but also in a way that *guarantees* an ROI.

You received the dataset from the non-profit, which is approximately evenly split into training and test data. If you send the mailer to absolutely everybody in the test dataset, you make a profit of $11,173, but if you manage to identify only those who will donate, the maximum yield of $73,136 will be attained. Your goal is to achieve a high-profit lift and reasonable ROI. When the campaign runs, it will identify the most probably donors for the entire mailing list, and the non-profit hopes to spend not much more than $35,000 in total. However, the dataset has 435 columns, and some simple statistical tests and modeling exercises show that the data is too noisy to identify the potential donors' reliability because of overfitting.

The approach

You've decided to first fit a base model with all the features and assess it at different levels of complexity to understand the relationship between the increased number of features and the propensity for the predictive model to overfit to the training data. Then, you will employ a series of feature selection methods ranging from simple filter-based methods to the most advanced ones to determine which one achieves the profitability and reliability goals sought by the client. Lastly, once a list of final features has been selected, you can try feature engineering.

Given the cost-sensitive nature of the problem, thresholds are important to optimize the profit lift. We will get into the role of thresholds later on, but one significant effect is that even though this is a classification problem, it is best to use regression models, and then use predictions to classify so that there's only one threshold to tune. That is, for classification models, you would need a threshold for the label, say those that donated over $1, and then another one for probabilities predicted. On the other hand, regression predicts the donation, and the threshold can be optimized based on that.

The preparations

The code for this example can be found at `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/10/Mailer.ipynb`.

Loading the libraries

To run this example, we need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas`, `numpy`, and `scipy` to manipulate it
- `mlxtend`, `sklearn-genetic-opt`, `xgboost`, and `sklearn` (scikit-learn) to fit the models
- `matplotlib` and `seaborn` to create and visualize the interpretations

To load the libraries, use the following code block:

```
import math
import os
import mldatasets
import pandas as pd
import numpy as np
```

```
import timeit
from tqdm.notebook import tqdm
from sklearn.feature_selection import VarianceThreshold,\
                                      mutual_info_classif, SelectKBest
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LogisticRegression,\
                                 LassoCV, LassoLarsCV, LassoLarsIC
from mlxtend.feature_selection import SequentialFeatureSelector
from sklearn.feature_selection import RFECV
from sklearn.decomposition import PCA import shap
from sklearn-genetic-opt import GAFeatureSelectionCV
from scipy.stats import rankdata
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns
```

Next, we will load and prepare the dataset.

Understanding and preparing the data

We load the data like this into two DataFrames (`X_train` and `X_test`) with the features and two `numpy` arrays with corresponding labels (`y_train` and `y_test`). Please note that these DataFrames have already been previously prepared for us to remove sparse or unnecessary features, treat missing values, and encode categorical features:

```
X_train, X_test, y_train, y_test = mldatasets.load(
    "nonprofit-mailer",
    prepare=True
)
y_train = y_train.squeeze()
y_test = y_test.squeeze()
```

All features are numeric with no missing values and categorical features have already been one-hot encoded for us. Between both train and test mailing lists, there should be over 191,500 records and 435 features. You can check this is the case like this:

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

The preceding code should output the following:

```
(95485, 435)
(95485,)
(96017, 435)
(96017,)
```

Next, we can verify that the test labels have the right number of donors (`test_donors`), donations (`test_donations`), and hypothetical profit ranges (`test_min_profit` and `test_max_profit`) using the variable cost of $0.68 (`var_cost`). We can print these, and then do the same for the training dataset:

```
var_cost = 0.68
y_test_donors = y_test[y_test > 0]
test_donors = len(y_test_donors)
test_donations = sum(y_test_donors)
test_min_profit = test_donations - (len(y_test)*var_cost)
test_max_profit = test_donations - (test_donors*var_cost)
print(
    '%s test donors totaling $%.0f (min profit: $%.0f,\
    max profit: $%.0f)'
    %(test_donors, test_donations, test_min_profit,\
        test_max_profit))

y_train_donors = y_train[y_train > 0]
train_donors = len(y_train_donors)
train_donations = sum(y_train_donors)
train_min_profit = train_donations – (len(y_train)*var_cost)
train_max_profit = train_donations – (train_donors*var_cost)
print(
    '%s train donors totaling $%.0f (min profit: $%.0f,\
    max profit: $%.0f)'
    %(train_donors, train_donations, train_min_profit,\
    train_max_profit))
```

The preceding code should output the following:

```
4894 test donors totaling $76464 (min profit: $11173, max profit: $73136)
4812 train donors totaling $75113 (min profit: $10183, max profit: $71841)
```

Indeed, if the non-profit mass-mailed to everyone on the test mailing list, they'd make about $11,000 profit but would have to go grossly over budget to achieve this. The non-profit recognizes that making the max profit by identifying and targeting only donors is nearly an impossible feat. Therefore, they would be content with producing a model that reliably can yield more than the min profit but with a smaller cost, preferably under budget.

Understanding the effect of irrelevant features

Feature selection is also known as **variable** or **attribute selection.** It is the method by which you can automatically or manually select a subset of specific features useful to the construction of ML models.

It's not necessarily true that more features lead to better models. Irrelevant features can impact the learning process, leading to overfitting. Therefore, we need some strategies to remove any features that might adversely affect learning. Some of the advantages of selecting a smaller subset of features include the following:

- *It's easier to understand simpler models*: For instance, feature importance for a model that uses 15 variables is much easier to grasp than one that uses 150 variables.
- *Shorter training time*: Reducing the number of variables decreases the cost of computing, speeds up model training, and perhaps most notably, simpler models have quicker inference times.
- *Improved generalization by reducing overfitting*: Sometimes, with little prediction value, many of the variables are just noise. The ML model, however, learns from this noise and triggers overfitting to the training data while minimizing generalization simultaneously. We may significantly enhance the generalization of ML models by removing these irrelevant or noisy features.
- *Variable redundancy*: It is common for datasets to have collinear features, which could mean some are redundant. In cases like these, as long as no significant information is lost, we can retain only one of the correlated features and delete the others.

Now, we will fit some models to demonstrate the effect of too many features.

Creating a base model

Let's create a base model for our mailing list dataset to see how this plays out. But first, let's set our random seed for reproducibility:

```
rand = 9
os.environ['PYTHONHASHSEED']=str(rand)
np.random.seed(rand)
```

We will use XGBoost's **Random Forest** (**RF**) regressor (`XGBRFRegressor`) throughout this chapter. It's just like scikit-learn's but faster because it uses second-order approximations of the objective function. It also has more options, such as setting the learning rate and monotonic constraints, examined in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*. We initialize `XGBRFRegressor` with a conservative initial `max_depth` value of 4 and always use 200 estimators for consistency. Then, we fit it with our training data. We will use `timeit` to measure how long it takes, which we save in a variable (`baseline_time`) for later reference:

```
stime = timeit.default_timer()
reg_mdl = xgb.XGBRFRegressor(max_depth=4, n_estimators=200, seed=rand)
fitted_mdl = reg_mdl.fit(X_train, y_train)
etime = timeit.default_timer()
baseline_time = etime-stime
```

Now that we have a base model, let's evaluate it.

Evaluating the model

Next, let's create a dictionary (`reg_mdls`) to house all the models we will fit in this chapter to test which feature subsets produce the best models. Here, we can evaluate the RF model with all the features and a `max_depth` value of 4 (`rf_4_all`) using `evaluate_reg_mdl`. It will make a summary and a scatter plot with a regression line:

```
reg_mdls = {}
reg_mdls['rf_4_all'] = mldatasets.evaluate_reg_mdl(
    fitted_mdl,
    X_train,
    X_test,
    y_train,
    y_test,
    plot_regplot=True,
    ret_eval_dict=True
)
```

The preceding code produces the metrics and plot shown in *Figure 10.1*:
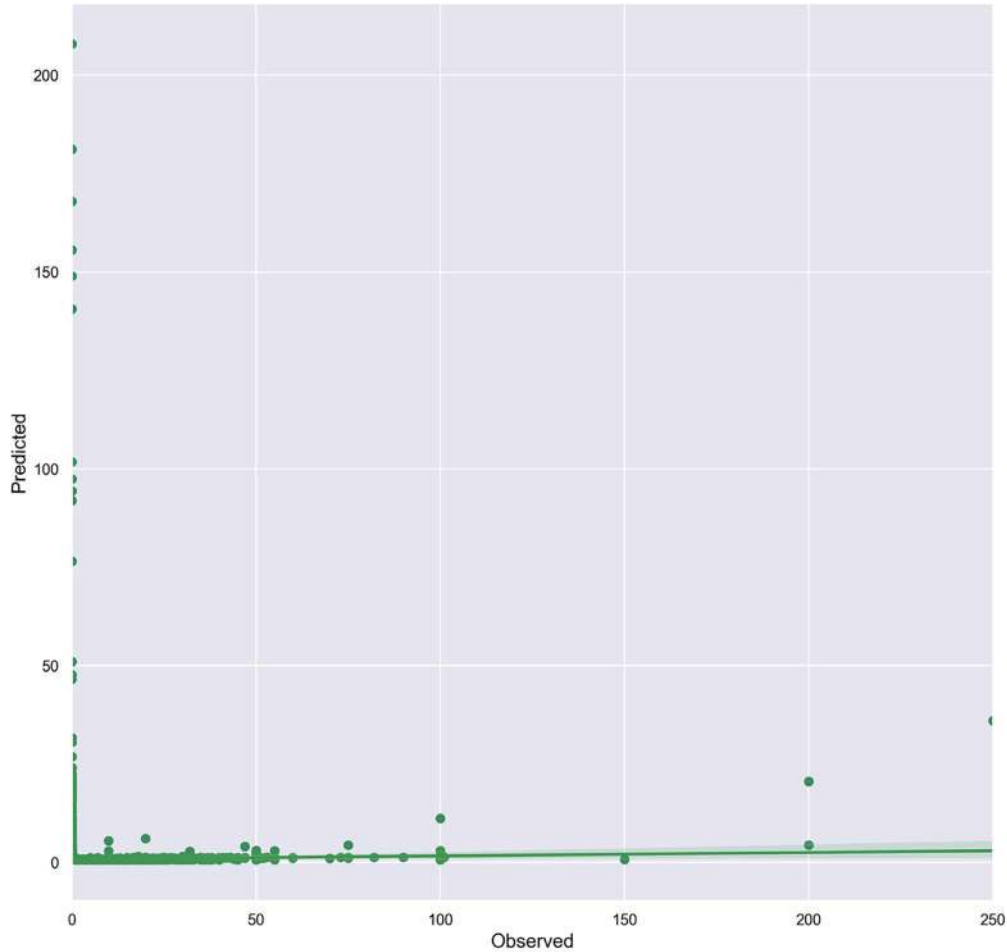
RMSE_train: 4.3210 RMSE_test: 4.6359   r2: -0.1084

Figure 10.1: Base model predictive performance

For a plot like the one in *Figure 10.1*, usually, a diagonal line is expected, so one glance at this plot would tell you that the model is not predictive. Also, the RMSEs may not seem bad but in the context of such a lopsided problem, they are dismal. Consider this: only 5% of the list makes a donation, and only 20% of those are over $20, so an average error of $4.3 – $4.6 is enormous.

So, is this model useless? The answer lies in what thresholds we use to classify with it. Let's start by defining an array of thresholds (`threshs`), ranging from $0.40 to $25. We start spacing these out by a cent until it reaches $1, then by 10 cents until it reaches $3, and after that, space by $1:

```
threshs = np.hstack(
    [
      np.linspace(0.40,1,61),
```

```
        np.linspace(1.1,3,20),
        np.linspace(4,25,22)
    ]
)
```

There's a function in `mldatasets` that can compute profit at every threshold (`profits_by_thresh`). All it needs is the actual (`y_test`) and predicted labels, followed by the thresholds (`threshs`), the variable cost (`var_costs`), and the `min_profit` required. It produces a `pandas` DataFrame with the revenue, costs, profit, and ROI for every threshold, as long as the profit is above `min_profit`. Remember, we had set this minimum at the beginning of the chapter as $11,173 because it makes no sense to target donors under this amount. After we generate these profit DataFrames for the test and train datasets, we can place the maximum and minimum amounts in the model's dictionary for later use. And then, we employ `compare_df_plots` to plot the costs, profits, and ROI ratio for testing and training for every threshold where it exceeded the profit minimum:

```
y_formatter = plt.FuncFormatter(
    lambda x, loc: "${:,}K".format(x/1000)
)
profits_test = mldatasets.profits_by_thresh(
    y_test,
    reg_mdls['rf_4_all']['preds_test'],
    threshs,
    var_costs=var_cost,
    min_profit=test_min_profit
)
profits_train = mldatasets.profits_by_thresh(
    y_train,
    reg_mdls['rf_4_all']['preds_train'],
    threshs,
    var_costs=var_cost,
    min_profit=train_min_profit
)
reg_mdls['rf_4_all']['max_profit_train'] =profits_train.profit.max()
reg_mdls['rf_4_all']['max_profit_test'] = profits_test.profit.max()
reg_mdls['rf_4_all']['max_roi'] = profits_test.roi.max()
reg_mdls['rf_4_all']['min_costs'] = profits_test.costs.min()
reg_mdls['rf_4_all']['profits_train'] = profits_train
reg_mdls['rf_4_all']['profits_test'] = profits_test
mldatasets.compare_df_plots(
    profits_test[['costs', 'profit', 'roi']],
    profits_train[['costs', 'profit', 'roi']],
    'Test',
    'Train',
```

```
        y_formatter=y_formatter,
        x_label='Threshold',\
        plot_args={'secondary_y':'roi'}
    )
```

The preceding snippet generates the plots in *Figure 10.2*. You can tell that **Test** and **Train** are almost identical. Costs decrease steadily at a high rate and profit at a lower rate, while ROI increases steadily. However, some differences exist, such as ROI, which becomes a bit higher eventually, and although viable thresholds start at the same point, **Train** does end at a different threshold. It turns out the model can turn a profit, so despite the appearance of the plot in *Figure 10.1*, the model is far from useless:
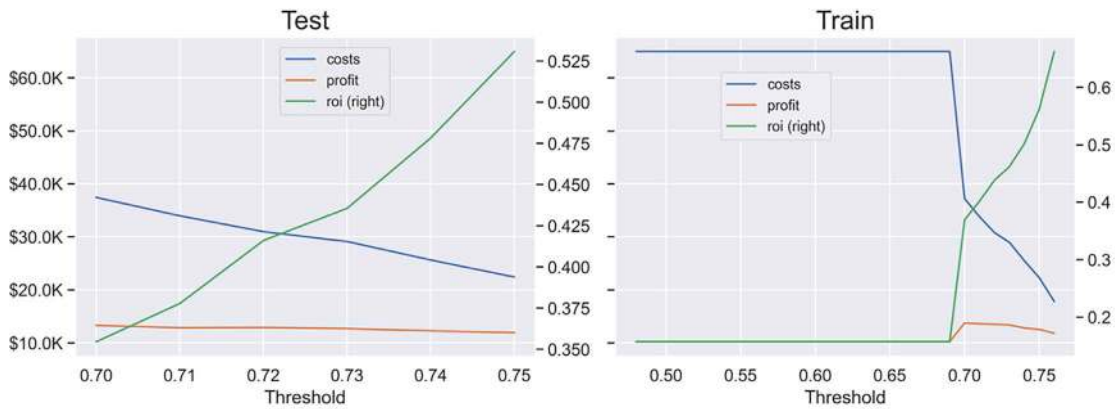
Figure 10.2: Comparison between profit, costs, and ROI for the test and train datasets for the base model across thresholds

The difference in RMSEs for the train and test sets didn't lie. The model did not overfit. The main reason for this is that we used relatively shallow trees by setting our `max_depth` value at 4. We can easily see this effect of using shallow trees by computing how many features had a `feature_importances_` value of over 0:

```
reg_mdls['rf_4_all']['total_feat'] =\
    reg_mdls['rf_4_all']['fitted'].feature_importances_.shape[0] reg_
mdls['rf_4_all']['num_feat'] = sum(
    reg_mdls['rf_4_all']['fitted'].feature_importances_ > 0
)
print(reg_mdls['rf_4_all']['num_feat'])
```

The preceding code outputs 160. In other words, only 160 were used out of 435—there are only so many features that can be accommodated in such a shallow tree! Naturally, this leads to lowering overfitting, but at the same time, the choice of features with measures of impurity over a random selection of features is not necessarily the most optimal.

Training the base model at different max depths

So, what happens if we make the trees deeper? Let's repeat all the steps we did for the shallow one but for max depths between 5 and 12:

```python
for depth in tqdm(range(5, 13)):
mdlname = 'rf_'+str(depth)+'_all'
stime = timeit.default_timer()
reg_mdl = xgb.XGBRFRegressor(
    max_depth=depth,
    n_estimators=200,
    seed=rand
)
fitted_mdl = reg_mdl.fit(X_train, y_train)
etime = timeit.default_timer()
reg_mdls[mdlname] = mldatasets.evaluate_reg_mdl(
    fitted_mdl,
    X_train,
    X_test,
    y_train,
    y_test,
    plot_regplot=False,
    show_summary=False,
    ret_eval_dict=True
)
reg_mdls[mdlname]['speed'] = (etime - stime)/baseline_time
reg_mdls[mdlname]['depth'] = depth
reg_mdls[mdlname]['fs'] = 'all'
profits_test = mldatasets.profits_by_thresh(
    y_test,
    reg_mdls[mdlname]['preds_test'],
    threshs,
    var_costs=var_cost,
    min_profit=test_min_profit
)
profits_train = mldatasets.profits_by_thresh(
    y_train,
    reg_mdls[mdlname]['preds_train'],
    threshs,
    var_costs=var_cost,
    min_profit=train_min_profit
)
```

```
reg_mdls[mdlname]['max_profit_train'] = profits_train.profit.max()
reg_mdls[mdlname]['max_profit_test'] = profits_test.profit.max()
reg_mdls[mdlname]['max_roi'] = profits_test.roi.max()
reg_mdls[mdlname]['min_costs'] = profits_test.costs.min()
reg_mdls[mdlname]['profits_train'] = profits_train
reg_mdls[mdlname]['profits_test'] = profits_test
reg_mdls[mdlname]['total_feat'] =\
reg_mdls[mdlname]['fitted'].feature_importances_.shape[0]
reg_mdls[mdlname]['num_feat'] = sum(
    reg_mdls[mdlname]['fitted'].feature_importances_ > 0)
```

Now, let's plot the details in the profits DataFrames for the "deepest" model (with a max depth of 12) as we did before with `compare_df_plots`, producing *Figure 10.3*:
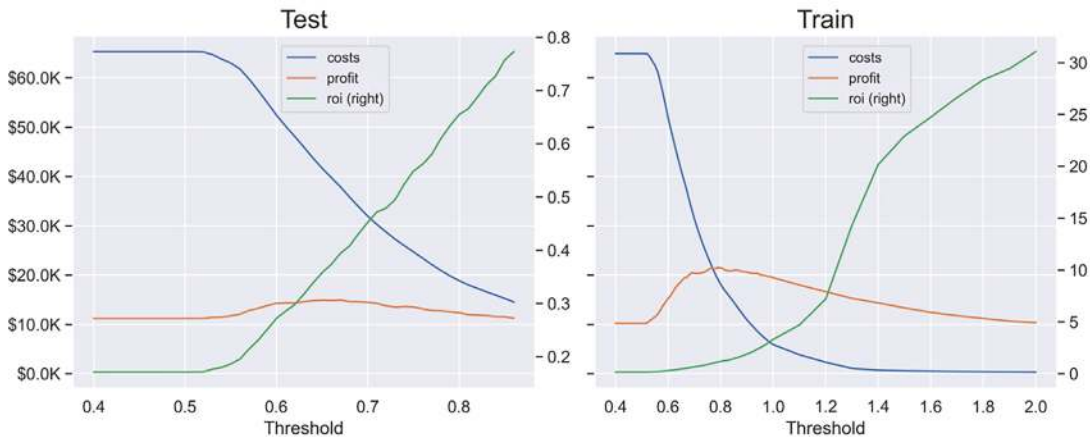
Figure 10.3: Comparison between profit, costs, and ROI for the test and train datasets for a "deep" base model across thresholds

See how different **Test** and **Train** are this time in *Figure 10.3*. **Test** reaches a max of about $15,000 and **Train** exceeds $20,000. **Train**'s costs dramatically fall, making the ROI orders of magnitude much higher than **Test**. Also, the ranges of thresholds are much different. Why is this a problem, you ask? If we had to guess what threshold to use to pick who to target in the next mailer, the optimal for **Train** is higher than for **Test**—meaning that by using an overfit model, we could miss the mark and under-perform on unseen data.

Next, let's convert our model dictionary (`reg_mdls`) into a DataFrame and extract some details from it. Then, we can sort it by depth, format it, color-code it, and output it:

```
def display_mdl_metrics(reg_mdls, sort_by='depth', max_depth=None):
    reg_metrics_df = pd.DataFrame.from_dict( reg_mdls, 'index')\
                        [['depth', 'fs', 'rmse_train', 'rmse_test',\
                          'max_profit_train',\
                          'max_profit_test', 'max_roi',\
                          'min_costs', 'speed', 'num_feat']]
```

```
    pd.set_option('precision', 2)
    html = reg_metrics_df.sort_values(
        by=sort_by, ascending=False).style.\
        format({'max_profit_train':'${0:,.0f}',\
        'max_profit_test':'${0:,.0f}', 'min_costs':'${0:,.0f}'}).\
        background_gradient(cmap='plasma', low=0.3, high=1,
                            subset=['rmse_train', 'rmse_test']).\
        background_gradient(cmap='viridis', low=1, high=0.3,
                            subset=[
                                'max_profit_train', 'max_profit_test'
                            ]
                            )
    return html

display_mdl_metrics(reg_mdls)
```

The preceding snippet leverages the `display_mdl_metrics` function to output the DataFrame shown in *Figure 10.4*. Something that should be immediately visible is how RMSE train and RMSE test are inverses. One decreases dramatically, and another increases slightly as the depth increases. The same can be said for profit. ROI tends to increase with depth and training speed and the number of features used as well:

	depth	fs	rmse_train	rmse_test	max_profit_train	max_profit_test	max_roi	min_costs	speed	num_feat
rf_12_all	12	all	3.94	4.69	$21,522	$14,933	0.77	$14,532	2.89	415
rf_11_all	11	all	3.99	4.69	$19,904	$15,142	0.76	$14,928	2.74	398
rf_10_all	10	all	4.05	4.68	$18,604	$14,987	0.78	$14,396	2.53	383
rf_9_all	9	all	4.10	4.68	$17,452	$14,778	0.80	$13,997	2.20	346
rf_8_all	8	all	4.14	4.67	$16,440	$14,563	0.73	$15,309	1.98	315
rf_7_all	7	all	4.18	4.66	$15,435	$14,186	0.66	$17,165	1.71	277
rf_6_all	6	all	4.23	4.65	$14,655	$13,851	0.59	$19,305	1.49	240
rf_5_all	5	all	4.27	4.64	$14,242	$13,752	0.59	$19,199	1.18	201
rf_4_all	4	all	4.32	4.64	$13,716	$13,262	0.53	$22,392	1.00	160

Figure 10.4: Comparing metrics for all base RF models with different depths

We could be tempted to use `rf_11_all` with the highest profitability, but it would be risky to use it! A common misunderstanding is that black-box models can effectively cut through any number of irrelevant features. While they are often be able to find something of value and make the most out of it, too many features could hinder their reliability by overfitting to noise in the training dataset. Fortunately, there is a sweet spot where you can reach high profitability with minimal overfitting, but to get there, we have to reduce the number of features first!

Reviewing filter-based feature selection methods

Filter-based methods independently select features from a dataset without employing any ML. These methods depend only on the variables' characteristics and are relatively effective, computationally inexpensive, and quick to perform. Therefore, being the low-hanging fruit of feature selection methods, they are usually the first step in any feature selection pipeline.

Filter-based methods can be categorized as:

- **Univariate**: Individually and independently of the feature space, they evaluate and rate a single feature at a time. One problem that can occur with univariate methods is that they may filter out too much since they don't take into consideration the relationship between features.
- **Multivariate**: These take into account the entire feature space and how features interact with each other.

Overall, for the removal of obsolete, redundant, constant, duplicated, and uncorrelated features, filter methods are very strong. However, by not accounting for complex, non-linear, non-monotonic correlations and interactions that only ML models can find, they aren't effective whenever these relationships are prominent in the data.

We will review three categories of filter-based methods:

- Basic
- Correlation
- Ranking

We will explain them further in their own sections.

Basic filter-based methods

We employ **basic filter methods** in the data preparation stage, specifically, the data cleaning stage, before any modeling. The reason for this is there's a low risk of making feature selection decisions that would adversely impact models. They involve common-sense operations such as removing features that carry no information or duplicate it.

Constant features with a variance threshold

Constant features don't change in the training dataset and, therefore, carry no information, and the model can't learn from them. We can use a univariate method called `VarianceThreshold`, which removes low-variance features. We will use a threshold of zero because we want to filter out only features with **zero variance**—in other words, constant features. It only works with numeric features, so we must first identify which features are numeric and which are categorical. Once we fit the method on the numeric columns, `get_support()` returns the list of features that aren't constant, and we can use set algebra to return only the constant features (`num_const_cols`):

```
num_cols_l = X_train.select_dtypes([np.number]).columns
cat_cols_l = X_train.select_dtypes([np.bool, np.object]).columns
```

```
num_const = VarianceThreshold(threshold=0)
num_const.fit(X_train[num_cols_l])
num_const_cols = list(
    set(X_train[num_cols_l].columns) -
    set(num_cols_l[num_const.get_support()])
)
```

The preceding snippet produced a list of constant numeric features, but how about categorical features? Categorical features would only have one category or unique value. We can easily check this by applying the nunique() function on categorical features. It will return a pandas series, and then a lambda function can filter out only those with one unique value. Then, .index.tolist() returns the name of the features as a list. Now, we just join both lists of constant features, and voilà! We have all constants (all_const_cols). We can print them; there should be three:

```
cat_const_cols = X_train[cat_cols_l].nunique()[lambda x:\
                                             x<2].index.tolist()
all_const_cols = num_const_cols + cat_const_cols
print(all_const_cols)
```

In most cases, removing constant features isn't good enough. A redundant feature might be almost constant or **quasi-constant**.

Quasi-constant features with value_counts

Quasi-constant features are almost entirely the same value. Unlike constant filtering, using a variance threshold won't work because high variance and quasi-constantness aren't mutually exclusive. Instead, we will iterate all features and get value_counts(), which returns the number of rows for each value. Then, divide these counts by the total number of rows to get a percentage and sort by the highest. If the top value is higher than the predetermined threshold (thresh), we append it to a list of quasi-constant columns (quasi_const_cols). Please note that choosing this threshold must be done with a lot of care and understanding of the problem. For instance, in this case, we know that it's lopsided because only 5% donate, most of whom donate a low amount, so even a tiny percentage of a feature might make an impact, which is why our threshold is so high at 99.9%:

```
thresh = 0.999
quasi_const_cols = []
num_rows = X_train.shape[0]
for col in tqdm(X_train.columns):
    top_val = (
        X_train[col].value_counts() / num_rows
        ).sort_values(ascending=False).values[0]
    if top_val >= thresh:
        quasi_const_cols.append(col)
print(quasi_const_cols)
```

The preceding code should have printed five features, which include the three that were previously obtained. Next, we will deal with another form of irrelevant features: duplicates!

Duplicating features

Usually, when we discuss duplicates with data, we immediately think of duplicate rows, but **duplicate columns** are also problematic. We can find them just as you would find duplicate rows with the pandas `duplicated()` function, except we would transpose the DataFrame first, inversing columns and rows:

```
X_train_transposed = X_train.T
dup_cols = X_train_transposed[
    X_train_transposed.duplicated()].index.tolist()
print(dup_cols)
```

The preceding snippet outputs a list with the two duplicated columns.

Removing unnecessary features

Unlike other feature selection methods, which you should test with models, you can apply basic filter-based feature selection methods right away by removing the features you deemed useless. But just in case, it's good practice to make a copy of the original data. Please note that we don't include constant columns (`all_constant_cols`) in the columns we are to drop (`drop_cols`) because the quasi-constant ones already include them:

```
X_train_orig = X_train.copy()
X_test_orig = X_test.copy()
drop_cols = quasi_const_cols + dup_cols
X_train.drop(labels=drop_cols, axis=1, inplace=True)
X_test.drop(labels=drop_cols, axis=1, inplace=True)
```

Next, we will explore multivariate filter-based methods on the remaining features.

Correlation filter-based methods

Correlation filter-based methods quantify the strength of the relationship between two features. It is useful for feature selection because we might want to filter out extremely correlated features or those that aren't correlated with others at all. Either way, it is a multivariate feature selection method—bivariate, to be precise.

But first, we ought to choose a correlation method:

- **Pearson's correlation coefficient**: Measures the linear correlation between two features. It outputs a coefficient between -1 (negative) and 1 (positive), with 0 meaning no linear correlation. Like linear regression, it assumes linearity, normality, and homoscedasticity—that is, the error term around the linear regression line is similarly sized across all values.

- **Spearman's rank correlation coefficient:** Measures the strength of monotonicity of two features regardless of whether they are linearly related or not. Monotonicity is the degree to which as one feature increases, the other one consistently increases or decreases. It is measured between -1 and 1, with 0 meaning no monotonic correlation. It makes no distribution assumptions and can work with both continuous and discrete features. However, its weakness is with non-monotonic relationships.
- **Kendall's tau correlation coefficient:** Measures the ordinal association between features—that is, it computes the similarity between lists of ordered numbers. It also ranges between -1 and 1, but they mean low and high, respectively. It's useful with discrete features.

The dataset is a mix of continuous and discrete, and we cannot make any linear assumptions about it, so spearman is the right choice. All three can be used with the pandas corr function though:

```
corrs = X_train.corr(method='spearman')
print(corrs.shape)
```

The preceding code should output the shape of the correlation matrix, which is (428, 428). This dimension makes sense because there are 428 features left, and each feature has a relationship with 428 features, including itself.

We can now look for features to remove in the correlation matrix (corrs). Note that to do so, we must establish thresholds. For instance, we can say that an extremely correlated feature has an absolute value coefficient of over 0.99 and less than 0.15 for an uncorrelated feature. With these thresholds in mind, we can find features that are correlated to only one feature and extremely correlated to more than one feature. Why one feature? Because the diagonals in a correlation matrix are always 1 because a feature is always perfectly correlated with itself. The lambda functions in the following code make sure we are accounting for this:

```
extcorr_cols = (abs(corrs) > 0.99).sum(axis=1)[lambda x: x>1]\
.index.tolist()
print(extcorr_cols)
uncorr_cols = (abs(corrs) > 0.15).sum(axis=1)[lambda x: x==1]\
.index.tolist()
print(uncorr_cols)
```

The preceding code outputs the two lists as follows:

```
['MAJOR', 'HHAGE1', 'HHAGE3', 'HHN3', 'HHP1', 'HV1', 'HV2', 'MDMAUD_R',
'MDMAUD_F', 'MDMAUD_A']
['TCODE', 'MAILCODE', 'NOEXCH', 'CHILD03', 'CHILD07', 'CHILD12', 'CHILD18',
'HC15', 'MAXADATE']
```

The first list contains features that are extremely correlated with ones other than themselves. While this is useful to know, you shouldn't remove features from this list without understanding what features they are correlated with and how, as well as with the target. Then, only if redundancy is found, make sure you only remove one of them. The second list is of features uncorrelated to any others than themself, which, in this case, is suspicious given the sheer number of features. That being said, we also should inspect them one by one, especially to measure them against the target to see whether they are redundant. However, we will take a chance and make a feature subset (`corr_cols`) excluding the uncorrelated ones:

```
corr_cols = X_train.columns[
    ~X_train.columns.isin(uncorr_cols)
].tolist()
print(len(corr_cols))
```

The preceding code should output 419. Let's now fit the RF model with only these features. Given that there are still over 400 features, we will use a `max_depth` value of 11. Except for that and a different model name (`mdlname`), it's the same code as before:

```
mdlname = 'rf_11_f-corr'
stime = timeit.default_timer()
reg_mdl = xgb.XGBRFRegressor(
    max_depth=11,
    n_estimators=200,
    seed=rand
)
fitted_mdl = reg_mdl.fit(X_train[corr_cols], y_train)
reg_mdls[mdlname]['num_feat'] = sum(
    reg_mdls[mdlname]['fitted'].feature_importances_ > 0
)
```

Before we compare the results for the preceding model, let's learn about ranking filter methods.

Ranking filter-based methods

Ranking filter-based methods are based on statistical univariate ranking tests, which assess the strength of the dependency between a feature and the target. These are some of the most popular methods:

- **ANOVA F-test: Analysis of Variance** (**ANOVA**) F-test measures the linear dependency between features and the target. As the name suggests, it does this by decomposing the variance. It makes similar assumptions to linear regression, such as normality, independence, and homoscedasticity. In scikit-learn, you can use `f_regression` and `f_classification` for regression and classification, respectively, to rank features by the F-score yielded by the F-test.

- **Chi-square test of independence**: This test measures the association between non-negative categorical variables and binary targets, so it's only suitable for classification problems. In scikit-learn, you can use `chi2`.

- **Mutual information** (**MI**): Unlike the two previous methods, this one is derived from information theory rather than classical statistical hypothesis testing. It's a different name but a concept we have already discussed in this book as the **Kullback-Leibler** (**KL**) **divergence** because it's the KL for feature *X* and target *Y*. The Python implementation in scikit-learn uses a numerically stable and symmetric offshoot of KL called **Jensen-Shannon** (**JS**) divergence instead and leverages k-nearest neighbors to compute distances. Features can be ranked by MI with `mutual_info_regression` and `mutual_info_classif` for regression and classification, respectively.

Of the three options mentioned, the one that is most appropriate for this dataset is MI because we cannot assume linearity among our features, and most of them aren't categorical either. We can try classification with a threshold of $0.68, which at least covers the cost of sending the mailer. To that end, we must first create a binary classification target (`y_train_class`) with that threshold:

```
y_train_class = np.where(y_train > 0.68, 1, 0)
```

Next, we can use `SelectKBest` to get the top 160 features according to **MI Classification** (**MIC**). We then employ `get_support()` to obtain a Boolean vector (or mask), which tells us which features are in the top 160, and we subset the list of features with this mask:

```
mic_selection = SelectKBest(
    mutual_info_classif, k=160).fit(X_train, y_train_class)
mic_cols = X_train.columns[mic_selection.get_support()].tolist()
print(len(mic_cols))
```

The preceding code should confirm that there are 160 features in the `mic_cols` list. Incidentally, this is an arbitrary number. Ideally, we could test different thresholds for the classification target and *k*s for the MI, looking for the model that achieved the highest profit lift while underfitting the least. Next, we can fit the RF model as we've done before with the MIC features. This time, we will use a max depth of 5 because there are significantly fewer features:

```
mdlname = 'rf_5_f-mic'
stime = timeit.default_timer()
reg_mdl = xgb.XGBRFRegressor(max_depth=5, n_estimators=200, seed=rand)
fitted_mdl = reg_mdl.fit(X_train[mic_cols], y_train)
reg_mdls[mdlname]['num_feat'] = sum(
    reg_mdls[mdlname]['fitted'].feature_importances_ > 0
)
```

Now, let's plot the profits for **Test** and **Train** as we did in *Figure 10.3*, but for the MIC model. It will produce what's shown in *Figure 10.5*:
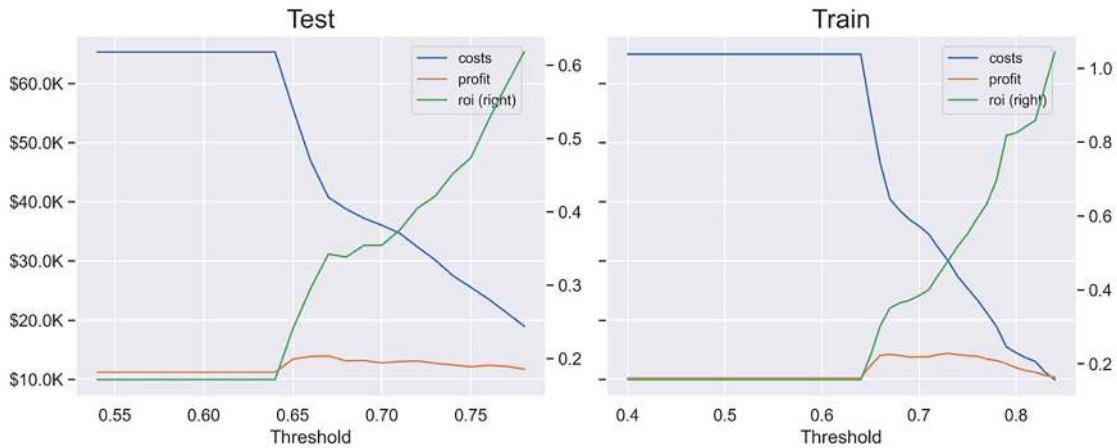
Figure 10.5: Comparison between profit, costs, and ROI for the test and train datasets for a model with MIC features across thresholds

In *Figure 10.5*, you can tell that there is quite a bit of difference between **Test** and **Train**, yet similarities indicate minimal overfitting. For instance, the highest profitability can be found between 0.66 and 0.75 for **Train**, and while **Test** is mostly between 0.66 and 0.7, it only gradually decreases afterward.

Although we have visually examined the MIC model, it's nice to have some reassurance by looking at raw metrics. Next, we will compare all the models we have trained so far using consistent metrics.

Comparing filter-based methods

We have been saving metrics into a dictionary (`reg_mdls`), which we easily convert to a DataFrame and output as we have done before, but this time we sort by `max_profit_test`:

```
display_mdl_metrics(reg_mdls, 'max_profit_test')
```

The preceding snippet generated what is shown in *Figure 10.6*. It is evident that the filter MIC model is the least overfitted of all. It ranked higher than more complex models with more features and took less time to train than any model. Its speed is an advantage for hyperparameter tuning. What if we wanted to find the best classification target thresholds or MIC *k*? We won't do this now, but we would likely get a better model if we ran every combination, but it would take time to do and even more with more features:

	depth	fs	rmse_train	rmse_test	max_profit_train	max_profit_test	max_roi	min_costs	speed	total_feat	num_feat
rf_11_all	11	all	3.99	4.69	$19,904	$15,142	0.76	$14,928	2.74	435	398
rf_10_all	10	all	4.05	4.68	$18,604	$14,987	0.78	$14,396	2.53	435	383
rf_12_all	12	all	3.94	4.69	$21,522	$14,933	0.77	$14,532	2.89	435	415
rf_11_f-corr	11	f-corr	3.98	4.67	$19,924	$14,895	0.77	$14,593	2.47	419	404
rf_9_all	9	all	4.10	4.68	$17,452	$14,778	0.80	$13,997	2.20	435	346
rf_8_all	8	all	4.14	4.67	$16,440	$14,563	0.73	$15,309	1.98	435	315
rf_7_all	7	all	4.18	4.66	$15,435	$14,186	0.66	$17,165	1.71	435	277
rf_5_f-mic	5	f-mic	4.31	4.60	$14,367	$13,944	0.62	$18,971	0.41	160	105
rf_6_all	6	all	4.23	4.65	$14,655	$13,851	0.59	$19,305	1.49	435	240
rf_5_all	5	all	4.27	4.64	$14,242	$13,752	0.59	$19,199	1.18	435	201
rf_4_all	4	all	4.32	4.64	$13,716	$13,262	0.53	$22,392	1.00	435	160

Figure 10.6: Comparing metrics for all base models and filter-based feature-selected models

In *Figure 10.6*, we can tell that the correlation filter model (`rf_11_f-corr`) performs worse than the model with more features and an equal amount of `max_depth` (`rf_11_all`), which suggests that we must have removed an important feature. As cautioned in that section, the problem with blindly setting thresholds and removing anything above it is that you can inadvertently remove something useful. Not all extremely correlated and uncorrelated features are useless, so further inspection is required. Next, we will explore some embedded methods that, when combined with cross-validation, require less oversight.

Exploring embedded feature selection methods

Embedded methods exist within models themselves by naturally selecting features during training. You can leverage the intrinsic properties of any model that has them to capture the features selected:

- **Tree-based models:** For instance, we have used the following code many times to count the number of features used by the RF models, which is evidence of feature selection naturally occurring in the learning process:

```
sum(reg_mdls[mdlname]['fitted'].feature_importances_ > 0)
```

  XGBoost's RF uses gain by default, which is the average decrease in error in all splits where it used the feature to compute feature importance. We can increase the threshold above 0 to select even fewer features according to their relative contribution. However, by constraining the trees' depth, we forced the model to choose even fewer features already.

- **Regularized models with coefficients:** We will study this further in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*, but many model classes can incorporate penalty-based regularization, such as L1, L2, and elastic net. However, not all of them have intrinsic parameters such as coefficients that can be extracted to determine which features were penalized.

This section will only cover regularized models given that we are using a tree-based model already. It's best to leverage different model classes to get different perspectives on what features matter the most.

We covered some of these models in *Chapter 3, Interpretation Challenges*, but these are a few model classes that incorporate penalty-based regularization and output feature-specific coefficients:

- **Least Absolute Shrinkage and Selection Operator** (**LASSO**): Because it uses L1 penalty in the loss function, LASSO can set coefficients to 0.
- **Least-Angle Regression** (**LARS**): Similar to LASSO but is vector-based and is more suitable for high-dimensional data. It is also fairer toward equally correlated features.
- **Ridge regression**: Uses L2 penalty in the loss function and because of this, can only shrink coefficients of irrelevance close to 0 but not to 0.
- **Elastic net regression**: Uses a mix of both L1 and L2 norms as penalties.
- **Logistic regression**: Contingent on the solver, it can handle L1, L2, or elastic net penalties.

There are also several variations of the preceding models, such as **LASSO LARS**, which is a LASSO fit using the LARS algorithm, or even **LASSO LARS IC**, which is the same but uses AIC or BIC criteria for the model section:

- **Akaike's Information Criteria** (**AIC**): A relative goodness of fit measure founded in information theory
- **Bayesian Information Criteria** (**BIC**): Has a similar formula to AIC but has a different penalty term

OK, now let's use `SelectFromModel` to extract top features from a LASSO model. We will use `LassoCV` because it can automatically cross-validate to find the optimal penalty strength. Once you fit it, we can get the feature mask with `get_support()`. We can then print the number of features and list of features:

```
lasso_selection = SelectFromModel(
    LassoCV(n_jobs=-1, random_state=rand)
)
lasso_selection.fit(X_train, y_train)
lasso_cols = X_train.columns[lasso_selection.get_support()].tolist()
print(len(lasso_cols))
print(lasso_cols)
```

The preceding code outputs the following:

```
7
['ODATEDW', 'TCODE', 'POP901', 'POP902', 'HV2', 'RAMNTALL', 'MAXRDATE']
```

Now, let's try the same but with `LassoLarsCV`:

```
llars_selection = SelectFromModel(LassoLarsCV(n_jobs=-1))
llars_selection.fit(X_train, y_train)
llars_cols = X_train.columns[llars_selection.get_support()].tolist()
```

```
print(len(llars_cols))
print(llars_cols)
```

The preceding snippet produces the following output:

```
8
['RECPGVG', 'MDMAUD', 'HVP3', 'RAMNTALL', 'LASTGIFT', 'AVGGIFT', 'MDMAUD_A',
'DOMAIN_SOCIALCLS']
```

LASSO shrunk coefficients for all but seven features to 0, and LASSO LARS did the same but for eight. However, notice how there's no overlap between both lists! OK, so let's try incorporating AIC model selection into LASSO LARS with `LassoLarsIC`:

```
llarsic_selection = SelectFromModel(LassoLarsIC(criterion='aic'))
llarsic_selection.fit(X_train, y_train)
llarsic_cols = X_train.columns[
    llarsic_selection.get_support()
].tolist()
print(len(llarsic_cols))
print(llarsic_cols)
```

The preceding snippet generates the following output:

```
111
['TCODE', 'STATE', 'MAILCODE', 'RECINHSE', 'RECP3', 'RECPGVG', 'RECSWEEP',...,
'DOMAIN_URBANICITY', 'DOMAIN_SOCIALCLS', 'ZIP_LON']
```

It's the same algorithm but with a different method for selecting the value of the regularization parameter. Note how this less-conservative approach expands the number of features to 111. Now, so far, all of the methods we have used have the L1 norm. Let's try one with L2—more specifically, L2-penalized logistic regression. We do exactly what we did before, but this time, we fit with the binary classification targets (`y_train_class`):

```
log_selection = SelectFromModel(
    LogisticRegression(
        C=0.0001,
        solver='sag',
        penalty='l2',
        n_jobs=-1,
        random_state=rand
    )
)
log_selection.fit(X_train, y_train_class)
log_cols = X_train.columns[log_selection.get_support()].tolist()
print(len(log_cols))
print(log_cols)
```

The preceding code produces the following output:

```
87
['ODATEDW', 'TCODE', 'STATE', 'POP901', 'POP902', 'POP903', 'ETH1', 'ETH2',
'ETH5', 'CHIL1', 'HHN2',..., 'AMT_7', 'ZIP_LON']
```

Now that we have a few feature subsets to test, we can place their names into a list (`fsnames`) and the feature subset lists into another list (`fscols`):

```
fsnames = ['e-lasso', 'e-llars', 'e-llarsic', 'e-logl2']
fscols = [lasso_cols, llars_cols, llarsic_cols, log_cols]
```

We can then iterate across all list names and fit and evaluate our `XGBRFRegressor` model as we have done before, but increasing `max_depth` at every iteration:

```python
def train_mdls_with_fs(reg_mdls, fsnames, fscols, depths):
    for i, fsname in tqdm(enumerate(fsnames), total=len(fsnames)):
        depth = depths[i]
        cols = fscols[i]
        mdlname = 'rf_'+str(depth)+'_'+fsname
        stime = timeit.default_timer()
        reg_mdl = xgb.XGBRFRegressor(
            max_depth=depth, n_estimators=200, seed=rand
        )
        fitted_mdl = reg_mdl.fit(X_train[cols], y_train)
        reg_mdls[mdlname]['num_feat'] = sum(
            reg_mdls[mdlname]['fitted'].feature_importances_ > 0
        )

train_mdls_with_fs(reg_mdls, fsnames, fscols, [3, 4, 5, 6])
```

Now, let's see how our embedded feature-selected models fare in comparison to the filtered ones. We will rerun the code we ran to output what was shown in *Figure 10.6*. This time, we will get what is shown in *Figure 10.7*:

	depth	fs	rmse_train	rmse_test	max_profit_train	max_profit_test	max_roi	min_costs	speed	total_feat	num_feat
rf_11_all	11	all	3.99	4.69	$19,904	$15,142	0.76	$14,928	2.74	435	398
rf_10_all	10	all	4.05	4.68	$18,604	$14,987	0.78	$14,396	2.53	435	383
rf_12_all	12	all	3.94	4.69	$21,522	$14,933	0.77	$14,532	2.89	435	415
rf_11_f-corr	11	f-corr	3.98	4.67	$19,924	$14,895	0.77	$14,593	2.47	419	404
rf_9_all	9	all	4.10	4.68	$17,452	$14,778	0.80	$13,997	2.20	435	346
rf_5_e-llarsic	5	e-llarsic	4.28	4.45	$15,168	$14,768	0.56	$20,441	0.30	111	87
rf_8_all	8	all	4.14	4.67	$16,440	$14,563	0.73	$15,309	1.98	435	315
rf_6_e-logl2	6	e-logl2	4.28	4.60	$15,353	$14,199	0.67	$16,904	0.31	87	84
rf_7_all	7	all	4.18	4.66	$15,435	$14,186	0.66	$17,165	1.71	435	277
rf_5_f-mic	5	f-mic	4.31	4.60	$14,367	$13,944	0.62	$18,971	0.41	160	105
rf_6_all	6	all	4.23	4.65	$14,655	$13,851	0.59	$19,305	1.49	435	240
rf_5_all	5	all	4.27	4.64	$14,242	$13,752	0.59	$19,199	1.18	435	201
rf_4_e-llars	4	e-llars	4.36	4.45	$14,014	$13,633	0.52	$22,906	0.04	8	8
rf_4_all	4	all	4.32	4.64	$13,716	$13,262	0.53	$22,392	1.00	435	160
rf_3_e-lasso	3	e-lasso	4.46	4.49	$14,167	$12,930	0.51	$22,249	0.03	7	7

Figure 10.7: Comparing metrics for all base models and filter-based and embedded feature-selected models

According to *Figure 10.7*, three of the four embedded methods we tried produced models with the lowest test RMSE (`rf_5_e-llarsic`, `rf_e-lasso`, and `rf_4_e-llars`). They also all trained much faster than the others and are more profitable than any other model of equal complexity. One of them (`rf_5_e-llarsic`) is even highly profitable. Compare this with `rf_9_all` with similar test profitability to see how performance diverges from the training data.

Discovering wrapper, hybrid, and advanced feature selection methods

The feature selection methods studied so far are computationally inexpensive because they require no model fitting or fitting simpler white-box models. In this section, we will learn about other, more exhaustive methods with many possible tuning options. The categories of methods included here are as follows:

- **Wrapper:** Exhaustively searches for the best subset of features by fitting an ML model using a search strategy that measures improvement on a metric.
- **Hybrid:** A method that combines embedded and filter methods with wrapper methods.
- **Advanced:** A method that doesn't fall into any of the previously discussed categories. Examples include dimensionality reduction, model-agnostic feature importance, and **Genetic Algorithms (GAs)**.

And now, let's get started with wrapper methods!

Wrapper methods

The concept behind **wrapper methods** is reasonably simple: evaluate different subsets of features on the ML model and choose the one that achieves the best score in a predetermined objective function. What varies here is the search strategy:

- **Sequential Forward Selection** (**SFS**): This approach begins without a feature and adds one, one at a time.
- **Sequential Forward Floating Selection** (**SFFS**): The same as the previous except for every feature it adds, it can remove one, as long as the objective function increases.
- **Sequential Backward Selection** (**SBS**): This process begins with all features present and eliminates one feature at a time.
- **Sequential Floating Backward Selection** (**SFBS**): The same as the previous except for every feature it removes, it can add one, as long as the objective function increases.
- **Exhaustive Feature Selection** (**EFS**): This approach seeks all possible combinations of features.
- **BiDirectional Search** (**BDS**): This last one simultaneously allows both forward and backward function selection to get one unique solution.

These methods are greedy algorithms because they solve the problem piece by piece, choosing pieces based on their immediate benefit. Even though they may arrive at a global maximum, they take an approach more suited for finding local maxima. Depending on the number of features, they might be too computationally expensive to be practical, especially EFS, which grows exponentially. Another important distinction is the difference between forward methods' accuracy increases as features are added and backward ones, monitor accuracy decreases as features are removed. To allow for shorter search times, we will do two things:

1. Start our search with the features collectively selected by other methods to have a smaller feature space to choose from. To that end, we combine feature lists from several methods into a single `top_cols` list:

   ```
   top_cols = list(set(mic_cols).union(set(llarsic_cols)\
   ).union(set(log_cols)))
   len(top_cols)
   ```

2. Sample our datasets so that ML models speed up. We can use `np.random.choice` to do a random selection of row indexes without replacement:

   ```
   sample_size = 0.1
   sample_train_idx = np.random.choice(
       X_train.shape[0],
       math.ceil(X_train.shape[0]*sample_size),
       replace=False
   )
   ```

```
    sample_test_idx = np.random.choice(
        X_test.shape[0],
        math.ceil(X_test.shape[0]*sample_size),
        replace=False
    )
```

Out of the wrapper methods presented, we will only perform SFS, given how time-consuming they are. Still, with an even smaller dataset, you can try the other options, which the `mlextend` library also supports.

Sequential forward selection (SFS)

The first argument of a wrapper method is an unfitted estimator (a model). In `SequentialFeatureSelector`, we are placing a `LinearDiscriminantAnalysis` model. Other arguments include the direction (`forward=true`), whether it's floating (`floating=False`), which means it might undo the previous exclusion or inclusion of a feature, the number of features we wish to select (`k_features=27`), the number of cross-validations (`cv=3`), and the loss function to use (`scoring=f1`). Some recommended optional arguments to enter are the verbosity (`verbose=2`) and the number of jobs to run in parallel (`n_jobs=-1`). Since it could take a while, we'll definitely want it to output something and use as many processors as possible:

```
sfs_lda = SequentialFeatureSelector(
    LinearDiscriminantAnalysis(n_components=1),
    forward=True,
    floating=False,
    k_features=100,
    cv=3,
    scoring='f1',
    verbose=2,
    n_jobs=-1
)
sfs_lda = sfs_lda.fit(X_train.iloc[sample_train_idx][top_cols],\
                      y_train_class[sample_train_idx])
sfs_lda_cols = X_train.columns[list(sfs_lda.k_feature_idx_)].tolist()
```

Once we fit the SFS, it will return the index of features that have been selected with `k_feature_idx_`, and we can use those to subset the columns and obtain the list of feature names.

Hybrid methods

Starting with 435 features, there are over 10^{42} combinations of 27 feature subsets alone! So, you can see how EFS would be impractical in such a large feature space. Therefore, except for EFS on the entire dataset, wrapper methods will invariably take some shortcuts to select the features. Whether you are going forward, backward, or both, as long as you are not assessing every single combination of features, you could easily miss out on the best one.

However, we can leverage the more rigorous, exhaustive search approach of wrapper methods with filter and embedded methods' efficiency. The result of this is **hybrid methods**. For instance, you could employ filter or embedded methods to derive only the top 10 features and perform EFS or SBS on only those.

Recursive Feature Elimination (RFE)

Another, more common approach is something such as SBS, but instead of removing features based on improving a metric alone, using the model's intrinsic parameters to rank the features and only removing the least ranked. The name of this approach is **Recursive Feature Elimination** (**RFE**), and it is a hybrid between embedded and wrapper methods. We can only use models with `feature_importances_` or coefficients (`coef_`) because this is how the method knows what features to remove. Model classes in scikit-learn with these attributes are classified under `linear_model`, `tree`, and `ensemble`. Also, scikit-learn-compatible versions of XGBoost, LightGBM, and CatBoost also have `feature_importances_`.

We will use the cross-validated version of RFE because it's more reliable. `RFECV` takes the estimator first (`LinearDiscriminantAnalysis`). We can then define `step`, which sets how many features it should remove in every iteration, the number of cross-validations (`cv`), and the metric used for evaluation (`scoring`). Lastly, it is recommended to set the verbosity (`verbose=2`) and leverage as many processors as possible (`n_jobs=-1`). To speed it up, we will use a sample again for the training and start with the 267 for `top_cols`:

```
rfe_lda = RFECV(
    LinearDiscriminantAnalysis(n_components=1),
    step=2, cv=3, scoring='f1', verbose=2, n_jobs=-1
)
rfe_lda.fit(
    X_train.iloc[sample_train_idx][top_cols],
    y_train_class[sample_train_idx]
)
rfe_lda_cols = np.array(top_cols)[rfe_lda.support_].tolist()
```

Next, we will try different methods that don't relate to the main three feature selection categories: filter, embedded, and wrapper.

Advanced methods

Many methods can be categorized under advanced feature selection methods, including the following subcategories:

- **Model-agnostic feature importance**: Any feature importance method covered in *Chapter 4, Global Model-Agnostic Interpretation Methods*, can be used to obtain the top features of a model for feature selection purposes.

- **Genetic algorithms:** This is a wrapper method in the sense that it "wraps" a model assessing predictive performance across many feature subsets. However, unlike the wrapper methods we examined, it doesn't make the most locally optimal choice. It's more optimized to work with large feature spaces. It's called genetic because it's inspired by biology—natural selection, specifically.

- **Dimensionality reduction:** Some dimensionality reduction methods, such as **Principal Component Analysis** (**PCA**), can return explained variance on a feature basis. For others, such as factor analysis, it can be derived from other outputs. Explained variance can be used to rank features.

- **Autoencoders:** We won't delve into this one, but deep learning can be leveraged for feature selection with autoencoders. This method has many variants you can find in Google Scholar and is not widely adopted in industry.

We will briefly cover the first two in this section so you can understand how they can be implemented. Let's dive right in!

Model-agnostic feature importance

A popular model-agnostic feature importance method that we have used throughout this book is SHAP, and it has many properties that make it more reliable than other methods. In the following code, we can take our best model and extract `shap_values` for it using `TreeExplainer`:

```
fitted_rf_mdl = reg_mdls['rf_11_all']['fitted']
shap_rf_explainer = shap.TreeExplainer(fitted_rf_mdl)
shap_rf_values = shap_rf_explainer.shap_values(
    X_test_orig.iloc[sample_test_idx]
)
shap_imps = pd.DataFrame(
    {'col':X_train_orig.columns, 'imp':np.abs(shap_rf_values).mean(0)}
).sort_values(by='imp',ascending=False)
shap_cols = shap_imps.head(120).col.tolist()
```

Then, the average for the absolute value of the SHAP values across the first dimension is what provides us with a ranking for each feature. We put this value in a DataFrame and sort it as we did for PCA. Lastly, also take the top 120 and place them in a list (`shap_cols`).

Genetic algorithms

GAs are a stochastic global optimization technique inspired by natural selection, which wrap a model much like wrapper methods do. However, they don't follow a sequence on a step-by-step basis. GAs don't have iterations but generations, which include populations of chromosomes. Each chromosome is a binary representation of your feature space, where 1 means to select a feature and 0 to not. Each generation is produced with the following operations:

- **Selection:** Like with natural selection, this is partially random (exploration) and partially based on what has already worked (exploitation). What has worked is its fitness. Fitness is assessed with a "scorer" much like wrapper methods. Poor fitness chromosomes are removed, whereas good ones get to reproduce through "crossover."

- **Crossover**: Randomly, some good bits (or features) of each parent go to a child.
- **Mutation**: Even when a chromosome has proved effective, given a low mutation rate, it will occasionally mutate or flip one of its bits, in other words, features.

The Python implementation we will use has many options. We won't explain all of them here, but they are documented well in the code should you be interested. The first attribute is the estimator. We can also define the cross-validation iterations (`cv=3`) and `scoring` to determine whether chromosomes are fit. There are some important probabilistic properties, such as the probability for a mutated bit (`mutation_probability`) and that bits will get exchanged (`crossover_probability`). Generation-wise, `n_gen_no_change` provides a means for early stopping if generations haven't improved, and `generations` by default is 40, but we will use 5. We can fit `GeneticSelectionCV` as you would any model. It can take a while, so it is best to define the verbosity and allow it to use all the processing capacity. Once finished, we can use the Boolean mask (`support_`) to subset the features:

```python
ga_rf = GAFeatureSelectionCV(
    RandomForestRegressor(random_state=rand, max_depth=3),
    cv=3,
    scoring='neg_root_mean_squared_error',
    crossover_probability=0.8,
    mutation_probability=0.1,
    generations=5, n_jobs=-1
)
ga_rf = ga_rf.fit(
    X_train.iloc[sample_train_idx][top_cols].values,
    y_train[sample_train_idx]
)
ga_rf_cols = np.array(top_cols)[ga_rf.best_features_].tolist()
```

OK, now that we have covered a wide variety of wrapper, hybrid, and advanced feature selection methods in this section, let's evaluate all of them at once and compare the results.

Evaluating all feature-selected models

As we have done with embedded methods, we can place feature subset names (`fsnames`), lists (`fscols`), and corresponding `depths` in lists:

```python
fsnames = ['w-sfs-lda', 'h-rfe-lda', 'a-shap', 'a-ga-rf']
fscols = [sfs_lda_cols, rfe_lda_cols, shap_cols, ga_rf_cols]
depths = [5, 6, 5, 6]
```

Then, we can use the two functions we created to first iterate across all feature subsets, training and evaluating a model with them. Then the second function outputs the results of the evaluation in a DataFrame with previously trained models:

```python
train_mdls_with_fs(reg_mdls, fsnames, fscols, depths)
display_mdl_metrics(reg_mdls, 'max_profit_test', max_depth=7)
```

This time, we are limiting the models to those with no more than a depth of 7 since those are very overfitted. The result of the snippet is depicted in *Figure 10.8*:

	depth	fs	rmse_train	rmse_test	max_profit_train	max_profit_test	max_roi	min_costs	speed	total_feat	num_feat
rf_5_e-llarsic	5	e-llarsic	4.28	4.45	$15,168	$14,768	0.56	$20,441	0.30	111	87
rf_6_h-rfe-lda	6	h-rfe-lda	4.28	4.50	$15,705	$14,410	0.68	$16,542	0.47	145	115
rf_6_e-logl2	6	e-logl2	4.28	4.60	$15,353	$14,199	0.67	$16,904	0.31	87	84
rf_6_a-ga-rf	6	a-ga-rf	4.26	4.67	$15,710	$14,004	0.72	$15,987	0.47	134	111
rf_5_f-mic	5	f-mic	4.31	4.60	$14,367	$13,944	0.62	$18,971	0.41	160	105
rf_6_all	6	all	4.23	4.65	$14,655	$13,851	0.59	$19,305	1.49	435	240
rf_5_all	5	all	4.27	4.64	$14,242	$13,752	0.59	$19,199	1.18	435	201
rf_4_e-llars	4	e-llars	4.36	4.45	$14,014	$13,633	0.52	$22,906	0.04	8	8
rf_5_a-shap	5	a-shap	4.28	4.51	$14,068	$13,350	0.59	$18,935	0.35	120	102
rf_5_w-sfs-lda	5	w-sfs-lda	4.33	4.47	$13,763	$13,262	0.46	$24,553	0.29	100	81
rf_4_all	4	all	4.32	4.64	$13,716	$13,262	0.53	$22,392	1.00	435	160
rf_3_e-lasso	3	e-lasso	4.46	4.49	$14,167	$12,930	0.51	$22,249	0.03	7	7

Figure 10.8: Comparing metrics for all feature-selected models

Figure 10.8 shows how feature-selected models are more profitable than ones that include all the features compared at the same depths. Also, the embedded LASSO LARS with AIC (`e-llarsic`) method and the MIC (`f-mic`) filter method outperform all wrapper, hybrid, and advanced methods with the same depths. Still, we also impeded these methods by using a sample of the training dataset, which was necessary to speed up the process. Maybe they would have outperformed the top ones otherwise. However, the three feature selection methods that follow are pretty competitive:

- RFE with LDA: Hybrid method (`h-rfe-lda`)
- Logistic regression with L2 regularization: Embedded method (`e-logl2`)
- GAs with RF: Advanced method (`a-ga-rf`)

It would make sense to spend many days running many variations of the methods reviewed in this book. For instance, perhaps RFE with L1 regularized logistic regression or GA with support vector machines with additional mutation yields the best model. There are so many different possibilities! Nevertheless, if you were forced to make a recommendation based on *Figure 10.8*, by profit alone, the 111-feature `e-llarsic` is the best option, but it also has higher minimum costs and lower maximum ROI than any of the top models. There's a trade-off. And even though it has among the highest test RMSEs, the 160-feature model (`f-mic`) has a similar spread between max profit train and test and beat it in max ROI and min costs. Therefore, these are the two reasonable options. But before making a final determination, profitability would have to be compared side by side across different thresholds to assess when each model can make the most reliable predictions and at what costs and ROIs.

Considering feature engineering

Let's assume that the non-profit has chosen to use the model whose features were selected with LASSO LARS with AIC (`e-llarsic`) but would like to evaluate whether you can improve it further. Now that you have removed over 300 features that might have only marginally improved predictive performance but mostly added noise, you are left with more relevant features. However, you also know that 8 features selected by `e-llars` produced the same amount of RMSE as the 111 features. This means that while there's something in those extra features that improves profitability, it does not improve the RMSE.

From a feature selection standpoint, many things can be done to approach this problem. For instance, examine the overlap and difference of features between `e-llarsic` and `e-llars`, and do feature selection variations strictly on those features to see whether the RMSE dips on any combination while keeping or improving on current profitability. However, there's also another possibility, which is feature engineering. There are a few important reasons you would want to perform feature engineering at this stage:

- **Make model interpretation easier to understand:** For instance, sometimes features have a scale that is not intuitive, or the scale is intuitive, but the distribution makes it hard to understand. As long as transformations to these features don't worsen model performance, there's value in transforming the features to understand the outputs of interpretation methods better. As you train models on more engineered features, you realize what works and why it does. This will help you understand the model and, more importantly, the data.

- **Place guardrails on individual features:** Sometimes, features have an uneven distribution, and models tend to overfit in sparser areas of the feature's histogram or where influential outliers exist.

- **Clean up counterintuitive interactions:** Some interactions that models find make no sense and only exist because the features correlate, but not for the right reasons. They could be confounding variables or perhaps even redundant ones (such as the one we found in *Chapter 4, Global Model-Agnostic Interpretation Methods*). You could decide to engineer an interaction feature or remove a redundant one.

In reference to the last two reasons, we will examine feature engineering strategies in more detail in *Chapter 12, Monotonic Constraints and Model Tuning for Interpretability*. This section will focus on the first reason, particularly because it's a good place to start since it will allow you to understand the data better until you know it well enough to make more transformational changes.

So, we are left with 111 features but have no idea how they relate to the target or each other. The first thing we ought to do is run a feature importance method. We can use SHAP's `TreeExplainer` on the `e-llarsic` model. An advantage of `TreeExplainer` is that it can compute SHAP interaction values, `shap_interaction_values`. Instead of outputting an array of (`N`, `111`) dimensions where *N* is the number of observations as `shap_values` does, it will output (`N`, `111`, `111`). We can produce a `summary_plot` graph with it that ranks both individual features and interactions. The only difference for interaction values is you use `plot_type="compact_dot"`:

```
winning_mdl = 'rf_5_e-llarsic'
fitted_rf_mdl = reg_mdls[winning_mdl]['fitted']
shap_rf_explainer = shap.TreeExplainer(fitted_rf_mdl)
shap_rf_interact_values = \
    shap_rf_explainer.shap_interaction_values(
        X_test.iloc[sample_test_idx][llarsic_cols]
    )
shap.summary_plot(
    shap_rf_interact_values,
    X_test.iloc[sample_test_idx][llarsic_cols],
    plot_type="compact_dot",
    sort=True
)
```

The preceding snippet produces the SHAP interaction summary plot shown in *Figure 10.9*:
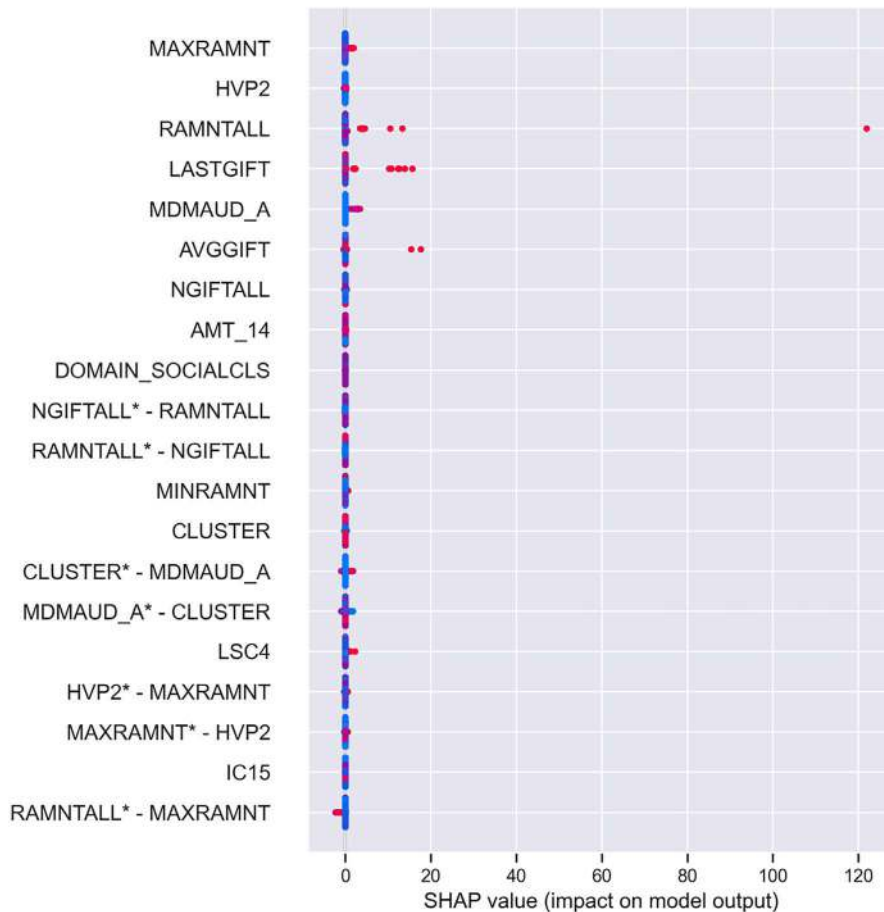
Figure 10.9: SHAP interaction summary plot

We can read *Figure 10.9* as we would any summary plot except it includes bivariate interactions twice—first with one feature and then with another. For instance, `MDMAUD_A* - CLUSTER` is the interaction SHAP values for that interaction from `MDMAUD_A`'s perspective, so the feature values correspond to that feature alone, but the SHAP values are for the interaction. One thing that we can agree on here is that the plot is hard to read given the scale of the importance values and complexity of comparing bivariate interactions in no order. We will address this later.

Throughout this book, chapters with tabular data have started with a data dictionary. This one was an exception, given that there were 435 features to begin with. Now, it makes sense to at the very least understand what the top features are. The complete data dictionary can be found at `https://kdd.ics.uci.edu/databases/kddcup98/epsilon_mirror/cup98dic.txt`, but some of the features have already been changed because of categorical encoding, so we will explain them in more detail here:

- `MAXRAMNT`: Continuous, the dollar amount of the largest gift to date
- `HVP2`: Discrete, percentage of homes with a value of >= $150,000 in the neighborhoods of donors (values between 0 and 100)
- `LASTGIFT`: Continuous, the dollar amount of the most recent gift
- `RAMNTALL`: Continuous, the dollar amount of lifetime gifts to date
- `AVGGIFT`: Continuous, the average dollar amount of gifts to date
- `MDMAUD_A`: Ordinal, the donation amount code for donors who have given a $100 + gift at any time in their giving history (values between 0 and 3, and -1 for those who have never exceeded $100). The amount code is the third byte of an **RFA** (**recency/frequency/amount**) major customer matrix code, which is the amount given. The categories are as follows:

  0: Less than $100 (low dollar)

  1: $100 – 499 (core)

  2: $500 – 999 (major)

  3: $1,000 + (top)

- `NGIFTALL`: Discrete, number of lifetime gifts to date
- `AMT_14`: Ordinal, donation amount code of the RFA for the 14th previous promotion (2 years prior), which corresponds to the last dollar amount given back then:

  0: $0.01 – 1.99

  1: $2.00 – 2.99

  2: $3.00 – 4.99

  3: $5.00 – 9.99

  4: $10.00 – 14.99

  5: $15.00 – 24.99

  6: $25.00 and above

- • `DOMAIN_SOCIALCLS`: Nominal, **Socio-Economic Status** (**SES**) of the neighborhood, which combines with `DOMAIN_URBANICITY` (0: Urban, 1: City, 2: Suburban, 3: Town, and 4: Rural), meaning the following:

  1: Highest SES

  2: Average SES, except above average for urban communities

  3: Lowest SES, except below average for urban communities

  4: Lowest SES for urban communities only

- • `CLUSTER`: Nominal, code indicating which cluster group the donor falls in
- • `MINRAMNT`: Continuous, dollar amount of the smallest gift to date
- • `LSC2`: Discrete, percentage of Spanish-speaking families in the donor's neighborhood (values between 0 and 100)
- • `IC15`: Discrete, percentage of families with an income of < $15,000 in the donor's neighborhood (values between 0 and 100)

The following insights can be distilled from the preceding dictionary and *Figure 10.9*:

- • **Gift amounts prevail:** Seven of the top features pertain to gift amounts, whether it's a total, min, max, average, or last. If you include the count of gifts (`NGIFTALL`), there are eight features involving donation history, making complete sense. So, why is this relevant? Because they are likely highly correlated and understanding how could hold the keys to improving the model. Perhaps other features can be created that distill these relationships much better.
- • **High values of continuous gift amount features have high SHAP values:** Plot a box plot of any of those features like this, `plt.boxplot(X_test.MAXRAMNT)`, and you'll see how right-skewed these features are. Perhaps a transformation such as breaking them into bins—called "discretization"—or using a different scale, such as logarithmic (try `plt.boxplot(np.log(X_test.MAXRAMNT))`), can help interpret these features but also help find the pockets where the likelihood of donation dramatically increases.
- • **Relationship with the 14th previous promotion:** What happened two years before they made that promotion connect to the one denoted in the dataset labels? Were the promotional materials similar? Is there a seasonality factor occurring at the same time every couple of years? Maybe you can engineer a feature that better identifies this phenomenon.
- • **Inconsistent classifications:** `DOMAIN_SOCIALCLS` has different categories depending on the `DOMAIN_URBANICITY` value. We can make this consistent by using all five categories in the scale (Highest, Above Average, Average, Below Average, and Lowest) even if this means non-urban donors would be using only three. The advantage to doing this would be easier interpretation, and it's highly unlikely it would adversely impact the model's performance.

The SHAP interaction summary plot can be useful for identifying feature and interaction rankings and some commonalities between them, but in this case (see *Figure 10.9*), it was hard to read. But to dig deeper into interactions, you first need to quantify their impact. To this end, let's create a heatmap with only the top interactions as measured by their mean absolute SHAP value (`shap_rf_interact_avgs`). We should then set all the diagonal values to 0 (`shap_rf_interact_avgs_nodiag`) because these aren't interactions but feature SHAP values, and it's easier to observe the interactions without them. We can place this matrix in a DataFrame, but it's a DataFrame of 111 columns and 111 rows, so to filter it by those features with the most interactions, we sum them and rank them with `scipy's` `rankdata`. Then, we use the ranking to identify the 12 most interactive features (`most_interact_cols`) and subset the DataFrame by them. Finally, we plot the DataFrame as a heatmap:

```python
shap_rf_interact_avgs = np.abs(shap_rf_interact_values).mean(0)
shap_rf_interact_avgs_nodiag = shap_rf_interact_avgs.copy()
np.fill_diagonal(shap_rf_interact_avgs_nodiag, 0)
shap_rf_interact_df = pd.DataFrame(shap_rf_interact_avgs_nodiag)
shap_rf_interact_df.columns = X_test[llarsic_cols].columns
shap_rf_interact_df.index = X_test[llarsic_cols].columns
shap_rf_interact_ranks = 112 -rankdata(np.sum(
    shap_rf_interact_avgs_nodiag, axis=0)
)
most_interact_cols = shap_rf_interact_df.columns[
    shap_rf_interact_ranks < 13
]
shap_rf_interact_df = shap_rf_interact_df.loc[
most_interact_cols,most_interact_cols
]
sns.heatmap(
    shap_rf_interact_df,
    cmap='Blues',
    annot=True,
    annot_kws={'size':10},
    fmt='.3f',
    linewidths=.5
)
```

The preceding snippet outputs what is shown in *Figure 10.10*. It depicts the most salient feature interactions according to SHAP interaction absolute mean values. Note that these are averages, so given how right-skewed most of these features are, it is likely much higher for many observations. However, it's still a good indication of relative impact:

	STATE	CLUSTER	PEPSTRFL	HVP2	LSC4	RAMNTALL	NGIFTALL	MAXRAMNT	LASTGIFT	AVGGIFT	MDMAUD_A	DOMAIN_SOCIALCLS
STATE	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.002	0.001	0.000	0.000
CLUSTER	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.005	0.000
PEPSTRFL	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.001
HVP2	0.000	0.000	0.000	0.000	0.000	0.003	0.002	0.004	0.000	0.001	0.000	0.000
LSC4	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000	0.000
RAMNTALL	0.000	0.000	0.000	0.003	0.000	0.000	0.006	0.003	0.000	0.000	0.000	0.000
NGIFTALL	0.000	0.000	0.002	0.002	0.000	0.006	0.000	0.001	0.000	0.000	0.000	0.001
MAXRAMNT	0.001	0.001	0.000	0.004	0.002	0.003	0.001	0.000	0.001	0.003	0.001	0.001
LASTGIFT	0.002	0.001	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.002	0.001	0.000
AVGGIFT	0.001	0.001	0.000	0.001	0.000	0.000	0.000	0.003	0.002	0.000	0.001	0.000
MDMAUD_A	0.000	0.005	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.000	0.000
DOMAIN_SOCIALCLS	0.000	0.000	0.001	0.000	0.000	0.000	0.001	0.001	0.000	0.000	0.000	0.000

Figure 10.10: SHAP interactions heatmap

One way in which we can understand feature interactions one by one is with SHAP's `dependence_plot`. For instance, we can take our top feature, `MAXRAMNT`, and plot it with color-coded interactions with features such as `RAMNTALL`, `LSC4`, `HVP2`, and `AVGGIFT`. But first, we will need to compute `shap_values`. There are a couple of problems though that need to be addressed, which we mentioned earlier. They have to do with the following:

- **The prevalence of outliers**: We can cut them out of the plot by limiting the *x*- and *y*-axes using percentiles for the feature and SHAP values, respectively, with `plt.xlim` and `plt.ylim`. This essentially zooms in on cases that lie between the 1st and 99th percentiles.
- **Lopsided distribution of dollar amount features**: It is common in any feature involving money for it to be right-skewed. There are many ways to simplify it, such as using percentiles to bin the feature, but a quick way to make it easier to appreciate is by using a logarithmic scale. In `matplotlib`, you can do this with `plt.xscale('log')` without any need to transform the feature.

The following code accounts for the two issues. You can try commenting out `xlim`, `ylim`, or `xscale` to see the big difference they individually make in understanding `dependence_plot`:

```
shap_rf_values = shap_rf_explainer.shap_values(
    X_test.iloc[sample_test_idx] [llarsic_cols]
)
maxramt_shap = shap_rf_values[:,llarsic_cols.index("MAXRAMNT")]
shap.dependence_plot(
    "MAXRAMNT",
    shap_rf_values,
    X_test.iloc[sample_test_idx][llarsic_cols],
    interaction_index="AVGGIFT",
    show=False, alpha=0.1
)
plt.xlim(xmin=np.percentile(X_test.MAXRAMNT, 1),\
        xmax=np.percentile(X_test.MAXRAMNT, 99))
plt.ylim(ymin=np.percentile(maxramt_shap, 1),\
        ymax=np.percentile(maxramt_shap, 99))
plt.xscale('log')
```

The preceding code generates what is shown in *Figure 10.11*. It shows how there's a tipping point somewhere between 10 and 100 for `MAXRAMNT` where the mean impact on the model output starts to creep out, and these correlate with a higher `AVGGIFT` value:
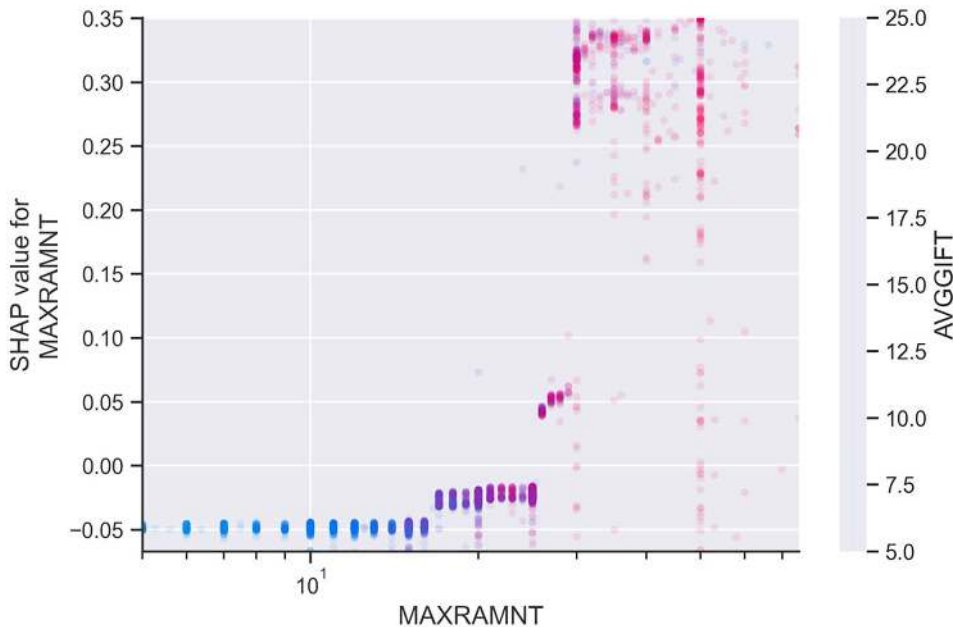
Figure 10.11: SHAP interaction plot between MAXRAMNT and AVGGIFT

A lesson you could take from *Figure 10.11* is that a cluster is formed by certain values of these features and possibly a few others that increase the likelihood of a donation. From a feature engineering standpoint, we could take unsupervised methods to create special cluster features solely based on the few features you have identified as related. Or we could take a more manual route, comparing different plots to understand how to best identify clusters. We could derive binary features from this process or even a ratio between features that more clearly depict interactions or cluster belonging.

The idea here is not to reinvent the wheel trying to do what the model already does so well but to, first and foremost, aim for a more straightforward model interpretation. Hopefully, that will even have a positive impact on predictive performance by tidying up the features, because if you understand them better, maybe the model does too! It's like smoothing a grainy image; it might confuse you less and the model too (see *Chapter 13, Adversarial Robustness*, for more on that)! But understanding the data better through the model has other positive side effects.

In fact, the lessons don't stop with feature engineering or modeling but can be directly applied to promotions. What if tipping points identified could be used to encourage donations? Perhaps get a free mug if you donate over $X? Or set up a recurring donation of $X and be on the exclusive list of "silver" patrons?

We will end this topic on that curious note, but hopefully, this inspires you to appreciate how we can apply lessons from model interpretation to feature selection, engineering, and much more.

Mission accomplished

To approach this mission, you have reduced overfitting using primarily the toolset of feature selection. The non-profit is pleased with a profit lift of roughly 30%, costing a total of $35,601, which is $30,000 less than it would cost to send everyone in the test dataset the mailer. However, they still want assurance that they can safely employ this model without worries that they'll experience losses.

In this chapter, we've examined how overfitting can cause the profitability curves not to align. Misalignment is critical because it could mean that choosing a threshold based on training data would not be reliable on out-of-sample data. So, you use `compare_df_plots` to compare profitability between the test and train sets as you've done before, but this time, for the chosen model (`rf_5_e-llarsic`):

```
profits_test = reg_mdls['rf_5_e-llarsic']['profits_test']
profits_train = reg_mdls['rf_5_e-llarsic']['profits_train']
mldatasets.compare_df_plots(
    profits_test[['costs', 'profit', 'roi']],
    profits_train[['costs', 'profit', 'roi']],
    'Test',
    'Train',
    x_label='Threshold',
    y_formatter=y_formatter,
    plot_args={'secondary_y':'roi'}
)
```

The preceding code generates what is shown in *Figure 10.12*. You can show this to the non-profit to prove that there's a sweet spot at $0.68 that is the second highest profit attainable in **Test**. It is also within reach of their budget and achieves an ROI of 41%. More importantly, these numbers are not far from what they are for **Train**. Another thing that is great to see is that the profit curve slowly slides down for both **Train** and **Test** instead of dramatically falling off a cliff. The non-profit can be assured that the operation would still be profitable if they choose to increase the threshold. After all, they want to target donors from the entire mailing list, and for that to be financially feasible, they have to be more exclusive. Say they are using a threshold of $0.77 on the entire mailing list, the campaign would cost about $46,000 but return over $24,000 in profit:
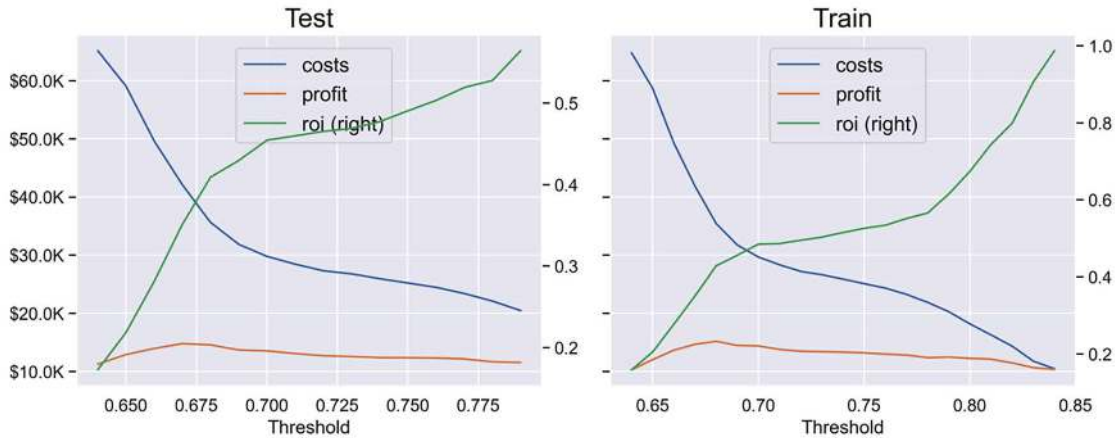
Figure 10.12: Comparison between profit, costs, and ROI for the test and train datasets for the model with LASSO LARS via AIC features across different thresholds

Congratulations! You have accomplished this mission!

But there's one crucial detail that we'd be careless if we didn't bring up.

Although we trained this model with the next campaign in mind, the model will likely be used in future direct marketing campaigns without retraining. This model reusing presents a problem. There's a concept called **data drift**, also known as **feature drift**, which is that, over time, what the model learned about the features concerning the target variable no longer holds true. Another concept, **concept drift**, is about how the definition of the target feature changes over time. For instance, what constitutes a profitable donor can change. Both drifts can happen simultaneously, and with problems involving human behavior, this is to be expected. Behavior is shaped by cultures, habits, attitudes, technologies, and fashions, which are always evolving. You can caution the non-profit that you can only assure them that the model will be reliable for the next campaign, but they can't afford to hire you for model retraining every single time!

You can propose to the client creating a script that monitors drift directly on their mailing list database. If it finds significant changes in the features used by the model, it will alert both them and you. You could, at this point, trigger automatic retraining of the model. However, if the drift is due to data corruption, you won't have an opportunity to address the problem. And even if automatic retraining is done, it can't be deployed if performance metrics don't meet predetermined standards. Either way, you should keep a close eye on predictive performance to be able to guarantee reliability. Reliability is an essential theme in model interpretability because it relates heavily to accountability. We won't cover drift detection in this book, but future chapters discuss data augmentation (*Chapter 11*, *Bias Mitigation and Causal Inference Methods*) and adversarial robustness (*Chapter 13*, *Adversarial Robustness*), which pertain to reliability.

Summary

In this chapter, we have learned about how irrelevant features impact model outcomes and how feature selection provides a toolset to solve this problem. We then explored many different methods in this toolset, from the most basic filter methods to the most advanced ones. Lastly, we broached the subject of feature engineering for interpretability. Feature engineering can make for a more interpretable model that will perform better. We will cover this topic in more detail in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*.

In the next chapter, we will discuss methods for bias mitigation and causal inference.

Dataset sources

- Ling, C., and Li, C., 1998, *Data Mining for Direct Marketing: Problems and Solutions*. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98). AAAI Press, 73–79: `https://dl.acm.org/doi/10.5555/3000292.3000304`
- UCI Machine Learning Repository, 1998, KDD Cup 1998 Data Data Set: `https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1998+Data`

Further reading

- Ross, B.C., 2014*, Mutual Information between Discrete and Continuous Data Sets*. PLoS ONE, 9: `https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0087357`
- Geurts, P., Ernst, D., and Wehenkel, L., 2006, *Extremely randomized trees*. Machine Learning, 63(1), 3-42: `https://link.springer.com/article/10.1007/s10994-006-6226-1`
- Abid, A., Balin, M.F., and Zou, J., 2019, *Concrete Autoencoders for Differentiable Feature Selection and Reconstruction*. ICML: `https://arxiv.org/abs/1901.09346`
- Tan, F., Fu, X., Zhang, Y., and Bourgeois, A.G., 2008, *A genetic algorithm-based method for feature subset selection*. Soft Computing, 12, 111-120: `https://link.springer.com/article/10.1007/s00500-007-0193-8`
- Calzolari, M., 2020, October 12, manuel-calzolari/sklearn-genetic: sklearn-genetic 0.3.0 (Version 0.3.0). Zenodo: `http://doi.org/10.5281/zenodo.4081754`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

11

Bias Mitigation and Causal Inference Methods

In *Chapter 6*, *Anchors and Counterfactual Explanations*, we examined fairness and its connection to decision-making but limited to post hoc model interpretation methods. In *Chapter 10*, *Feature Selection and Engineering for Interpretability*, we broached the topic of cost-sensitivity, which often relates to balance or fairness. In this chapter, we will engage with methods that will balance data and tune models for fairness.

With a credit card default dataset, we will learn how to leverage target visualizers such as class balance to detect undesired bias, then how to reduce it via preprocessing methods such as reweighting and disparate impact remover for in-processing and equalized odds for post-processing. Extending from the topics of *Chapter 6*, *Anchors and Counterfactual Explanations*, and *Chapter 10*, *Feature Selection and Engineering for Interpretability*, we will also study how policy decisions can have unexpected, counterintuitive, or detrimental effects. A decision, in the context of hypothesis testing, is called a **treatment**. For many decision-making scenarios, it is critical to estimate their effect and make sure this estimate is reliable.

Therefore, we will hypothesize treatments for reducing credit card default for the most vulnerable populations and leverage causal modeling to determine its **Average Treatment Effects** (**ATE**) and **Conditional Average Treatment Effects** (**CATE**). Finally, we will test causal assumptions and the robustness of estimates using a variety of methods.

These are the main topics we are going to cover:

- Detecting bias
- Mitigating bias
- Creating a causal model
- Understanding heterogeneous treatment effects
- Testing estimate robustness

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `lightgbm`, `xgboost`, `matplotlib`, `seaborn`, `xai`, `aif360`, `econml`, and `dowhy` libraries. Instructions on how to install all these libraries are in the preface.

> The code for this chapter is located here:
>
> `https://packt.link/xe6ie`

The mission

Over 2.8 billion credit cards are circulating worldwide, and we collectively spend over $25 trillion (US) on them every year (`https://www.ft.com/content/ad826e32-2ee8-11e9-ba00-0251022932c8`). This is an astronomical amount, no doubt, but the credit card industry's size is best measured not by what is spent, but by what is owed. Card issuers such as banks make the bulk of their money from interest. So, the over $60 trillion owed by consumers (2022), of which credit card debt is a sizable portion, provides a steady income to lenders in the form of interest. It could be argued this is good for business, but it also poses ample risk because if a borrower defaults before the principal plus operation costs have been repaid, the lender could lose money, especially once they've exhausted legal avenues to collect the debt.

When there's a credit bubble, this problem is compounded because an unhealthy level of debt can compromise lenders' finances and take their stakeholders down with them when the bubble bursts. Such was the case with the 2008 housing bubble, also known as the subprime mortgage crisis. These bubbles often begin with speculation on growth and seeking unqualified demand to fuel that growth. In the case of the mortgage crisis, the banks offered mortgages to people with no proven capacity to repay. They also, sadly, targeted minorities who had their entire net worth wiped out once the bubble burst. Financial crises and depressions, and every calamity in between, tend to affect those that are most vulnerable at much higher rates.

Credit cards have also been involved in catastrophic bubbles, notably in South Korea in 2003 (`https://www.bis.org/repofficepubl/arpresearch_fs_200806.10.pdf`) and Taiwan in 2006. This chapter will examine data from 2005, leading to the Taiwanese credit card crisis. By 2006, delinquent credit card debt reached $268 billion owed by over 700,000 people. Just over 3% of the Taiwanese population could not pay even the credit card's minimum balance and colloquially were known as **credit card slaves**. Significant societal ramifications ensued, such as a sharp increase in homelessness, drug trafficking/abuse, and even suicide. In the aftermath of the 1997 Asian financial crisis, suicide steadily increased around the region. A 23% jump between 2005 and 2006 pushed Taiwan's suicide rate to the world's second highest.

If we trace back the crisis to its root causes, it was about new card-issuing banks having exhausted a saturated real-estate market, slashing requirements to obtain credit cards, which at the time were poorly regulated by authorities.

It hit younger people the most because they typically have less income and experience in managing money. In 2005, the Taiwanese Financial Supervisory Commission issued new regulations to raise credit card applicants' requirements, preventing new credit card slaves. However, more policies would be needed to attend to the debt and the debtors already in the system.

Authorities started discussing the creation of **asset management corporations** (**AMCs**) to take bad debts from the balance sheet of banks. They also wanted to pass a *debtors' repayment regulation* that would provide a framework to negotiate a reasonable repayment plan. Neither of these policies were codified into law untill 2006.

Hypothetically, let's say it's August 2005, and you have come from the future armed with novel machine learning and causal inference methods! A Taiwanese bank wants to create a classification model to predict customers that will default on their loans. They have provided you with a dataset of 30,000 of their credit card customers. Regulators are still drafting the laws, so there's an opportunity to propose policies that benefit both the bank and the debtors. When the laws have passed, using the classification model, they can then anticipate which debts they should sell to the AMCs and, with the causal model, estimate which policies would benefit other customers and the bank, but they want to do this fairly and robustly—this is your mission!

The approach

The bank has stressed to you how important it is that there's fairness embedded in your methods because the regulators and the public at large want assurance that banks will not cause any more harm. Their reputation depends on it too, because in recent months, the media has been relentless in blaming them for dishonest and predatory lending practices, causing distrust in consumers. For this reason, they want to use state-of-the-art robustness testing to demonstrate that the prescribed policies will alleviate the problem. Your proposed approach includes the following points:

- Younger lenders have been reported to be more prone to defaulting on repayment, so you expect to find age bias, but you will also *look for bias* with other protected groups such as gender.
- Once you have detected bias, you can *mitigate bias* with preprocessing, in-processing, and post-processing algorithms using the **AI Fairness 360** (**AIF360**) library. In this process, you will train different models with each algorithm, assess their fairness, and choose the fairest model.
- To be able to understand the impact of policies, the bank has conducted an experiment on a small portion of customers. With the experimental results, you can fit a *causal model* through the `dowhy` library, which will identify the *causal effect*. These effects are broken down further by the causal model to reveal the heterogeneous treatment effects.
- Then, you can *assess the heterogeneous treatment effects* to understand them and decide which treatment is the most effective.
- Lastly, to *ensure that your conclusions are robust*, you will refute the results with several methods to see if the effects hold.

Let's dig in!

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python/blob/master/Chapter11/CreditCardDefaults.ipynb`.

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `sklearn` (scikit-learn), `xgboost`, `aif360`, and `lightgbm` to split the data and fit the models
- `matplotlib`, `seaborn`, and `xai` to visualize the interpretations
- `econml` and `dowhy` for causal inference

You should load all of them first, as follows:

```python
import math
import os
import mldatasets
import pandas as pd
import numpy as np
from tqdm.notebook import tqdm
from sklearn import model_selection, tree
import lightgbm as lgb
import xgboost as xgb
from aif360.datasets import BinaryLabelDataset
from aif360.metrics import BinaryLabelDatasetMetric,\
                           ClassificationMetric
from aif360.algorithms.preprocessing import Reweighing,\
                                            DisparateImpactRemover
from aif360.algorithms.inprocessing import ExponentiatedGradientReduction,
GerryFairClassifier
from aif360.algorithms.postprocessing.\
               calibrated_eq_odds_postprocessing \
                    import CalibratedEqOddsPostprocessing
from aif360.algorithms.postprocessing.eq_odds_postprocessing\
                    import EqOddsPostprocessing
from econml.dr import LinearDRLearner
import dowhy
from dowhy import CausalModel
import xai
from networkx.drawing.nx_pydot import to_pydot
from IPython.display import Image, display
import matplotlib.pyplot as plt
import seaborn as sns
```

Understanding and preparing the data

We load the data like this into a DataFrame called `ccdefault_all_df`:

```
ccdefault_all_df = mldatasets.load("cc-default", prepare=True)
```

There should be 30,000 records and 31 columns. We can verify this is the case with `info()`, like this:

```
ccdefault_all_df.info()
```

The preceding code outputs the following:

```
Int64Index: 30000 entries, 1 to 30000
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   CC_LIMIT_CAT     30000 non-null  int8
 1   EDUCATION        30000 non-null  int8
 2   MARITAL_STATUS   30000 non-null  int8
 3   GENDER           30000 non-null  int8
 4   AGE_GROUP        30000 non-null  int8
 5   pay_status_1     30000 non-null  int8
 6   pay_status_2     30000 non-null  int8
 7   pay_status_3     30000 non-null  int8
 8   pay_status_4     30000 non-null  int8
 9   pay_status_5     30000 non-null  int8
 10  pay_status_6     30000 non-null  int8
 11  paid_pct_1       30000 non-null  float64
 12  paid_pct_2       30000 non-null  float64
 13  paid_pct_3       30000 non-null  float64
 14  paid_pct_4       30000 non-null  float64
 15  paid_pct_5       30000 non-null  float64
 16  paid_pct_6       30000 non-null  float64
 17  bill1_over_limit 30000 non-null  float64
 18  IS_DEFAULT       30000 non-null  int8
 19  _AGE             30000 non-null  int16
 20  _spend           30000 non-null  int32
 21  _tpm             30000 non-null  int16
 22  _ppm             30000 non-null  int16
 23  _RETAIL          30000 non-null  int8
 24  _URBAN           30000 non-null  int8
 25  _RURAL           30000 non-null  int8
 26  _PREMIUM         30000 non-null  int8
 27  _TREATMENT       30000 non-null  int8
 28  _LTV             30000 non-null  float64
 29  _CC_LIMIT        30000 non-null  int32
 30  _risk_score      30000 non-null  float64
```

The output checks out. All features are numerical, with no missing values because we used `prepare=True`, which ensures that all null values are imputed. Categorical features are all `int8` because they have already been encoded.

The data dictionary

There are 30 features, but we won't use them together because 18 of them are for the bias mitigation exercise, and the remaining 12 that start with an underscore (_) are for the causal inference exercise. Soon, we will split the data into the corresponding datasets for each exercise. It's important to note that lowercase features have to do with each client's transactional history, whereas client account or target features are uppercase.

We will use the following features in the *bias mitigation exercise*:

- `CC_LIMIT_CAT`: ordinal; the credit card limit (`_CC_LIMIT`) separated into eight approximately equally distributed quartiles
- `EDUCATION`: ordinal; the customer's educational attainment level (0: Other, 1: High School, 2: Undergraduate, 3: Graduate)
- `MARITAL_STATUS`: nominal; the customer's marital status (0: Other, 1: Single, 2: Married)
- `GENDER`: nominal; the gender of the customer (1: Male, 2: Female)
- `AGE_GROUP`: binary; denoting if the customer belongs to a privileged age group (1: privileged (26-47 years old), 0: underprivileged (every other age))
- `pay_status_1` `pay_status_6`: ordinal; the repayment status for the previous six periods from April, `pay_status_6`, to August 2005, `pay_status_1` (-1: payment on time, 1: payment is 1 month delayed, 2: payment is 2 months delayed 8: 8 months delayed, 9: 9 months and above)
- `paid_pct_1` `paid_pct_6`: continuous; the percentage of the bill due each month from April, `paid_pct_6`, to August 2005, `paid_pct_1`, that was paid
- `bill1_over_limit`: continuous; the last bill's ratio in August 2005 over the corresponding credit limit
- `IS_DEFAULT`: binary; target variable; whether the customer defaulted

These are the features we will use only in the *causal inference exercise*:

- `_AGE`: continuous; the age in years of the customer.
- `_spend`: continuous; how much was spent by each customer in **New Taiwan Dollar** (**NT$**).
- `_tpm`: continuous; median transactions per month made by the customer with the credit card over the previous 6 months.
- `_ppm`: continuous; median purchases per month made by the customer with the credit card over the previous 6 months.
- `_RETAIL`: binary; if the customer is a retail customer, instead of a customer obtained through their employer.
- `_URBAN`: binary; if the customer is an urban customer.
- `_RURAL`: binary; if the customer is a rural customer.

- _PREMIUM: binary; if the customer is "premium." Premium customers get cashback offers and other spending incentives.

- _TREATMENT: nominal; the intervention or policy prescribed to each customer (-1: Not part of the experiment, 0: Control group, 1: Lower Credit Limit, 2: Payment Plan, 3: Payment Plan and Credit Limit).

- _LTV: continuous; the outcome of the intervention, which is the lifetime value estimated in *NT$* given the credit payment behavior over the previous 6 months.

- _CC_LIMIT: continuous; the original credit card limit in *NT$* that the customer had before the treatment. Bankers expect the outcome of the treatment to be greatly impacted by this feature.

- _risk_score: continuous; the risk score that the bank computed 6 months prior for each customer based on credit card bills' ratio over their credit card limit. It's like `bill1_over_limit` except it's a weighted average of 6 months of payment history, and it was produced 5 months before choosing the treatment.

We will explain the causal inference features a bit more and their purpose in the following sections. Meanwhile, let's break down the _TREATMENT feature by its values with `value_counts()` to understand how we will split this dataset, as follows:

```
ccdefault_all_df._TREATMENT.value_counts()
```

The preceding code outputs the following:

```
-1    28904
 3      274
 2      274
 1      274
 0      274
```

Most of the observations are treatment -1, so they are not part of the causal inference. The remainder was split evenly between the three treatments (1-3) and the control group (0). Naturally, we will use these four groups for the causal inference exercise. However, since the control group wasn't prescribed treatment, we can use it in our bias mitigation exercise along with the -1 treatments. We have to be careful to exclude customers whose behaviors were manipulated in the bias mitigation exercise. The whole point is to predict which customers are most likely to default under "business as usual" circumstances while attempting to reduce bias.

Data preparation

Our single data preparation step, for now, is to split the datasets, which can be easily done by subsetting the pandas DataFrames using the _TREATMENT column. We will create one DataFrame for each exercise with this subsetting: bias mitigation (`ccdefault_bias_df`) and causal inference (`ccdefault_causal_df`). These can be seen in the following code snippet:

```
ccdefault_bias_df = ccdefault_all_df[
    ccdefault_all_df._TREATMENT < 1
]
```

```
ccdefault_causal_df =ccdefault_all_df[
    ccdefault_all_df._TREATMENT >= 0
]
```

We will do a few other data preparation steps in the in-depth sections but, for now, we are good to go to get started!

Detecting bias

There are many sources of bias in machine learning. As outlined in *Chapter 1, Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, there are ample sources of bias. Those rooted in the *truths* that the data represents, such as systemic and structural ones, lead to prejudice bias in the data. There are also biases rooted in the data, such as sample, exclusion, association, and measurement biases. Lastly, there are biases in the insights we derive from data or models we have to be careful with, such as conservatism bias, salience bias, and fundamental attribution error.

For this example, to properly disentangle so many bias levels, we ought to connect our data to census data for Taiwan in 2005 and historical lending data split by demographics. Then, using these external datasets, control for credit card contract conditions, as well as gender, income, and other demographic data to ascertain if young people, in particular, were targeted for high-interest credit cards they shouldn't have qualified for. We would also need to trace the dataset to the authors and consult with them and the domain experts to examine the dataset for bias-related data quality issues. Ideally, these steps would be necessary to validate the hypothesis, but that would be a monumental task requiring several chapters' worth of explanation.

Therefore, in the spirit of expediency, we take the premise of this chapter at face value. That is, due to predatory lending practices, certain age groups are more vulnerable to credit card default, not through any fault of their own. We will also take at face value the quality of the dataset. With these caveats in place, it means that if we find disparities between age groups in the data or any model derived from this data, it can be attributed solely to predatory lending practices.

There are also two types of fairness, outlined here:

- **Procedural fairness:** This is about fair or equal treatment. It's hard to define this term legally because it depends so much on the context.
- **Outcome fairness:** This is solely about measuring fair outcomes.

These two concepts aren't mutually exclusive since the procedure may be fair but the outcome unfair, or vice versa. In this example, the unfair *procedure* was the offering of high-interest credit cards to unqualified customers. Nevertheless, we are going to focus on outcome fairness in this chapter.

When we discuss bias in machine learning, it will impact *protected* groups, and within these groups, there will be *privileged* and *underprivileged* groups. The latter is a group that is adversely impacted by bias. There are also many ways in which bias is manifested, and thus addressed, as follows:

- **Representation:** There can be a lack of representation or an overrepresentation of the underprivileged group. The model will learn either too little or too much about this group, compared to others.

- **Distribution**: Differences in the distribution of features between groups can lead the model to make biased associations that can impact model outcomes either directly or indirectly.
- **Probability**: For classification problems, class balance discrepancies between groups such as those discussed in *Chapter 6*, *Anchors and Counterfactual Explanations*, can lead to the model learning that one group has a higher probability of being part of one class or another. These can be easily observed through confusion matrices or by comparing their classification metrics, such as false positive or false negative rates.
- **Hybrid**: A combination of any of the preceding manifestations.

Strategies for any bias manifestation are discussed in the *Mitigating bias* section, but the kind we address in the chapter pertains to disparities with probability for our main protected attribute (`_AGE`). We will observe this through these means:

- **Visualizing dataset bias**: Observing disparities in the data for the protected feature through visualizations.
- **Quantifying dataset bias**: Measuring bias using fairness metrics.
- **Quantifying model bias**: We will train a classification model and use other fairness metrics designed for models.

Model bias can be visualized, as we have done already in *Chapter 6*, *Anchors and Counterfactual Explanations*, or as we will do in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*. We will quickly explore some other visualizations later in this chapter, in a subsection called *Tying it all together!* Without further ado, let's move on to the practical portion of this section.

Visualizing dataset bias

The data itself tells the story of how probable it is that one group belongs to a positive class versus another. If it's a categorical feature, these probabilities can be obtained by dividing the `value_counts()` function for the positive class over all classes. For instance, for gender, we could do this:

```
ccdefault_bias_df[
    ccdefault_bias_df.IS_DEFAULT==1
].GENDER.value_counts()/ccdefault_bias_df.GENDER.value_counts()
```

The preceding snippet produces the following output, which shows that males have, on average, a higher probability of defaulting on their credit card:

```
2    0.206529
1    0.241633
```

The code for doing this for a continuous feature is a bit more complicated. It is recommended that you use `pandas' qcut` to divide the feature into quartiles first and then use the same approach used for categorical features. Fortunately, the `plot_prob_progression` function does this for you and plots the progression of probabilities for each quartile. The first attribute is a `pandas` series, an array or list with the protected feature (`_AGE`), and the second is the same but for the target feature (`IS_DEFAULT`). We then choose the number of intervals (`x_intervals`) that we are setting as quartiles (`use_quantiles=True`).

The rest of the attributes are aesthetic, such as the label, title, and adding a `mean_line`. The code can be seen in the following snippet:

```
mldatasets.plot_prob_progression(
    ccdefault_bias_df._AGE,
    ccdefault_bias_df.IS_DEFAULT,
    x_intervals=8,
    use_quantiles=True,
    xlabel='Age',
    mean_line=True,
    title='Probability of Default by Age'
)
```

The preceding code produced the following output, which depicts how the youngest (21-25) and oldest (47-79) are most likely to default. All other groups represent just over one standard deviation from the mean:
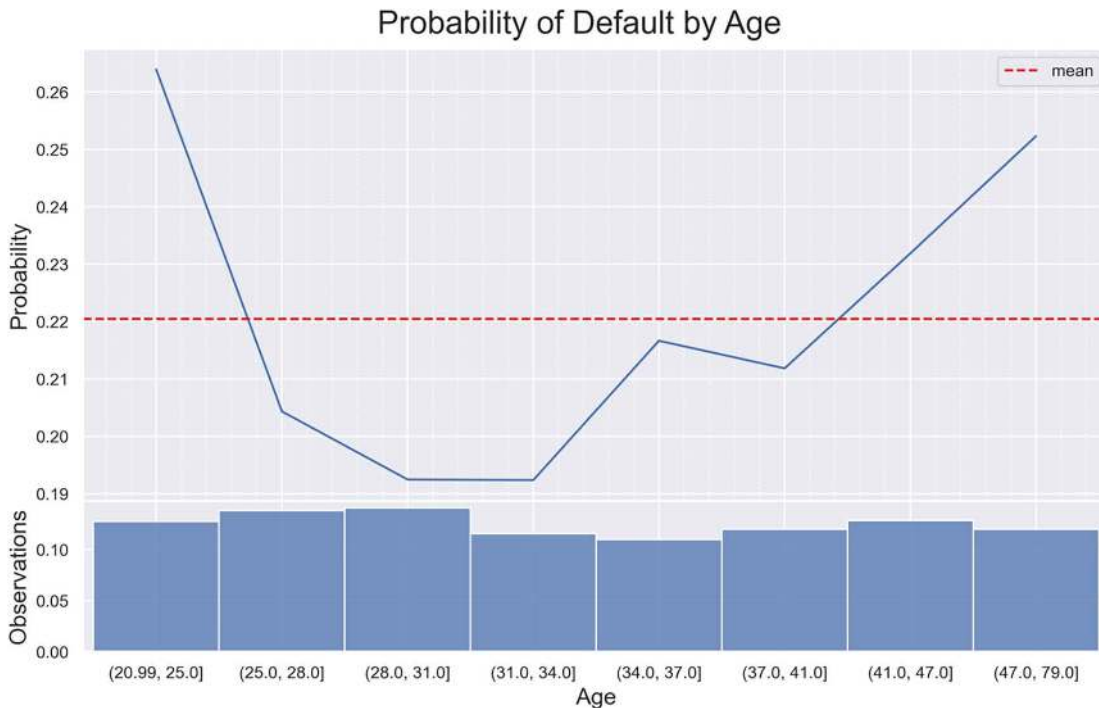
Figure 11.1: Probability of CC default by _AGE

We can call the youngest and oldest quartiles the underprivileged group and all others the privileged group. In order to detect and mitigate unfairness, it is best to code them as a binary feature—and we have done just that with `AGE_GROUP`. We can leverage `plot_prob_progression` again, but this time with `AGE_GROUP` instead of `AGE`, and we will `replace` the numbers with labels we can interpret more easily. The code can be seen in the following snippet:

```
mldatasets.plot_prob_progression(
    ccdefault_bias_df.AGE_GROUP.replace({0:'21-25,48+',1:'26-47'}),
    ccdefault_bias_df.IS_DEFAULT,
    xlabel='Age Group',
    title='Probability of Default by Age Group',
    mean_line=True
)
```

The preceding snippet produced the following output, in which the disparities between both groups are pretty evident:
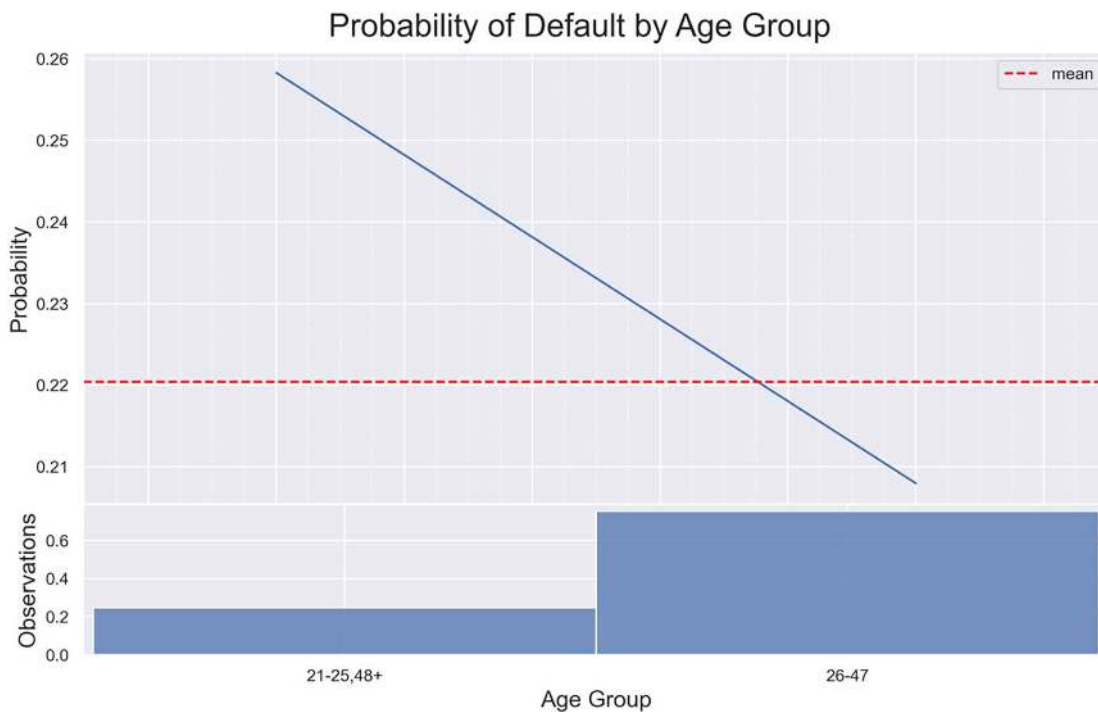
Figure 11.2: Probability of CC default by AGE_GROUP

Next, let's bring GENDER back into the picture. We can employ plot_prob_contour_map, which is like plot_prob_progression but in two dimensions, color-coding the probabilities instead of drawing a line. So, the first two attributes are the features we want on the *x*-axis (GENDER) and *y*-axis (AGE_GROUP), and the third is the target (IS_DEFAULT). Since both our features are binary, it is best to use plot_type='grid' as opposed to contour. The code can be seen in the following snippet:

```
mldatasets.plot_prob_contour_map(
    ccdefault_bias_df.GENDER.replace({1:'Male',2:'Female'}),
    ccdefault_bias_df.AGE_GROUP.replace({0:'21-25,48+',1:'26-47'}),
    ccdefault_bias_df.IS_DEFAULT,
    xlabel='Gender',
```

```
        ylabel='Age Group',
        annotate=True,
        plot_type='grid',
        title='Probability of Default by Gender/Age Group'
    )
```

The preceding snippet generates the following output. It is immediately evident how the most privileged group is 26- 47-year-old females, followed by their male counterparts at about 3-4% apart. The same happens with the underprivileged age group:
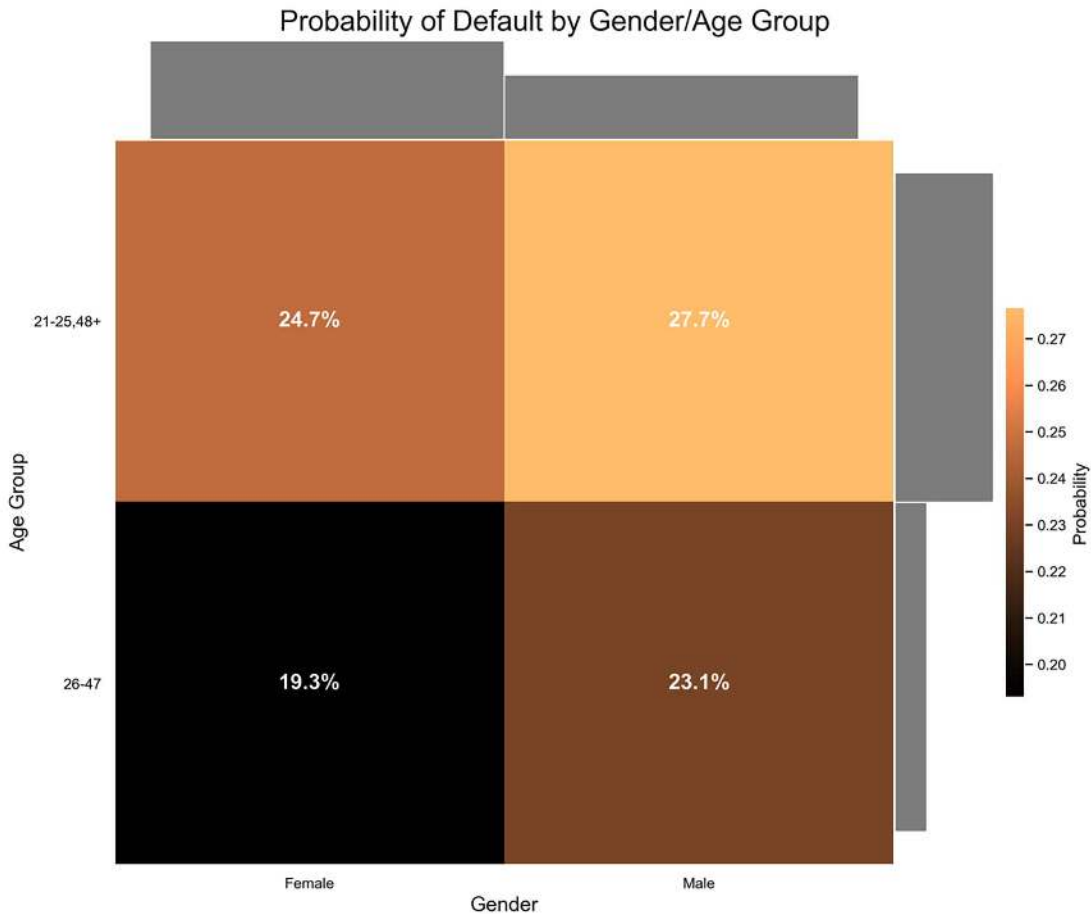
Figure 11.3: Probability grid of CC default by GENDER and AGE_GROUP

The gender difference is an interesting observation, and we could present a number of hypotheses as to why females default less. Are they just simply better at managing debt? Does it have to do with their marital status or education? We won't dig deeper into these questions. Given that we only know of age-based discrimination, we will only use AGE_GROUP in privilege groups but keep GENDER a protected attribute, which will be factored in some fairness metrics we will monitor. Speaking of metrics, we will quantify dataset bias next.

Quantifying dataset bias

There are three categories of fairness metrics, outlined here:

- **Individual fairness:** How close individual observations are to their peers in the data. Distance metrics such as *Euclidean* and *Manhattan distance* can serve this purpose.
- **Group fairness:** How labels or outcomes between groups are, on average, distant from each other. This can be measured either in the data or for a model.
- **Both:** A few metrics measure entropy or variance by factoring inequality both in-group and between groups, such as the *Theil index* and the *coefficient of variation*.

We will focus exclusively on group fairness metrics in this chapter.

Before we compute fairness metrics, there are a few pending data preparation steps. Let's make sure the dataset we will use for the bias mitigation exercise (`ccdefault_bias_df`) only has the pertinent columns, which are those that don't begin with an underscore (`"_"`). On the other hand, the causal inference exercise will include only the underscored columns plus `AGE_GROUP` and `IS_DEFAULT`. The code can be seen in the following snippet:

```
cols_bias_l = ccdefault_all_df.columns[
    ~ccdefault_all_df.columns.str.startswith('_')
].tolist()
cols_causal_l = ['AGE_GROUP','IS_DEFAULT'] +\
    ccdefault_all_df.columns[
        ccdefault_all_df.columns.str.startswith('_')
    ].tolist()
ccdefault_bias_df = ccdefault_bias_df[cols_bias_l]
ccdefault_causal_df = ccdefault_causal_df[cols_causal_l]
```

Also, it's more important to quantify dataset bias on the training data because that is the data the model will learn from, so let's go ahead and split the data into train and test X and y pairs. We do this after we have, of course, initialized the random seed to aim for some reproducibility. The code can be seen in the following snippet:

```
rand = 9
os.environ['PYTHONHASHSEED']=str(rand)
np.random.seed(rand)
y = ccdefault_bias_df['IS_DEFAULT']
X = ccdefault_bias_df.drop(['IS_DEFAULT'], axis=1).copy()
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size=0.25, random_state=rand
)
```

Even though we will use the `pandas` data we just split for training and performance evaluation, the library we will use for this exercise, called AIF360, abstracts datasets into base classes. These classes include the data converted to a `numpy` array and store attributes related to fairness.

For regression, AIF360 has `RegressionDataset`, but for this binary classification example, we will use `BinaryLabelDataset`. You can initialize it with the `pandas` DataFrame with both features and labels (`X_train.join(y_train)`). Then, you specify the name of the label (`label_names`) and protected attributes (`protected_attribute_names`), and it is recommended that you enter a value for `favorable_label` and `unfavorable_label`, which tells AIF360 which label values are preferred so that it factors them into how it assesses fairness. As confusing as it may seem, positive and, in contrast, negative in binary classification only pertain to what we are trying to predict—the positive class—and not whether it is a favorable outcome. The code can be seen in the following snippet:

```
train_ds = BinaryLabelDataset(
    df=X_train.join(y_train),
    label_names=['IS_DEFAULT'],
    protected_attribute_names=['AGE_GROUP', 'GENDER'],
    favorable_label=0,
    unfavorable_label=1
)
test_ds = BinaryLabelDataset(
    df=X_test.join(y_test),
    label_names=['IS_DEFAULT'],
    protected_attribute_names=['AGE_GROUP', 'GENDER'],
    favorable_label=0, unfavorable_label=1
)
```

Next, we create arrays for `underprivileged groups` and `privileged_groups`. Those in `AGE_GROUP=1` have a lower probability of default, so they are privileged, and vice versa. Then, with these and the abstracted dataset for training (`train_ds`), we can initialize a metrics class via `BinaryLabelDatasetMetric`. This class has functions for computing several group fairness metrics, judging the data alone. We will output three of them and then explain what they mean. The code can be seen in the following snippet:

```
underprivileged_groups=[{'AGE_GROUP': 0}]
privileged_groups=[{'AGE_GROUP': 1}]
metrics_train_ds = BinaryLabelDatasetMetric(
    train_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
print('Statistical Parity Difference (SPD): %.4f' %
        metrics_train_ds.statistical_parity_difference())
print('Disparate Impact (DI): %.4f' %
        metrics_train_ds.disparate_impact())
print('Smoothed Empirical Differential Fairness (SEDF): %.4f' %\
        metrics_train_ds.smoothed_empirical_differential_fairness())
```

The preceding snippet generates the following output:

```
Statistical Parity Difference (SPD):                -0.0437
Disparate Impact (DI):                               0.9447
Smoothed Empirical Differential Fairness (SEDF):     0.3514
```

Now, let's explain what each metric means, as follows:

- **Statistical Parity Difference** (**SPD**): Also known as the **mean difference**, this is the difference between the mean probability of favorable outcomes between underprivileged and privileged groups. A negative number represents unfairness to the underprivileged group and a positive number is better, yet a number closer to zero represents a fair outcome with no significant difference between the privileged and underprivileged groups. It's computed with the following formula, where *f* is the value for the favorable class, *D* is the group of the customer, and *Y* is whether the customer will default or not:

$$Pr(Y = f | D = \text{underprivileged}) - Pr(Y = f | D = \text{privileged})$$

- **Disparate Impact** (**DI**): DI is exactly like SPD except it's the ratio, not the difference. And, as ratios go, the closer to one the better for the underprivileged group. In other words, one represents a fair outcome between groups with no difference, below one means unfavorable outcomes to the underprivileged group compared to the privileged group, and over one means favorable outcomes to the underprivileged group compared to the privileged group. The formula is shown here:

$$\frac{Pr(Y = f | D = \text{underprivileged})}{Pr(Y = f | D = \text{privileged})}$$

- **Smoothed Empirical Differential Fairness** (**SEDF**): This fairness metric is one of the many newer ones from a paper called *"An Intersectional Definition of Fairness."* Unlike the previous two metrics, it's not restricted to the predetermined privileged and underprivileged groups, but it's extended to include all the categories in the protected attributes—in this case, the four in *Figure 11.3*. The authors of the paper argue that fairness is particularly tricky when you have a crosstab of protected attributes. This occurs because of **Simpson's paradox**, which is that one group can be advantaged or disadvantaged in aggregate but not when subdivided into crosstabs. We won't get into the math, but their method accounts for this possibility while measuring a sensible level of fairness in intersectional scenarios. To interpret it, zero represents absolute fairness, and the farther from zero it is, the less fair it is.

Next, we will quantify group fairness metrics for a model.

Quantifying model bias

Before we compute metrics, we will need to train a model. To that end, we will initialize a LightGBM classifier (`LGBMClassifier`) with optimal hyperparameters (`lgb_params`). These have already been hyperparameter-tuned for us (more details on how to do this in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*).

Please note that these parameters include `scale_pos_weight`, which is for class weighting. Since this is an unbalanced classification task, this is an essential parameter to leverage so that the classifier is cost-sensitive-trained, penalizing one form of misclassification over another. Once the classifier is initialized, it is `fit` and evaluated with `evaluate_class_mdl`, which returns a dictionary with predictive performance metrics that we can store in a model dictionary (`cls_mdls`). The code can be seen in the following snippet:

```python
cls_mdls = {}
lgb_params = {
    'learning_rate': 0.4,
    'reg_alpha': 21,
    'reg_lambda': 1,
    'scale_pos_weight': 1.8
}
lgb_base_mdl = lgb.LGBMClassifier(
    random_seed=rand,
    max_depth=6,
    num_leaves=33,
    **lgb_params
)
lgb_base_mdl.fit(X_train, y_train)
cls_mdls['lgb_0_base'] = mldatasets.evaluate_class_mdl(
    lgb_base_mdl,
    X_train,
    X_test,
    y_train,
    y_test,
    plot_roc=False,
    plot_conf_matrix=True,
    show_summary=True,
    ret_eval_dict=True
)
```

The preceding snippet of code outputs *Figure 11.4*. The `scale_pos_weight` parameter ensures a healthier balance between false positives in the top-right corner and false negatives at the bottom left. As a result, precision and recall aren't too far off from each other. We favor high precision for a problem such as this one because we want to maximize true positives, but, not at the great expense of recall, so a balance between both is critical. While hyperparameter tuning, the `F1` score, and the **Matthews correlation coefficient** (**MCC**) are useful metrics to use to this end. The evaluation of the LightGBM base model is shown here:

```
Accuracy_train:    0.8255        Accuracy_test:    0.8167
Precision_test:    0.5961        Recall_test:      0.4820
ROC-AUC_test:      0.7901        F1_test:          0.5330        MCC_test:  0.424
```

Figure 11.4: Evaluation of the LightGBM base model

Next, let's compute the fairness metrics for the model. To do this, we need to make a "deep" copy (`deepcopy=True`) of the AIF360 dataset, but we change the `labels` and `scores` to be those predicted by our model. The `compute_aif_metrics` function employs the `ClassificationMetric` class of AIF360 to do for the model what `BinaryLabelDatasetMetric` did for the dataset. However, it doesn't engage with the model directly. It computes fairness using the original dataset (`test_ds`) and the modified one with the model's predictions (`test_pred_ds`). The `compute_aif_metrics` function creates a dictionary with several precalculated metrics (`metrics_test_dict`) and the metric class (`metrics_test_cls`), which can be used to obtain metrics one by one. The code can be seen in the following snippet:

```
test_pred_ds = test_ds.copy(deepcopy=True)
test_pred_ds.labels =\
    cls_mdls['lgb_0_base']['preds_test'].reshape(-1,1)
test_pred_ds.scores = \
    cls_mdls['lgb_0_base']['probs_test'].reshape(-1,1)
metrics_test_dict, metrics_test_cls = \
    mldatasets.compute_aif_metrics(
        test_ds,
        test_pred_ds,
```

```
            unprivileged_groups=underprivileged_groups,
            privileged_groups=privileged_groups
    )
cls_mdls['lgb_0_base'].update(metrics_test_dict)
print('Statistical Parity Difference (SPD): %.4f' %
        metrics_test_cls.statistical_parity_difference())
print('Disparate Impact (DI): %.4f' %
        metrics_test_cls.disparate_impact())
print('Average Odds Difference (AOD): %.4f' %
        metrics_test_cls.average_odds_difference())
print('Equal Opportunity Difference (EOD): %.4f' %
        metrics_test_cls.equal_opportunity_difference())
print('Differential Fairness Bias Amplification(DFBA): %.4f' % \
    metrics_test_cls.differential_fairness_bias_amplification())
```

The preceding snippet generates the following output:

```
Statistical Parity Difference (SPD):              -0.0679
Disparate Impact (DI):                             0.9193
Average Odds Difference (AOD):                     -0.0550
Equal Opportunity Difference (EOD):                -0.0265
Differential Fairness Bias Amplification (DFBA):   0.2328
```

Now, putting the metrics we already explained aside, let's explain what the other ones mean, as follows:

- **Average Odds Difference** (**AOD**): The difference between **False-Positive Rates** (**FPR**) averaged with the difference between **False-Negative Rates** (**FNR**) for both privileged and underprivileged groups. Negative means there's a disadvantage for the underprivileged group, and the closer to zero, the better. The formula is shown here:

$$1/2[(FPR_{D=\text{underprivileged}} - FPR_{D=\text{privileged}}) + (TPR_{D=\text{underprivileged}} - TPR_{D=\text{privileged}})]$$

- **Equal Opportunity Difference** (**EOD**): It's only the **True Positive Rate** (**TPR**) differences of AOD, so it's only useful to measure the *opportunity* for TPRs. As with AOD, negative confirms a disadvantage for the underprivileged group, and the closer the value is to zero means there is no significant difference between groups. The formula is shown here:

$$TPR_{D=\text{underprivileged}} - TPR_{D=\text{privileged}}$$

- **Differential Fairness Bias Amplification** (**DFBA**): This metric was defined in the same paper as SEDF, and similarly has zero as the baseline of fairness and is also intersectional. However, it only measures the difference in unfairness in proportion between the model and the data in a phenomenon called bias amplification. In other words, the value represents how much more the model increases unfairness compared to the original data.

If you compare the model's `SPD` and `DI` metrics to that of the data, they are indeed worse. No surprise there, because it's to be expected since model-learned representations tend to amplify bias. You can confirm this with the `DFBA` metrics. As for `AOD` and `EOD`, they tend to be in the same neighborhood as the `SPD` metrics, but ideally, the `EOD` metric is substantially closer to zero than the `AOD` metric because we care more about TPRs in this example.

Next, we will go over methods to mitigate bias in the model.

Mitigating bias

We can mitigate bias at three different levels with methods that operate at these individual levels:

- **Preprocessing:** These are interventions to detect and remove bias from the training **data** before training the model. Methods that leverage pre-processing have the advantage that they tackle bias at the source. On the other hand, any undetected bias could still be amplified by the model.

- **In-processing:** These methods mitigate bias during the **model training** and are, therefore, highly dependent on the model and tend to not be model-agnostic like the pre-processing and post-processing methods. They also require hyperparameter tuning to calibrate fairness metrics.

- **Post-processing:** These methods mitigate bias during **model inference**. In *Chapter 6*, *Anchors and Counterfactual Explanations*, we touched on the subject of using the What-If tool to choose the right thresholds (see *Figure 6.13* in that chapter), and we manually adjusted them to achieve parity with false positives. Just as we did then, post-processing methods aim to detect and correct fairness directly in the outcomes, but what adjustments to make will depend on which metrics matter most to your problem. Post-processing methods have the advantage that they can tackle outcome unfairness where it can have the greatest impact, but since it's disconnected from the rest of the model development, it can distort things.

Please note that bias mitigation methods can hurt predictive performance, so there's often a trade-off. There can be opposing goals, especially in cases where the data is reflective of a biased truth. We can choose to aim for a better truth instead: a righteous one—*the one we want, not the one we have*.

This section will explain several methods for each level but will only implement and evaluate two for each. Also, we won't do it in this chapter, but you can combine different kinds of methods to maximize mitigation—for instance, you could use a preprocessing method to de-bias the data, then train a model with it, and lastly, use a post-processing method to remove bias added by the model.

Preprocessing bias mitigation methods

These are some of the most important preprocessing or data-specific bias mitigation methods:

- **Unawareness:** Also known as **suppression**. The most straightforward way to remove bias is to exclude biased features from the dataset, but it's a naïve approach because you assume that bias is strictly contained in those features.

- **Feature engineering:** Sometimes, continuous features capture bias because there are so many sparse areas where the model can fill voids with assumptions or learn from outliers. It can do the same with interactions. Feature engineering can place guardrails. We will discuss this topic in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*.

- **Balancing:** Also known as **resampling**. On their own, representation problems are relatively easy to fix by balancing the dataset. The XAI library (`https://github.com/EthicalML/xai`) has a `balance` function that does this by random downsampling and upsampling of group representations. Downsampling, or under-sampling, is what we typically call sampling, which is just taking a certain percentage of the observations, whereas upsampling, or over-sampling, creates a certain percentage of random duplicates. Some strategies synthetically upsample rather than duplicate, such as the **Synthetic Minority Oversampling TEchnique** (**SMOTE**). However, we must caution that it's always preferable to downsample than upsample if you have enough data. It's best not to use only the balancing strategy if there are other possible bias problems.

- **Relabeling:** Also known as **massaging,** this is having an algorithm change the labels for observations that appear to be most biased, resulting in *massaged data* by ranking them. Usually, this is performed with a Naïve-Bayes classifier, and to maintain class distribution, it not only promotes some observations but demotes an equal amount.

- **Reweighing:** This method similarly ranks observations as relabeling does, but instead of flipping their labels it derives a weight for each one, which we can then implement in the learning process. Much like class weights are applied to each class, sample weights are applied to each observation or sample. Many regressors and classifiers, `LGBMClassifier` included, support sample weights. Even though, technically, reweighting doesn't touch the data and solution applied to the model, it is a preprocessing method because we detected bias in the data.

- **Disparate impact remover:** The authors of this method were very careful to abide by legal definitions of bias and preserve the integrity of the data without changing the labels or the protected attributes. It implements a repair process that attempts to remove bias in the remaining features. It's an excellent process to use whenever we suspect that's where most of the bias is located—that is, the features are highly correlated with the protected attributes, but it doesn't address bias elsewhere. In any case, it's a good baseline to use to understand how much of the bias is non-protected features.

- **Learning fair representations:** This leverages an adversarial learning framework. There's a generator (autoencoder) that creates representations of the data excluding the protected attribute, and a critic whose goal is that the learned representations within privileged and underprivileged groups are as close as possible.

- **Optimized preprocessing for discrimination prevention**: This method produces transformations through mathematical optimization of the data in such a way that overall probability distributions are maintained. At the same time, the correlation between protected attributes and the target is nullified. The result of this process is data that is distorted slightly to de-bias it.

Given that there are so many pre-processing methods, we will only employ two of them in this chapter. Still, if you are interested in using the ones we won't cover, they are available in the AIF360 library, and you can read about them in their documentation (`https://aif360.res.ibm.com/`).

The Reweighing method

The `Reweighing` method is fairly simple to implement. You initialize it by specifying the groups, then `fit` and `transform` the data as you would with any scikit-learn encoder or scaler. For those that aren't familiar with `fit`, the algorithm learns how to transform the provided data, and `transform` uses what was learned to transform it. The code can be seen in the following snippet:

```
reweighter= Reweighing(
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
reweighter.fit(train_ds)
train_rw_ds = reweighter.transform(train_ds)
```

The transformation derived from this process doesn't change the data but creates weights for each observation. The AIF360 library is equipped to factor these weights into the calculations of fairness, so we can use `BinaryLabelDatasetMetric`, as we have before, to compute different metrics. The code can be seen in the following snippet:

```
metrics_train_rw_ds = BinaryLabelDatasetMetric(
    train_rw_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
print('Statistical Parity Difference (SPD): %.4f' %
      metrics_train_rw_ds.statistical_parity_difference())
print('Disparate Impact (DI): %.4f' %
       metrics_train_rw_ds.disparate_impact())
print('Smoothed Empirical Differential Fairness(SEDF): %.4f'%
metrics_train_rw_ds.smoothed_empirical_differential_fairness())
```

The preceding code outputs the following:

```
Statistical Parity Difference (SPD):                 -0.0000
Disparate Impact (DI):                                1.0000
Smoothed Empirical Differential Fairness (SEDF):    0.1942
```

The weights have a perfect effect on SPD and DI, making them absolutely fair from those metrics' standpoints. However, note that SEDF is better than before, but not zero. This is because privileged and underprivileged groups only pertain to the `AGE_GROUP` protected attribute, but not `GENDER`. SEDF is a measure of intersectional fairness that reweighting does not address.

You would think that adding weights to observations would adversely impact predictive performance. However, this method was designed to maintain balance. In an unweighted dataset, all observations have a weight of one, and therefore the average of all the weights is one. While reweighting changes the weights for observations, the mean is still approximately one. You can check this is the case by taking the absolute difference in the mean of `instance_weights` between the original dataset and the reweighted one. It should be infinitesimal. The code can be seen in the following snippet:

```
np.abs(train_ds.instance_weights.mean() -\
        train_rw_ds.instance_weights.mean()) < 1e-6
```

So, how can you apply `instance_weights`?, you ask. Many model classes have a lesser-known attribute in the `fit` method, called `sample_weight`. You simply plug it in there, and while training, it will learn from observations in accordance with the respective weights. This method is shown in the following code snippet:

```
lgb_rw_mdl = lgb.LGBMClassifier(
    random_seed=rand,
    max_depth=6,
    num_leaves=33,
    **lgb_params
)
lgb_rw_mdl.fit(
    X_train,
    y_train,
    sample_weight=train_rw_ds.instance_weights
)
```

We can evaluate this model as we have with the base model, with `evaluate_class_mdl`. However, when we calculate the fairness metrics with `compute_aif_metrics`, we will save them in the model dictionary. Instead of looking at each method's outcomes one by one, we will compare them at the end of the section. Have a look at the following code snippet:

```
cls_mdls['lgb_1_rw'] = mldatasets.evaluate_class_mdl(
    lgb_rw_mdl,
    train_rw_ds.features,
    X_test,
    train_rw_ds.labels,
    y_test,
    plot_roc=False,
    plot_conf_matrix=True,
    show_summary=True,
    ret_eval_dict=True
)
```

```
test_pred_rw_ds = test_ds.copy(deepcopy=True)
test_pred_rw_ds.labels = cls_mdls['lgb_1_rw']['preds_test'
    ].reshape(-1,1)
test_pred_rw_ds.scores = cls_mdls['lgb_1_rw']['probs_test'
    ].reshape(-1,1)
metrics_test_rw_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_rw_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
cls_mdls['lgb_1_rw'].update(metrics_test_rw_dict)
```

The preceding snippet outputs the confusion matrix and performance metrics, as shown in *Figure 11.5*:


```
Accuracy_train:   0.8240      Accuracy_test:   0.8138
Precision_test:   0.5847      Recall_test:     0.4908
ROC-AUC_test:     0.7886      F1_test:         0.5337        MCC_test: 0.421
```
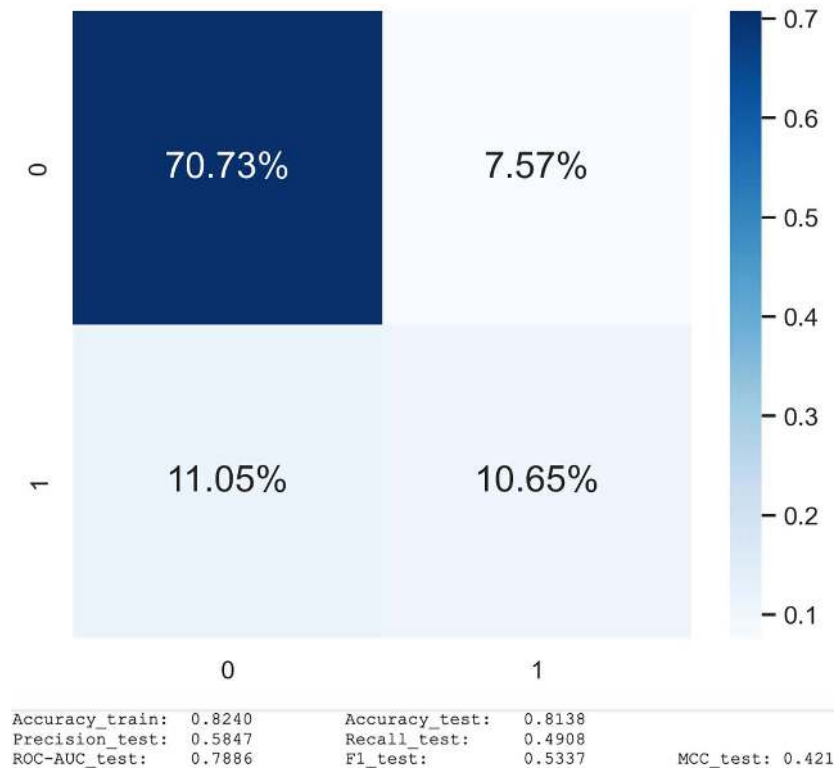
Figure 11.5: Evaluation of the LightGBM reweighted model

If you compare *Figure 11.5* to *Figure 11.4*, you can conclude that there's not much difference in predictive performance between the reweighted and the base model. This outcome was expected, but it's still good to verify it. Some bias-mitigation methods can adversely impact predictive performance, but reweighing did not. Neither should **Disparate Impact** (**DI**) remover, for that matter, which we will discuss next!

The disparate impact remover method

This method focuses on bias not located in the protected attribute (`AGE_GROUP`), so we will have to delete this feature during the process. To that end, we will need its index—in other words, what position it has within the list of columns. We can save this position (`protected_index`) as a variable, like this:

```
protected_index = train_ds.feature_names.index('AGE_GROUP')
```

DI remover is parametric. It requires a repair level between zero and one, so we need to find the optimal one. To that end, we can iterate through an array with different values for repair level (`levels`), initialize `DisparateImpactRemover` with each `level`, and `fit_transform` the data, which will de-bias the data. However, we then train the model without the protected attribute and use `BinaryLabelDatasetMetric` to assess the `disparate_impact`. Remember that DI is a ratio, so it's a metric that can be between over and under one, and an optimal DI is closest to one. Therefore, as we iterate across different repair levels, we will continuously save the model whose DI is closest to one. We will also append the DIs into an array for later use. Have a look at the following code snippet:

```python
di = np.array([])
train_dir_ds = None
test_dir_ds = None
lgb_dir_mdl = None
X_train_dir = None
X_test_dir = None
levels = np.hstack(
    [np.linspace(0., 0.1, 41), np.linspace(0.2, 1, 9)]
)
for level in tqdm(levels):
    di_remover = DisparateImpactRemover(repair_level=level)
    train_dir_ds_i = di_remover.fit_transform(train_ds)
    test_dir_ds_i = di_remover.fit_transform(test_ds)
    X_train_dir_i = np.delete(
        train_dir_ds_i.features,
        protected_index,
        axis=1
    )
```

```python
    X_test_dir_i = np.delete(
        test_dir_ds_i.features,
        protected_index,
        axis=1
    )
    lgb_dir_mdl_i = lgb.LGBMClassifier(
        random_seed=rand,
        max_depth=5,
        num_leaves=33,
        **lgb_params
    )
    lgb_dir_mdl_i.fit(X_train_dir_i, train_dir_ds_i.labels)
    test_dir_ds_pred_i = test_dir_ds_i.copy()
    test_dir_ds_pred_i.labels = lgb_dir_mdl_i.predict(
        X_test_dir_i
    )
    metrics_test_dir_ds = BinaryLabelDatasetMetric(
        test_dir_ds_pred_i,
        unprivileged_groups=underprivileged_groups,
        privileged_groups=privileged_groups
    )
    di_i = metrics_test_dir_ds.disparate_impact()
    if (di.shape[0]==0) or (np.min(np.abs(di-1)) >= abs(di_i-1)):
        print(abs(di_i-1))
        train_dir_ds = train_dir_ds_i
        test_dir_ds = test_dir_ds_i
        X_train_dir = X_train_dir_i
        X_test_dir = X_test_dir_i
        lgb_dir_mdl = lgb_dir_mdl_i
    di = np.append(np.array(di), di_i)
```

To observe the DI at different repair levels, we can use the following code, and if you want to zoom in on the area where the best DI is located, just uncomment the xlim line:

```python
plt.plot(levels, di, marker='o')
```

The preceding code generates the following output. As you can tell by this, there's an optimal repair level somewhere between 0 and 0.1 because that's where it gets closest to one:
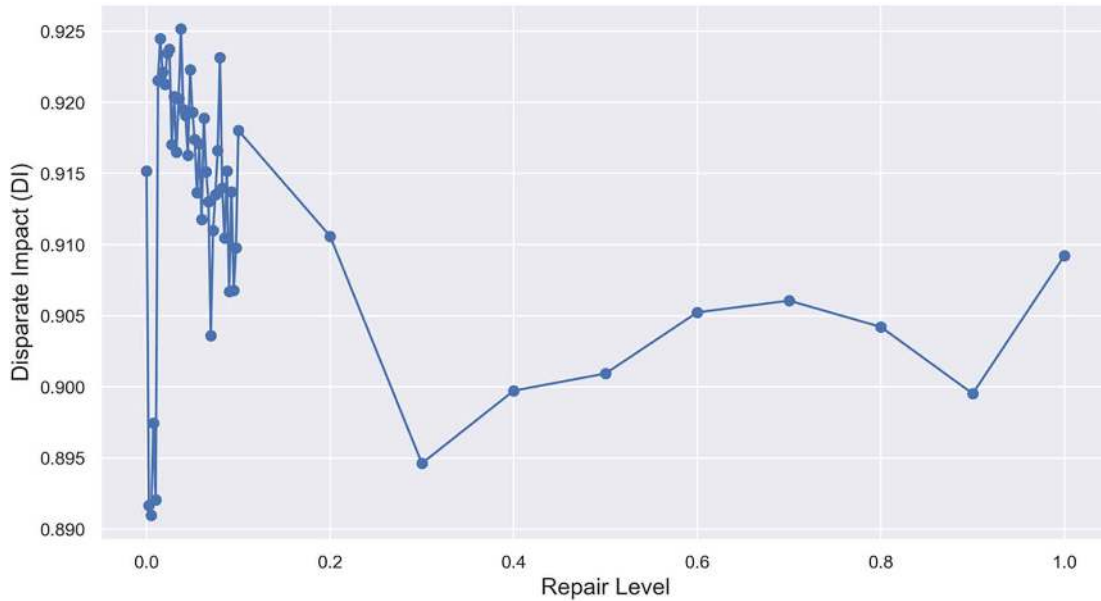
Figure 11.6: DI at different DI remover repair levels

Now, let's evaluate the best DI-repaired model with `evaluate_class_mdl` and compute the fairness metrics (`compute_aif_metrics`). We won't even plot the confusion matrix this time, but we will save all results into the `cls_mdls` dictionary for later inspection. The code can be seen in the following snippet:

```
cls_mdls['lgb_1_dir'] = mldatasets.evaluate_class_mdl(
    lgb_dir_mdl,
    X_train_dir,
    X_test_dir,
    train_dir_ds.labels,
    test_dir_ds.labels,
    plot_roc=False,
    plot_conf_matrix=False,
    show_summary=False,
    ret_eval_dict=True
)
test_pred_dir_ds = test_ds.copy(deepcopy=True)
test_pred_dir_ds.labels = cls_mdls['lgb_1_dir']['preds_test'
].reshape(-1,1)
metrics_test_dir_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_dir_ds,
```

```
        unprivileged_groups=underprivileged_groups,
        privileged_groups=privileged_groups
    )
    cls_mdls['lgb_1_dir'].update(metrics_test_dir_dict)
```

The next link in the chain after data is the model, so even if we de-bias the data, the model introduces bias on its own, thus it makes sense to train models that are equipped to deal with it, which is what we will learn how to do next!

In-processing bias mitigation methods

These are some of the most important in-processing or model-specific bias mitigation methods:

- **Cost-sensitive training**: We are already incorporating this method into every LightGBM model trained in this chapter through the `scale_pos_weight` parameter. It's typically used in imbalanced classification problems and is simply seen as a means to improve accuracy for minor classes. However, given that imbalances with classes tend to favor some groups over others, this method can also be used to mitigate bias, but there are no guarantees that it will. It can be incorporated as class weights or by creating a custom loss function. The implementation will vary according to the model class and what costs are associated with the bias. If they grow linearly with misclassifications, the class weighting will suffice, but otherwise, a custom loss function is recommended.

- **Constraints**: Many model classes support monotonic and interaction constraints, and **TensorFlow Lattice** (**TFL**) offers more advanced custom shape constraints. These ensure that relationships between features and targets are restricted to a certain pattern, placing guardrails at the model level. There are many reasons you would want to employ them, but chief among them is to mitigate bias. We will discuss this topic in *Chapter 12*, *Monotonic Constraints and Model Tuning for Interpretability*.

- **Prejudice remover regularizer**: This method defines prejudice as the statistical dependence between the sensitive and target variables. However, the aim of this method is to minimize indirect prejudice, which excludes the prejudice that can be avoided by simply removing the sensitive variable. Therefore, the method starts by quantifying it with a **Prejudice Index** (**PI**), which is the mutual information between the target and sensitive variable. Incidentally, we covered mutual information in *Chapter 10*, *Feature Selection and Engineering for Interpretability*. Then, along with L2, the PI is incorporated into a custom regularization term. In theory, any model classifier can regularize using the PI-based regularizer, but the only implementation, so far, uses logistic regression.

- **Gerry fair classifier**: This is inspired by the concept of **fairness gerrymandering**, which has the appearance of fairness in one group but lacks fairness when subdivided into subgroups. The algorithm leverages a **fictitious play** game-theory-inspired approach in which you have a zero-sum game between a *learner* and an *auditor*. The learner minimizes the prediction error and aggregate fairness-based penalty term. The auditor takes it one step further by penalizing the learner based on the worst outcomes observed in the most unfairly treated subgroup.

The game's objective is to achieve a **Nash equilibrium**, which is achieved when two non-cooperative players with possibly contradictory aims reach a solution that partially satisfies both. In this case, the learner gets a minimal prediction error and aggregate unfairness, and the auditor gets minimal subgroup unfairness. The implementation of this method is model-agnostic.

- **Adversarial debiasing**: Similar to the gerry fair classifier, adversarial debiasing leverages two opposing actors, but this time it's with two neural networks: a predictor and an adversary. We maximize the predictor's ability to predict the target while minimizing the adversary's ability to predict the protected feature, thus increasing the equality of odds between privileged and underprivileged groups.

- **Exponentiated gradient reduction**: This method automates cost-sensitive optimization by reducing it to a sequence of such problems and using fairness constraints concerning protected attributes such as demographic parity or equalized odds. It is model-agnostic but limited only to scikit-learn-compatible binary classifiers.

Given that there are so many in-processing methods, we will only employ two of them in this chapter. Still, if you are interested in using ones we won't cover, they are available in the AIF360 library and documentation.

The exponentiated gradient reduction method

The `ExponentiatedGradientReduction` method is an implementation of cost-sensitive training with constraints . We initialize it with a base estimator, the maximum number of iterations to perform (`max_iter`) and specify the disparity `constraints` to use. Then, we `fit` it. This method can be seen in the following code snippet:

```
lgb_egr_mdl = ExponentiatedGradientReduction(
    estimator=lgb_base_mdl,
    max_iter=50,
    constraints='DemographicParity'
)
lgb_egr_mdl.fit(train_ds)
```

We can use the `predict` function to get the training and test predictions and then employ `evaluate_class_metrics_mdl` and `compute_aif_metrics` to obtain predictive performance and fairness metrics, respectively. We place both into the `cls_mdls` dictionary, as illustrated in the following code snippet:

```
train_pred_egr_ds = lgb_egr_mdl.predict(train_ds)
test_pred_egr_ds = lgb_egr_mdl.predict(test_ds)
cls_mdls['lgb_2_egr'] = mldatasets.evaluate_class_metrics_mdl(
    lgb_egr_mdl,
    train_pred_egr_ds.labels,
    test_pred_egr_ds.scores,
    test_pred_egr_ds.labels,
    y_train,
    y_test
)
```

```
metrics_test_egr_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_egr_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
cls_mdls['lgb_2_egr'].update(metrics_test_egr_dict)
```

Next, we will learn about a partially model-agnostic in-processing method that takes into account intersectionality.

The gerry fair classifier method

The gerry fair classifier is partially model-agnostic. It only supports linear models, **Support Vector Machines** (**SVMs**), kernel regression, and decision trees. We initialize GerryFairClassifier by defining a regularization strength (C), a fairness approximation for early stopping (gamma), whether to be verbose (printflag), the maximum number of iterations (max_iters), the model (predictor), and the fairness notion to employ (fairness_def). We will use the fairness notion of false negatives ("FN") to compute the fairness violations' weighted disparity. Once it's been initialized, all we need to do is fit it and enable early_termination to stop if it hasn't improved in five iterations. The code is shown in the following snippet:

```
dt_gf_mdl = GerryFairClassifier(
    C=100,
    gamma=.005,
    max_iters=50,
    fairness_def='FN',
    printflag=True,
    predictor=tree.DecisionTreeRegressor(max_depth=3)
)
dt_gf_mdl.fit(train_ds, early_termination=True)
```

We can use the predict function to get the training and test predictions and then employ evaluate_class_metrics_mdl and compute_aif_metrics to obtain predictive performance and fairness metrics, respectively. We place both into the cl_smdls dictionary, as illustrated in the following code snippet:

```
train_pred_gf_ds = dt_gf_mdl.predict(train_ds, threshold=False)
test_pred_gf_ds = dt_gf_mdl.predict(test_ds, threshold=False)
cls_mdls['dt_2_gf'] = mldatasets.evaluate_class_metrics_mdl(
    dt_gf_mdl,
    train_pred_gf_ds.labels,
    None,
    test_pred_gf_ds.labels,
    y_train,
    y_test
)
```

```
metrics_test_gf_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_gf_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
cls_mdls['dt_2_gf'].update(metrics_test_gf_dict)
```

The next and last link in the chain after the model is inference, so even if you de-bias the data and the model there might be some bias left, thus it makes sense to deal with it in this stage too, which is what we will learn how to do next!

Post-processing bias mitigation methods

These are some of the most important post-processing or inference-specific bias mitigation methods:

- **Prediction abstention:** This has many potential benefits such as fairness, safety, and controlling costs, but which one applies will depend on your problem. Typically, a model will return all predictions, even low-confidence ones—that is, predictions that are close to the classification threshold or when the model returns confidence intervals that fall outside of a predetermined threshold. When fairness is involved, if we change predictions to **I don't know** (**IDK**) in low-confidence regions, the model will likely become fairer as a side-effect when we assess fairness metrics only against predictions that were made. It is also possible to make prediction abstention an in-processing method. A paper called *Predict Responsibly: Increasing Fairness by Learning to Defer* discusses two approaches to do this, by training a model to either **punt** (learn to predict IDK) or **defer** (predict IDK when the odds of being correct are lower than an expert opinion). Another paper called *The Utility of Abstaining in Binary Classification* employs a reinforcement learning framework called **Knows What It Knows** (**KWIK**), which has self-awareness of its mistakes but allows for abstentions.

- **Equalized odds postprocessing:** Also known as disparate mistreatment, this ensures that privileged and underprivileged groups have equal treatment for misclassifications, whether false-positive or false-negative. It finds optimal probability thresholds with which changing the labels equalizes the odds between groups.

- **Calibrated equalized odds postprocessing:** Instead of changing the labels, this method modifies the probability estimates so that they are on average equal. It calls this calibration. However, this constraint cannot be satisfied for false positives and false negatives concurrently, so you are forced to prefer one over the other. Therefore, it is advantageous in cases where recall is far more important than precision or vice versa, and there are benefits to calibrating the estimated probabilities.

- **Reject option classification:** This method leverages the intuition that predictions around the decision boundary tend to be the least fair. It then finds an optimal band around the decision boundary for which flipping the labels for underprivileged and privileged groups yields the most equitable outcomes.

We will only employ two of these post-processing methods in this chapter. Reject option classification is available in the AIF360 library and documentation.

The equalized odds post-processing method

The equalized odds post-processing method (`EqOddsPostprocessing`) is initialized with the groups we want to equalize odds for and the random `seed`. Then, we `fit` it. Note that fitting takes two datasets: the original one (`test_ds`) and then the dataset with predictions for our base model (`test_pred_ds`). What `fit` does is compute the optimal probability thresholds. Then, `predict` creates a new dataset where these thresholds have changed the `labels`. The code can be seen in the following snippet:

```
epp = EqOddsPostprocessing(
    privileged_groups=privileged_groups,
    unprivileged_groups=underprivileged_groups,
    seed=rand
)
epp = epp.fit(test_ds, test_pred_ds)
test_pred_epp_ds = epp.predict(test_pred_ds)
```

We can employ `evaluate_class_metrics_mdl` and `compute_aif_metrics` to obtain predictive performance and fairness metrics for **Equal-Proportion Probability** (**EPP**), respectively. We place both into the `cls_mdls` dictionary. The code can be seen in the following snippet:

```
cls_mdls['lgb_3_epp'] = mldatasets.evaluate_class_metrics_mdl(
    lgb_base_mdl,
    cls_mdls['lgb_0_base']['preds_train'],
    test_pred_epp_ds.scores,
    test_pred_epp_ds.labels,
    y_train,
    y_test
)
metrics_test_epp_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_epp_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
cls_mdls['lgb_3_epp'].update(metrics_test_epp_dict)
```

Next, we will learn about another post-processing method. The main difference is that it calibrates the probability scores rather than only changing the predicted labels.

The calibrated equalized odds postprocessing method

Calibrated equalized odds (`CalibratedEqOddsPostprocessing`) is implemented exactly like equalized odds, except it has one more crucial attribute (`cost_constraint`). This attribute defines which constraint to satisfy since it cannot make the scores fair for FPRs and FNRs simultaneously. We choose FPR and then `fit`, `predict`, and `evaluate`, as we did for equalized odds. The code can be seen in the following snippet:

```
cpp = CalibratedEqOddsPostprocessing(
    privileged_groups=privileged_groups,
    unprivileged_groups=underprivileged_groups,
    cost_constraint="fpr",
    seed=rand
)
cpp = cpp.fit(test_ds, test_pred_ds)
test_pred_cpp_ds = cpp.predict(test_pred_ds)
cls_mdls['lgb_3_cpp'] = mldatasets.evaluate_class_metrics_mdl(
    lgb_base_mdl,
    cls_mdls['lgb_0_base']['preds_train'],
    test_pred_cpp_ds.scores,
    test_pred_cpp_ds.labels,
    y_train,
    y_test
)
metrics_test_cpp_dict, _ = mldatasets.compute_aif_metrics(
    test_ds,
    test_pred_cpp_ds,
    unprivileged_groups=underprivileged_groups,
    privileged_groups=privileged_groups
)
cls_mdls['lgb_3_cpp'].update(metrics_test_cpp_dict)
```

Now that we have tried six bias mitigation methods, two at every level, we can compare them against each other and the base model!

Tying it all together!

To compare the metrics for all the methods, we can take the dictionary (`cls_mdls`) and place it in the DataFrame (`cls_metrics_df`). We are only interested in a few performance metrics and most of the fairness metrics recorded. Then, we output the DataFrame sorted by test accuracy and with all the fairness metrics color-coded. The code can be seen in the following snippet:

```
cls_metrics_df = pd.DataFrame.from_dict(cls_mdls, 'index')[
    [
```

```
        'accuracy_train',
        'accuracy_test',
        'f1_test',
        'mcc_test',
        'SPD',
        'DI',
        'AOD',
        'EOD',
        'DFBA'
    ]
]
metrics_fmt_dict = dict(
    zip(cls_metrics_df.columns,['{:.1%}']*3+ ['{:.3f}']*6)
)
cls_metrics_df.sort_values(
    by='accuracy_test',
    ascending=False
).style.format(metrics_fmt_dict)
```

The preceding snippet outputs the following DataFrame:

	accuracy_train	accuracy_test	f1_test	mcc_test	SPD	DI	AOD	EOD	DFBA
dt_2_gf	82.1%	**82.6%**	48.1%	0.413	-0.055	0.939	-0.043	-0.022	0.252
lgb_0_base	82.6%	81.7%	53.3%	**0.424**	-0.068	0.919	-0.055	-0.026	0.233
lgb_1_rw	82.4%	81.4%	**53.4%**	0.421	-0.037	0.955	-0.017	**-0.002**	0.035
lgb_1_dir	82.4%	81.3%	53.0%	0.417	-0.062	0.925	-0.049	-0.021	0.255
lgb_2_egr	82.7%	81.1%	52.3%	0.410	-0.039	0.953	-0.013	-0.012	-0.082
lgb_3_epp	82.6%	81.1%	51.8%	0.406	**-0.026**	**0.969**	**-0.001**	0.002	**-0.014**
lgb_3_cpp	82.6%	26.2%	21.3%	-0.306	-0.071	0.761	-0.064	-0.126	0.043

Figure 11.7: Comparison of all bias mitigation methods with different fairness metrics

Figure 11.7 shows that most methods yielded models that are fairer than the base model for SPD, DI, AOD, and EOD. Calibrated equalized odds post-processing (`lgb_3_cpp`) was the exception, but it had one of the best DFBAs but it yielded a suboptimal DI because of the lopsided nature of the calibration. Note that this method is particularly good at achieving parity for FPR or FNR while calibrating scores, but none of these fairness metrics are useful for picking up on this. Instead, you could create a metric that's the ratio between FPRs, as we did in *Chapter 6*, *Anchors and Counterfactual Explanations*. Incidentally, this would be the perfect use case for **Calibrated Equalized Odds** (**CPP**).

The method that obtained the best SPD, DI, AOD, and DFBA, and the second-best EOD was equalized odds post-processing (`lgb_3_epp`), so let's visualize fairness for it using XAI's plots. To this end, we first create a DataFrame with the test examples (`test_df`) and then use `replace` to make an `AGE_GROUP` categorical and obtain the list of categorical columns (`cat_cols_l`). Then, we can compare different metrics (`metrics_plot`) using the true labels (`y_test`), predicted probability scores for the EPP model, the DataFrame (`test_df`), the protected attribute (`cross_cols`), and categorical columns. We can do the same for the **Receiver Operating Characteristic** (**ROC**) plot (`roc_plot`) and the **Precision-Recall** (**PR**) plot (`pr_plot`). The code can be seen in the following snippet:

```
test_df = ccdefault_bias_df.loc[X_test.index]
test_df['AGE_GROUP'] = test_df.AGE_GROUP.replace(
    {0:'underprivileged', 1:'privileged'}
)
cat_cols_l = ccdefault_bias_df.dtypes[lambda x: x==np.int8
                                     ].index.tolist()
_ = xai.metrics_plot(
    y_test,cls_mdls['lgb_3_epp']['probs_test'],
    df=test_df, cross_cols=['AGE_GROUP'],
    categorical_cols=cat_cols_l
)
_ = xai.roc_plot(
    y_test, cls_mdls['lgb_3_epp']['probs_test'],
    df=test_df, cross_cols=['AGE_GROUP'],
    categorical_cols=cat_cols_l
)
_ = xai.pr_plot(
    y_test,
    cls_mdls['lgb_3_epp']['probs_test'],
    df=test_df, cross_cols=['AGE_GROUP'],
    categorical_cols=cat_cols_l
)
```

The preceding snippet outputs the three plots in *Figure 11.8*. The first one shows that even the fairest model still has some disparities between both groups, especially between precision and recall and, by extension, F1 score, which is their average. However, the ROC curve shows how close both groups are from an FPR versus a TPR standpoint. The third plot is where the disparities in precision and recall become even more evident. This all demonstrates how hard it is to keep a fair balance on all fronts! Some methods are best for making one aspect perfect but nothing else, while others are pretty good on a handful of aspects but nothing else. Despite the shortcomings of the methods, most of them achieved a sizable improvement. Ultimately, choosing methods will depend on what you most care about, and combining them is also recommended for maximum effect! The output is shown here:
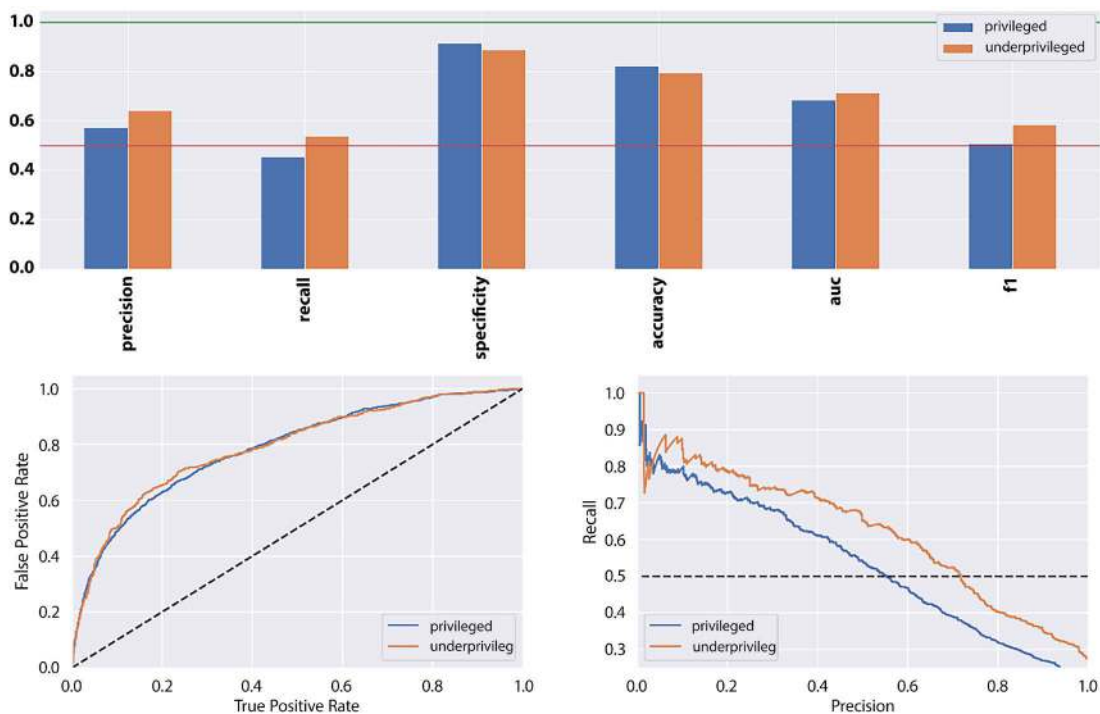
Figure 11.8: Plots demonstrating fairness for the fairest model

We've concluded the bias mitigation exercise and will move on to the causal inference exercise, where we will discuss how to ensure fair and robust policies.

Creating a causal model

Decision-making will often involve understanding cause and effect. If the effect is desirable, you can decide to replicate its cause, or otherwise avoid it. You can change something on purpose to observe how it changes outcomes, or trace an accidental effect back to its cause, or simulate which change will produce the most beneficial impact. Causal inference can help us do all this by creating causal graphs and models. These tie all variables together and estimate effects to make more principled decisions. However, to properly assess the impact of a cause, whether by design or accident, you'll need to separate its effect from confounding variables.

The reason causal inference is relevant to this chapter is that the bank's policy decisions have the power to impact cardholder livelihoods significantly and, given the rise in suicides, even life and death. Therefore, there's a moral imperative to assess policy decisions with the utmost care.

The Taiwanese bank conducted a lending policy experiment for 6 months. The bank saw the writing on the wall and knew that the customers with the highest risk of default would somehow be written off their balance sheets in a way that diminished those customers' financial obligations. Therefore, the experiment's focus only involved what the bank considered salvageable, which were low-to-mid risk-of-default customers, and now that the experiment has ended, they want to understand how the following policies have impacted customer behavior:

- **Lower credit limit**: Some customers had their credit limit reduced by 25%.
- **Payment plan**: They were given 6 months to pay back their current credit card debt. In other words, the debt was split up into six parts, and every month they would have to pay one part.
- **Both measures**: A reduction in credit limit and the payment plan.

Also, prevailing credit card interest rates in Taiwan were around 16-20% in 2005, but the bank caught wind that these would be capped at 4% by the Taiwanese Financial Supervisory Commission. Therefore, they ensured all customers in the experiment were automatically provided with interest rates at that level. Some bank executives thought this would only aggravate the indebtedness and create more "credit card slaves" in the process. These concerns prompted the proposal to conduct the experiment with a lower credit card limit as a countermeasure. On the other hand, the payment plan was devised to understand whether debt relief gave customers breathing room to use the card without fear.

On the business side, the rationale was that a healthy level of spending needed to be encouraged because with lower interest rates, the bulk of the profits would come from payment processing, cashback partnerships, and other sources tied to spending and, in turn, increased customer longevity. Yet, this would also be beneficial to customers because if they were more profitable as spenders than as debtors, it meant the incentives were in place to keep them from becoming the latter. All this justified the use of estimated lifetime value (`_LTV`) as a proxy metric for how the experiment's outcome benefited both the bank and its customers. For years, the bank has been using a reasonably accurate calculation to estimate how much value a credit card holder will provide to the bank given their spending and payment history, and parameters such as limits and interest rates.

In the parlance of experimental design, the chosen policy is called a **treatment**, and along with the three treated groups, there's a control group that wasn't prescribed a treatment—that is, no change in policy at all, not even the lower interest rates. Before we move forward, let's first initialize a list with the treatment names (`treatment_names`) and one that includes even the control group (`all_treatment_names`), as follows:

```
treatment_names = [
    'Lower Credit Limit',
    'Payment Plan',
    'Payment Plan &Credit Limit'
]
all_treatment_names = np.array(["None"] + treatment_names)
```

Now, let's examine the results of the experiment to help us design an optimal causal model.

Understanding the results of the experiment

A fairly intuitive way of assessing the effectiveness of a treatment is by comparing their outcomes. We want to know the answers to the following two simple questions:

- Did the treatment decrease the default rate compared to the control group?
- Were the spending behaviors conducive to an increase in lifetime value estimates?

We can visualize both in a single plot. To this end, we obtain a pandas series with the percentage for each group that defaulted (pct_s), then another one with the sum of lifetime values for each group (ltv_s) in thousands of NTD (K$). We put both series into a pandas DataFrame and plot it, as illustrated in the following code snippet:

```python
pct_s = ccdefault_causal_df[
    ccdefault_causal_df.IS_DEFAULT==1]
    .groupby(['_TREATMENT'])
    .size()
    /ccdefault_causal_df.groupby(['_TREATMENT']).size()
ltv_s = ccdefault_causal_df.groupby(
    ['_TREATMENT'])['_LTV'].sum()/1000
plot_df = pd.DataFrame(
    {'% Defaulted':pct_s,
     'Total LTV, K$':ltv_s}
)
plot_df.index = all_treatment_names
ax = plot_df.plot(secondary_y=['Total LTV, K$'], figsize=(8,5))
ax.get_legend().set_bbox_to_anchor((0.7, 0.99))
plt.grid(False)
```

The preceding snippet outputs the plot shown in *Figure 11.9*. It can be inferred that all treatments fare better than the control group. The lowering of the credit limit on its own decreases the default rate by over 12% and more than doubles the estimated LTV, while the payment plan only decreases the defaults by 3% and increases the LTV by about 85%. However, both policies combined quadrupled the control group's LTV and reduced the default rate by nearly 15%! The output can be seen here:
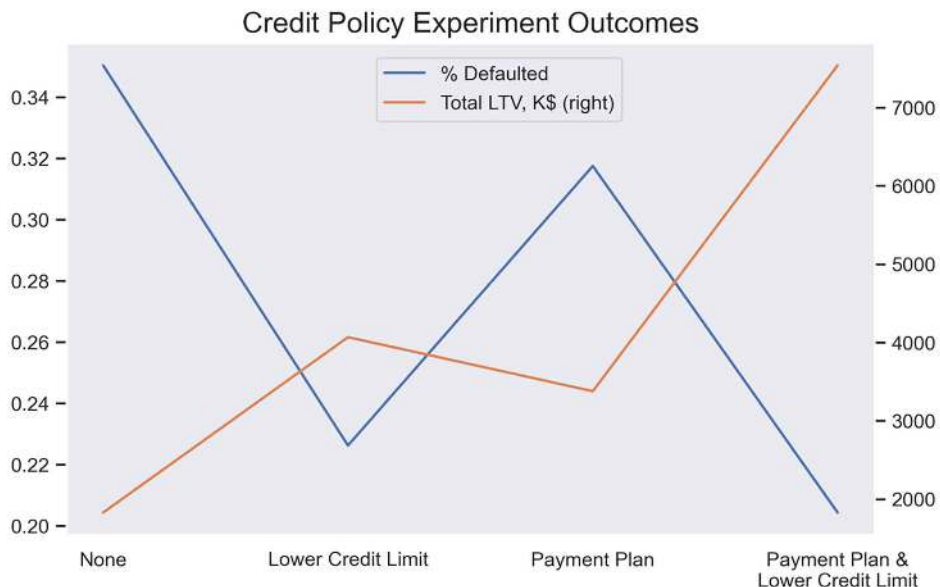
Figure 11.9: Outcomes for treatment experiment with different credit policies

Before bank executives rejoice that they have found the winning policy, we must examine how they distributed it among the credit cardholders in the experiment. We learned that they chose the treatment according to the risk factor, which is measured by the _risk_score variable. However, lifetime value is largely affected by the credit limit available (_CC_LIMIT), so we must take that into account. One way to understand the distribution is by plotting both variables against each other in a scatter plot color-coded by _TREATMENT. The code for this can be seen in the following snippet:

```
sns.scatterplot(
    x=ccdefault_causal_df['_CC_LIMIT'].values,
    y=ccdefault_causal_df['_risk_score'].values,
    hue=all_treatment_names[ccdefault_causal_df['_TREATMENT'].values],
    hue_order = all_treatment_names
)
```

The preceding code generated the plot in *Figure 11.10*. It shows that the three treatments correspond to different risk levels, while the control group (None) is spread out more vertically. The choice to assign treatments based on risk level also meant that they unevenly distributed the treatments based on _CC_LIMIT. We ought to ask ourselves if this experiment's biased conditions make it even viable to interpret the outcomes. Have a look at the following output:
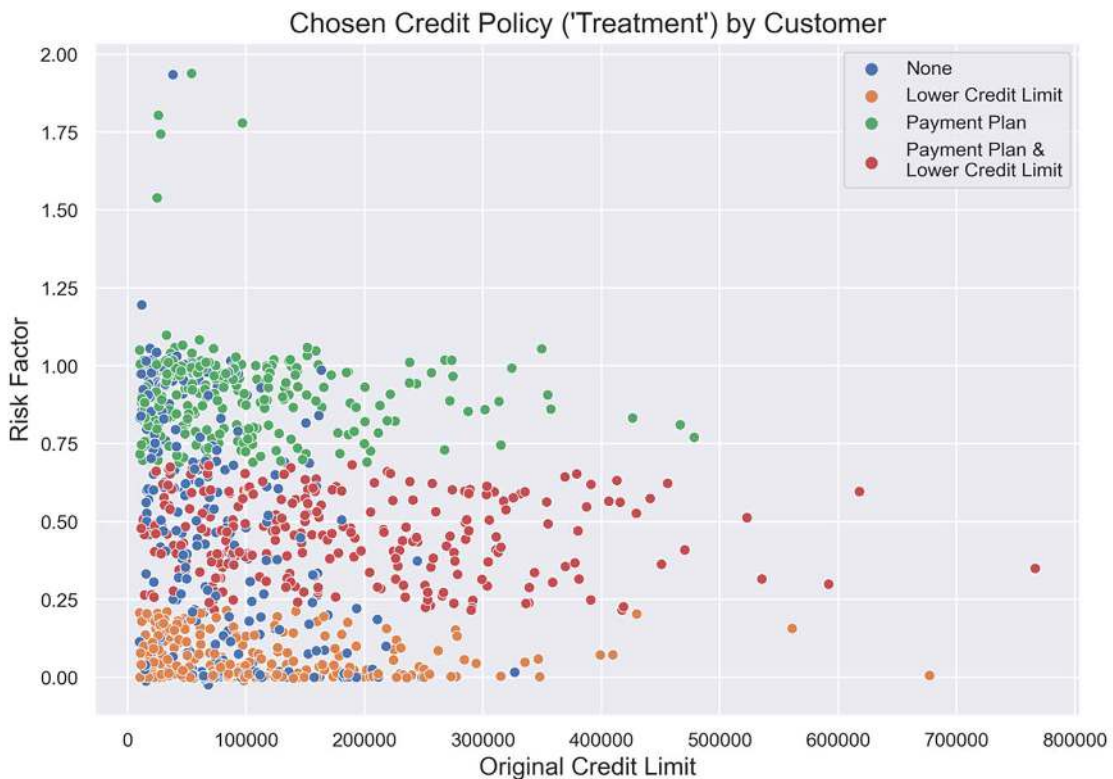
Figure 11.10: Risk factors versus original credit limit

The scatterplot in *Figure 11.10* demonstrates the stratification of the treatments across risk factors. However, scatter plots can be challenging to interpret to understand distributions. For that, it's best to use a **Kernel Density Estimate** (**KDE**) plot. So, let's see how _CC_LIMIT and lifetime value (_LTV) is distributed across all treatments with Seaborn's displot. Have a look at the following code snippet:

```
sns.displot(
    ccdefault_causal_df,
    x="_CC_LIMIT",
    hue="_TREATMENT",
    kind="kde",
    fill=True
)
sns.displot(
    ccdefault_causal_df,
    x="_LTV",
    hue="_TREATMENT",
    kind="kde", fill=True
)
```

The preceding snippet produced the two KDE plots in *Figure 11.11*. We can easily tell how far apart all four distributions are for both plots, mostly regarding treatment #3 (**Payment Plan & Lower Credit Limit**), which tends to be centered significantly more to the right and has a longer and fatter right tail. You can view the output here:
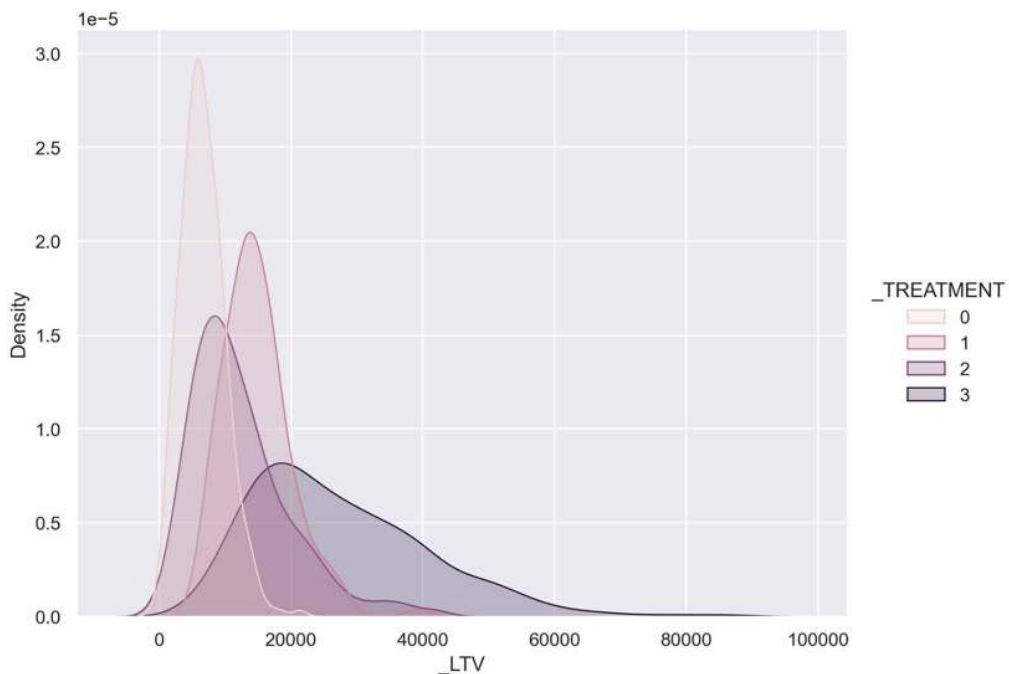
Figure 11.11: KDE distributions for _CC_LIMIT and _LTV by _TREATMENT

Ideally, when you design an experiment such as this, you should aim for equal distribution among all groups based on any pertinent factors that could alter the outcomes. However, this might not always be feasible, either because of logistical or strategic constraints. In this case, the outcome (`_LTV`) varies according to customer credit card limits (`_CC_LIMIT`), the **heterogeneity feature**—in other words, the varying feature that directly impacts the treatment effect, also known as the **heterogeneous treatment effect modifier**. We can create a causal model that includes both the `_TREATMENT` feature and the effect modifier (`_CC_LIMIT`).

Understanding causal models

The causal model we will build can be separated into four components, as follows:

- **Outcome** (*Y*): The outcome variable(s) of the causal model.
- **Treatments** (*T*): The treatment variable(s) that influences the outcome.
- **Effect modifiers** (*X*): The variable(s) that influences the effect's heterogeneity conditioning it. It sits in between the treatment and the outcome.
- **Controls** (*W*): Also known as **common causes** or **confounders**. They are the features that influence both the outcome and the treatment.

We will start by identifying each one of these components in the data as separate `pandas` DataFrames, as follows:

```
W = ccdefault_causal_df[
    [
        '_spend','_tpm', '_ppm', '_RETAIL','_URBAN', '_RURAL',
        '_PREMIUM'
    ]
]
X = ccdefault_causal_df[['_CC_LIMIT']]
T = ccdefault_causal_df[['_TREATMENT']]
Y = ccdefault_causal_df[['_LTV']]
```

We will use the **Doubly Robust Learning** (**DRL**) method to estimate the treatment effects. It's called "doubly" because it leverages two models, as follows:

- It predicts the outcome with a *regression model*, as illustrated here:

$$Y \sim W + X$$

- It predicts the treatment with a *propensity model*, as illustrated here:

$$Y \sim W + X$$

It's also *robust* because of the final stage, which combines both models while maintaining many desirable statistical properties such as confidence intervals and asymptotic normality. More formally, the estimation leverages regression model *g* and propensity model *p* conditional on treatment *t*, like this:

$$Y_t = g_t(W, X) + \epsilon_t$$

It also does this:

$$\Pr[T = t | X, W] = p_t(W, X)]$$

The goal is to derive the **Conditional Average Treatment Effect** (**CATE**) denoted as $\theta_t(X)$ associated with each treatment **t** given heterogeneous effect **X**. First, the DRL method de-biases the regression model by applying the inverse propensity, like this:

$$Y_{i,t}^{\text{DRL}} = g_t(W_i, X_i) + \frac{Y_i - g_t(W_i, X_i)}{p_t(W_i, X_i)} + 1\{T_i = t\}$$

How exactly to estimate coefficients $\theta_t(X)$ from model $Y_{i,t}^{\text{DRL}}$ will depend on the DRL variant employed. We will use a linear variant (`LinearDRLearner`) so that it returns coefficients and intercepts, which can be easily interpreted. It derives $\theta_t(X)$ by running **ordinary linear regression** (**OLS**) for the outcome differences between a treatment t and the control $(Y_{i,t}^{\text{DRL}} - Y_{i,0}^{\text{DRL}})$ on x_i. This intuitively makes sense because the estimated effect of a treatment minus the estimated effect of the absence of a treatment ($t = 0$) is the *net* effect of said treatment.

Now, with all the theory out of the way, let's dig in!

Initializing the linear doubly robust learner

We can initialize a `LinearDRLearner` from the `econml` library, which we call `drlearner`, by specifying any scikit-learn-compatible regressor (`model_regression`) and classifier (`model_propensity`). We will use XGBoost for both, but note that the classifier has an `objective=multi:softmax` attribute. Remember that we have multiple treatments, so it's a multiclass classification problem. The code can be seen in the following snippet:

```
drlearner = LinearDRLearner(
    model_regression=xgb.XGBRegressor(learning_rate=0.1),
    model_propensity=xgb.XGBClassifier(learning_rate=0.1,
    max_depth=2,
    objective="multi:softmax"),
    random_state=rand
)
```

If you want to understand what both the regression and propensity model are doing, you can easily fit `xgb.XGBRegressor().fit(W.join(X),Y)` and `xgb.XGBClassifier(objective="multi:softmax").fit(W.join(X), T)` models. We won't do this now but if you are curious, you could evaluate their performance and even run feature importance methods to understand what influences their predictions individually. The causal model brings them together with the DRL framework, leading to different conclusions.

Fitting the causal model

We can use `fit` in the `drlearner` to fit the causal model leveraging the `dowhy` wrapper of `econml`. The first attributes are the `Y`, `T`, `X`, and `Y` components: `pandas` DataFrames. Optionally, you can provide variable names for each of these components: the column names of each of the `pandas` DataFrames. Lastly, we would like to estimate the treatment effects. Optionally, we can provide the effect modifiers (`X`) to do this with, and we will use half of this data to do so, as illustrated in the following code snippet:

```
causal_mdl = drlearner.dowhy.fit(
    Y,
    T,
    X=X,
    W=W,
    outcome_names=Y.columns.to_list(),
    treatment_names=T.columns.to_list(),
    feature_names=X.columns.to_list(),
    confounder_names=W.columns.to_list(),
    target_units=X.iloc[:550].values
)
```

With the causal model initialized, we can visualize it. The `pydot` library with `pygraphviz` can do this for us. Please note that this library is difficult to configure in some environments, so it might not load and show you the much less attractive default graphic instead with `view_model`. Don't worry if this happens. Have a look at the following code snippet:

```
try:
    display(Image(to_pydot(causal_mdl._graph._graph).create_png()))
except:
    causal_mdl.view_model()
```

The code in the preceding snippet outputs the model diagram shown here. With it, you can appreciate how all the variables connect:
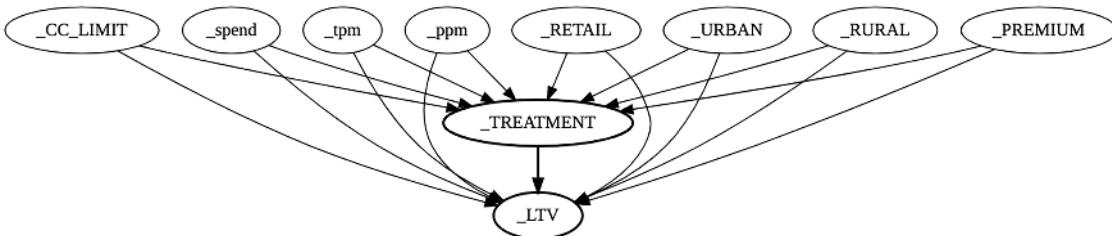
Figure 11.12: Causal model diagram

The causal model has already been fitted, so let's examine and interpret the results, shall we?

Understanding heterogeneous treatment effects

Firstly, it's important to note how the `dowhy` wrapper of `econml` has cut down on a few steps with the `dowhy.fit` method. Usually, when you build a `CausalModel` such as this one directly with `dowhy`, it has a method called `identify_effect` that derives the probability expression for the effect to be estimated (the *identified estimand*). In this case, this is called the **Average Treatment Effect** (**ATE**). Then, another method called `estimate_effect` takes this expression and the models it's supposed to tie together (regression and propensity). With them, it computes both the ATE, $Y_{i,t}^{\text{DRL}}$, and CATE, $\theta_{i,t}(X)$, for every outcome i and treatment t. However, since we used the wrapper to `fit` the causal model, it automatically takes care of both the identification and estimation steps.

You can access the identified ATE with the `identified_estimand_` property and the estimate results with the `estimate_` property for the causal model. The code can be seen in the following snippet:

```
identified_ate = causal_mdl.identified_estimand_
print(identified_ate)
drlearner_estimate = causal_mdl.estimate_
print(drlearner_estimate)
```

The code shown in the preceding snippet outputs the **estimand expression** for `identified_estimand_`, which is a derivation of the expected value for $Y \sim W + X \quad Y \sim W + X$, with some assumptions. Then, the causal-realized `estimate_` returns the ATE for treatment #1, as illustrated in the following code snippet:

```
Estimand type: nonparametric-ate
### Estimand : 1
Estimand name: backdoor1 (Default)
Estimand expression:
        d
───────────────(E[_LTV|_RETAIL,_URBAN,_PREMIUM,_RURAL,_CC_LIMIT,
d[_TREATMENT]
])

Estimand assumption 1, Unconfoundedness: If U→{_TREATMENT} and U→_LTV then \
P(_LTV|_TREATMENT,_RETAIL,_URBAN,_PREMIUM,_RURAL,_CC_LIMIT,_spend,_ppm,_tpm,U)
= \ P(_LTV|_TREATMENT,_RETAIL,_URBAN,_PREMIUM,_RURAL,_CC_LIMIT,_spend,_ppm,_
tpm)

*** Causal Estimate ***
## Identified estimand
Estimand type: nonparametric-ate

## Realized estimand
b:_LTV ~ _TREATMENT + _RETAIL + _URBAN + _PREMIUM + _RURAL + \ _CC_LIMIT + _
```

```
spend + _ppm + _tpm | _CC_LIMIT
Target units:

## Estimate
Mean value: 7227.904763676559
Effect estimates: [6766.07978487 7337.39526574 7363.36013004
                   7224.20893104 7500.84310705 7221.40328496]
```

Next, we can iterate across all treatments in the causal model and return a summary for each treatment, like this:

```
for i in range(causal_mdl._d_t[0]):
    print("Treatment: %s" % treatment_names[i])
    display(econml_mdl.summary(T=i+1))
```

The preceding code outputs three linear regression summaries. The first one looks like this:

Treatment: Lower Credit Limit

Coefficient Results						
	point_estimate	**stderr**	**zstat**	**pvalue**	**ci_lower**	**ci_upper**
_CC_LIMIT	0.006	0.02	0.322	0.747	-0.032	0.045

CATE Intercept Results						
	point_estimate	**stderr**	**zstat**	**pvalue**	**ci_lower**	**ci_upper**
cate_intercept	6514.633	1312.662	4.963	0.0	3941.863	9087.404

A linear parametric conditional average treatment effect (CATE) model was fitted:

$$Y = \Theta(X) \cdot T + g(X,W) + \epsilon$$

where T is the one-hot-encoding of the discrete treatment and for every outcome i and treatment j the CATE $\Theta_{ij}(X)$ has the form:

Figure 11.13: Summary of one of the treatments

To get a better sense of the coefficients and intercepts, we can plot them with their respective confidence intervals. To do this, we first create an index of treatments (`idxs`). There are three treatments, so this is just an array of numbers between 0 and 2. Then, place all the coefficients (`coef_`) and intercepts (`intercept_`) into an array using list comprehension. However, it's a bit more complicated for the 90% confidence intervals for both coefficients and intercepts because `coef__interval` and `intercept__interval` return the lower and upper bounds of these intervals. We need the length of the margin of error in both directions, not the bounds. We deduct the coefficient and intercepts from these bounds to obtain their respective margin of error, as illustrated in the following code snippet:

```python
idxs = np.arange(0, causal_mdl._d_t[0])
coefs = np.hstack([causal_mdl.coef_(T=i+1) for i in idxs])
intercepts = np.hstack(
    [causal_mdl.intercept_(T=i+1)for i in idxs]
)
coefs_err = np.hstack(
    [causal_mdl.coef__interval(T=i+1) for i in idxs]
)
coefs_err[0, :] = coefs - coefs_err[0, :]
coefs_err[1, :] = coefs_err[1, :] - coefs
intercepts_err = np.vstack(
    [causal_mdl.intercept__interval(T=i+1) for i in idxs]
).Tintercepts_err[0, :] = intercepts - intercepts_err[0, :]
intercepts_err[1, :] = intercepts_err[1, :] - intercepts
```

Next, we plot the coefficients for each treatment and respective errors using errorbar. We can do the same with the intercepts as another subplot, as illustrated in the following code snippet:

```python
ax1 = plt.subplot(2, 1, 1)
plt.errorbar(idxs, coefs, coefs_err, fmt="o")
plt.xticks(idxs, treatment_names)
plt.setp(ax1.get_xticklabels(), visible=False)
plt.title("Coefficients")
plt.subplot(2, 1, 2)
plt.errorbar(idxs, intercepts, intercepts_err, fmt="o")
plt.xticks(idxs, treatment_names)
plt.title("Intercepts")
```

The preceding snippet outputs the following:
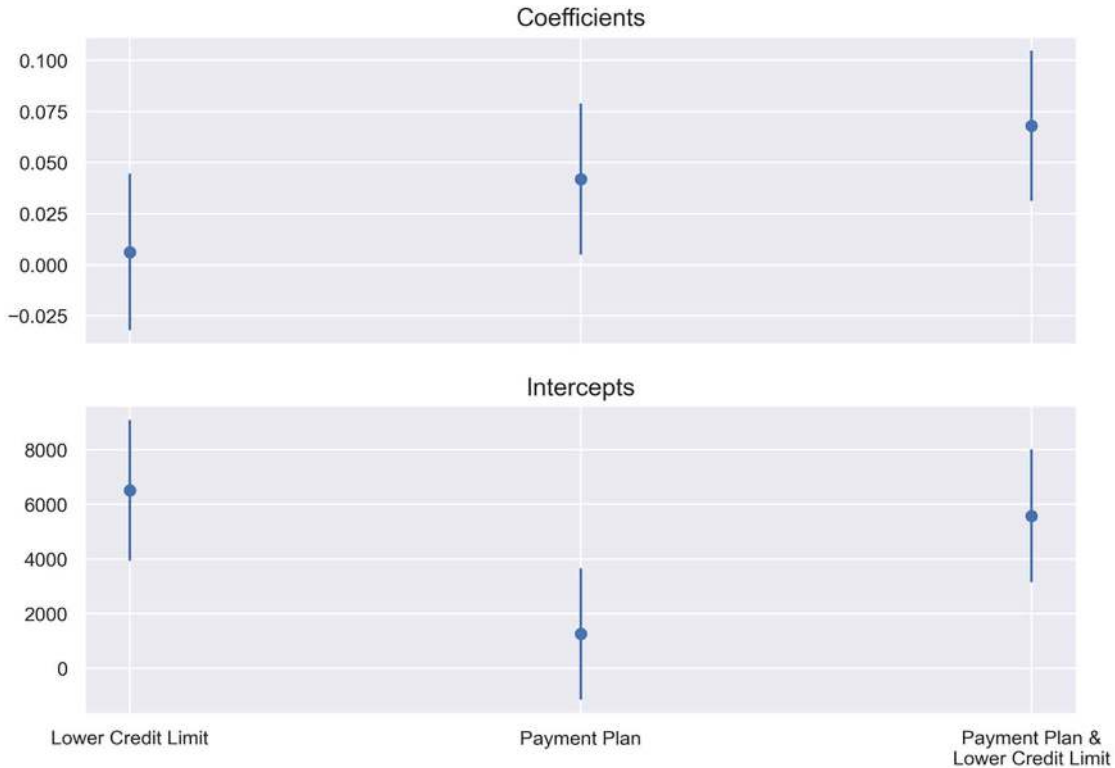
Figure 11.14: Coefficients and intercepts for all treatments

With *Figure 11.14*, you can appreciate how relatively large the margin of error is for all intercepts and coefficients. Nonetheless, it's pretty clear that from the coefficients alone, treatments keep getting marginally better when read from left to right. But before we conclude that **Payment Plan & Lower Credit Limit** is the best policy, we must consider the intercept, which is lower for this treatment than the first one. Essentially, this means that a customer with a minimal credit card limit is likely to improve lifetime value more with the first policy because the coefficients are multiplied by the limit, whereas the intercept is the starting point. Given that there's no one best policy for all customers, let's examine how to choose policies for each, using the causal model.

Choosing policies

We can decide on a credit policy on a customer basis using the `const_marginal_effect` method, which takes the X effect modifier (`_CC_LIMIT`) and computes the counterfactual CATE, $\theta(X)$. In other words, it returns the estimated `_LTV` for all treatments for all observations in X.

However, they don't all cost the same. Setting up a payment plan requires administrative and legal costs of about *NT$1,000* per contract, and according to the bank's actuarial department, lowering the credit limit by 25 has an opportunity cost estimated at *NT$72* per average payment per month (`_ppm`) over the lifetime of the customer. To factor these costs, we can set up a simple `lambda` function that takes the payment plan costs for all treatments and adds them to the variable credit limit costs, which, naturally, is multiplied by `_ppm`. Given an array with credit card limits of n length, the cost function returns an array of (n, 3) dimensions with a cost for each treatment. Then, we obtain the counter-factual CATE and deduct the costs (`treatment_effect_minus_costs`). Then, we expand the array to include a column of zeros representing the **None** treatment and use `argmax` to return each customer's recommended treatment index (`recommended_T`), as illustrated in the following code snippet:

```python
cost_fn = lambda X: np.repeat(
    np.array([[0, 1000, 1000]]),
    X.shape[0], axis=0) + (np.repeat(np.array([[72, 0, 72]]),
    X.shape[0], axis=0)
    *X._ppm.values.reshape(-1,1)
)
treatment_effect_minus_costs = causal_mdl.const_marginal_effect(
    X=X.values) - cost_fn(ccdefault_causal_df)
treatment_effect_minus_costs = np.hstack(
    [
        np.zeros(X.shape),
        treatment_effect_minus_costs
    ]
)
recommended_T = np.argmax(treatment_effect_minus_costs, axis=1)
```

We can use `scatterplot` `_CC_LIMIT` and `_ppm`, color-coded by the recommended treatment to observe the customer's optimal credit policy, as follows:

```python
sns.scatterplot(
    x=ccdefault_causal_df['_CC_LIMIT'].values,
    y=ccdefault_causal_df['_ppm'].values,
    hue=all_treatment_names[recommended_T],
    hue_order=all_treatment_names
)
plt.title("Optimal Credit Policy by Customer")
plt.xlabel("Original Credit Limit")
plt.ylabel("Payments/month")
```

The preceding snippet outputs the following scatterplot:
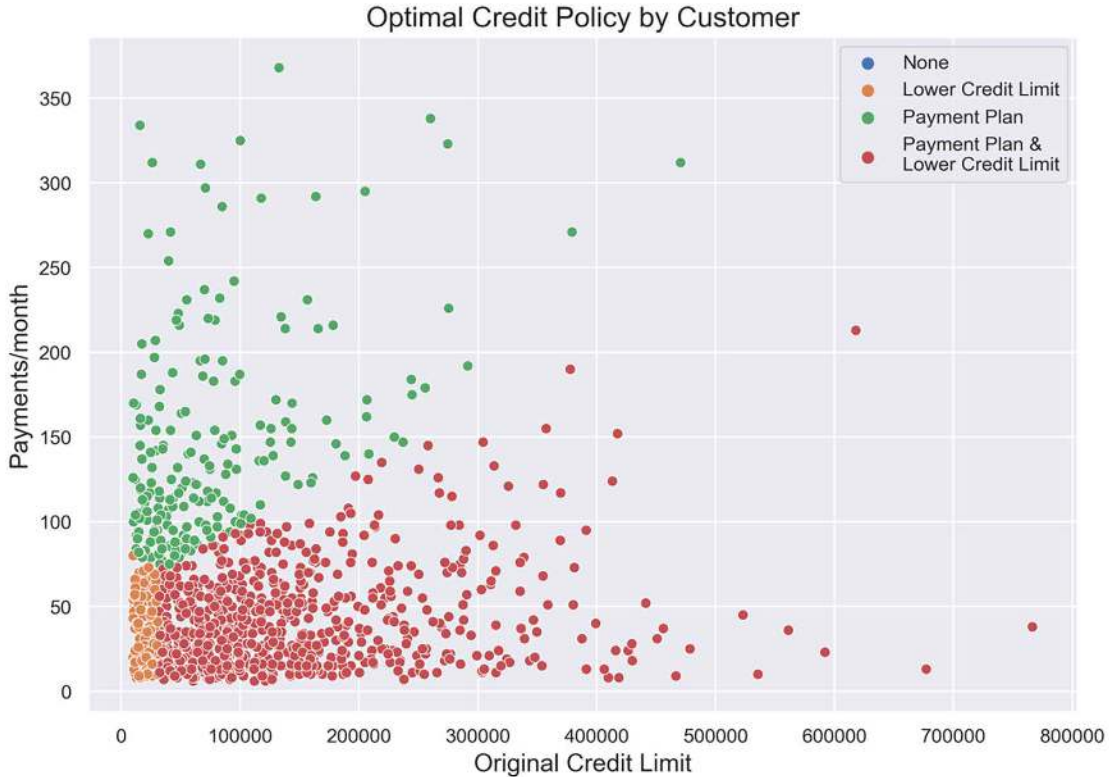
Figure 11.15: Optimal credit policy by customer depending on original credit limit and card usage

It's evident in *Figure 11.15* that "None" (no treatment) is never recommended for any customer. This fact holds even when costs aren't deducted—you can remove `cost_fn` from `treatment_effect_minus_costs` and rerun the code that outputs the plot to verify that treatment is always prescribed regardless of the costs. You can deduce that all treatments are beneficial to customers, some more than others. And, of course, some treatments benefit the bank more than others, depending on the customer. There's a thin line to tread here.

One of the biggest concerns is fairness to customers, especially those that the bank wronged the most: the underprivileged age group. Just because one policy is more costly to the bank than another, it should not preclude the opportunity to access other policies. One way to assess this would be with a percentage-stacked bar plot for all recommended policies. That way, we can observe how the recommended policy is split between privileged and underprivileged groups. Have a look at the following code snippet:

```
ccdefault_causal_df['recommended_T'] = recommended_T
plot_df = ccdefault_causal_df.groupby(
    ['recommended_T','AGE_GROUP']).size().reset_index()
plot_df['AGE_GROUP'] = plot_df.AGE_GROUP.replace(
```

```
      {0:'underprivileged', 1:'privileged'}
)
plot_df = plot_df.pivot(
    columns='AGE_GROUP',
    index='recommended_T',
    values=0
)
plot_df.index = treatment_names
plot_df = plot_df.apply(lambda r: r/r.sum()*100, axis=1)
plot_df.plot.bar(stacked=True, rot=0)
plt.xlabel('Optimal Policy')
plt.ylabel('%')
```

The code in the preceding snippet outputs the following:
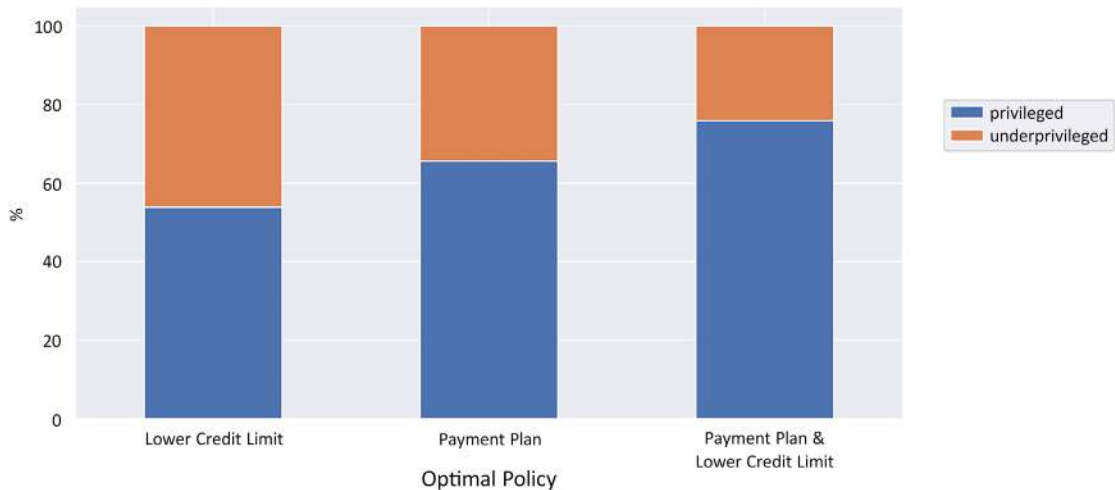
Figure 11.16: Fairness of optimal policy distributions

Figure 11.16 shows how privileged groups are at a higher proportion assigned one of the policies with the **Payment Plan.** This disparity is primarily due to the bank's costs being a factor, so if the bank were to absorb some of these costs, it could make it fairer. But what would be a fair solution? Choosing credit policies is an example of procedural fairness, and there are many possible definitions. Does equal treatment literally mean equal treatment or proportional treatment? Does it encompass notions of freedom of choice too? What if a customer prefers one policy over another? Should they be allowed to switch? Whatever the definition is, it can be resolved with help from the causal model. We can assign all customers the same policy, or the distribution of recommended policies can be calibrated so that proportions are equal, or every customer can choose between the first and second most optimal policy. There are so many ways to go about it!

Testing estimate robustness

The dowhy library comes with four methods to test the robustness of the estimated causal effect, outlined as follows:

- **Random common cause**: Adding a randomly generated confounder. If the estimate is robust, the ATE should not change too much.

- **Placebo treatment refuter**: Replacing treatments with random variables (placebos). If the estimate is robust, the ATE should be close to zero.

- **Data subset refuter**: Removing a random subset of the data. If the estimator generalizes well, the ATE should not change too much.

- **Add unobserved common cause**: Adding an unobserved confounder that is associated with both the treatment and outcome. The estimator assumes some level of unconfoundedness but adding more should bias the estimates. Depending on the strength of the confounder's effect, it should have an equal impact on the ATE.

We will test robustness with the first two next.

Adding a random common cause

This method is the easiest to implement by calling `refute_estimate` with `method_name="random_common_cause"`. This will return a summary that you can print. Have a look at the following code snippet:

```
ref_random = causal_mdl.refute_estimate(
    method_name="random_common_cause"
)
print(ref_random)
```

The code in the preceding snippet outputs the following:

```
Refute: Add a Random Common Cause
Estimated effect:7227.904763676559
New effect:7241.433599647397
```

The preceding output tells us that a new common cause, or *W* variable, doesn't have a sizable impact on the ATE.

Replacing the treatment variable with a random variable

With this method, we will replace the treatment variable with noise. If the treatment correlates robustly with the outcome, this should bring the average effect to zero. To implement it, we also call the `refute_estimate` function but with `placebo_treatment_refuter` for the method. We must also specify the `placebo_type` and the number of simulations (`num_simulations`). The placebo type we will use is `permute`, and the more simulations the better, but this will also take longer. The code can be seen in the following snippet:

```
ref_placebo = causal_mdl.refute_estimate(
    method_name="placebo_treatment_refuter",
    placebo_type="permute", num_simulations=20
)
print(ref_placebo)
```

The preceding code outputs the following:

```
Refute: Use a Placebo Treatment
Estimated effect:7227.904763676559
New effect:381.05420029741083
p value:0.32491556283289624
```

As you can tell by the preceding output, the new effect is close to zero. However, given that the p-value is above 0.05, we cannot reject the null hypothesis that ascertains that the ATE is greater than zero. This tells us that the estimated causal effect is not very robust. We can likely improve it by adding relevant confounders or by using a different causal model, but also, the experimental design had flaws that we cannot fix, such as the biased way the bank prescribed the treatments according to the risk factor.

Mission accomplished

The mission of this chapter was twofold, as outlined here:

- Create a fair predictive model to predict which customers are most likely to default.
- Create a robust causal model to estimate which policies are most beneficial to customers and the bank.

Regarding the first goal, we have produced four models with bias mitigation methods that are objectively fairer than the base model, according to four fairness metrics (SPD, DI, AOD, EOD)—when comparing privileged and underprivileged age groups. However, only two of these models are intersectionally fairer using both age group and gender, according to DFBA (see *Figure 11.7*). We can still improve fairness significantly by combining methods, yet any one of the four models improves the base model.

As for the second goal, the causal inference framework determined that any of the policies tested is better than no policy for both parties. Hooray! However, it yielded estimates that didn't establish a single winning one. Still, as expected, the recommended policy varies according to the customer's credit limit—on the other hand, if we aim to maximize bank profitability, we must factor in the average use of credit cards. The question of profitability presents two goals that we must reconcile: prescribing the recommended policies that benefit either the customer or the bank the most.

For this reason, how to be procedurally fair is a complicated question with many possible answers, and any of the solutions would involve the bank absorbing some of the costs associated with implementing the policies. As for robustness, despite the flawed experiment, we can conclude that our estimates have a mediocre level of robustness, passing one robustness test but not the other. That being said, it all depends on what we consider robust enough to validate our findings. Ideally, we would ask the bank to start a new unbiased experiment but waiting another 6 months might not be feasible.

In data science, we often find ourselves working with flawed experiments and biased data and have to make the most of it. Causal inference provides a way to do so by disentangling cause and effect, complete with estimates and their respective confidence intervals. We can then offer findings with all the disclaimers so that decision-makers can make informed decisions. Biased decisions lead to biased outcomes, so the moral imperative of tackling bias can start by shaping decision-making.

Summary

After reading this chapter, you should understand how bias can be detected visually and with metrics, both in data and models, then mitigated through preprocessing, in-processing, and post-processing methods. We also learned about causal inference by estimating heterogeneous treatment effects, making fair policy decisions with them, and testing their robustness. In the next chapter, we also discuss bias but learn how to tune models to meet several objectives, including fairness.

Dataset sources

Yeh, I. C., & Lien, C. H. (2009). *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients*. Expert Systems with Applications, 36(2), 2473-2480: `https://dl.acm.org/doi/abs/10.1016/j.eswa.2007.12.020`

Further reading

- Chang, C., Chang, H.H., and Tien, J., 2017, *A Study on the Coping Strategy of Financial Supervisory Organization under Information Asymmetry: Case Study of Taiwan's Credit Card Market*. Universal Journal of Management, 5, 429-436: `http://doi.org/10.13189/ujm.2017.050903`

- Foulds, J., and Pan, S., 2020, *An Intersectional Definition of Fairness*. 2020 IEEE 36th International Conference on Data Engineering (ICDE), 1918-1921: `https://arxiv.org/abs/1807.08362`

- Kamiran, F., and Calders, T., 2011, *Data preprocessing techniques for classification without discrimination*. Knowledge and Information Systems, 33, 1-33: `https://link.springer.com/article/10.1007/s10115-011-0463-8`

- Feldman, M., Friedler, S., Moeller, J., Scheidegger, C., and Venkatasubramanian, S., 2015, *Certifying and Removing DI*. Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining: `https://arxiv.org/abs/1412.3756`

- Kamishima, T., Akaho, S., Asoh, H., and Sakuma, J., 2012, *Fairness-Aware Classifier with Prejudice Remover Regularizer*. ECML/PKDD: `https://dl.acm.org/doi/10.5555/3120007.3120011`

- A. Agarwal, A. Beygelzimer, M. Dudik, J. Langford, and H. Wallach, *A Reductions Approach to Fair Classification*, International Conference on Machine Learning, 2018. `https://arxiv.org/pdf/1803.02453.pdf`

- Kearns, M., Neel, S., Roth, A., and Wu, Z., 2018, *Preventing Fairness Gerrymandering: Auditing and Learning for Subgroup Fairness*. ICML: `https://arxiv.org/pdf/1711.05144.pdf`

- Pleiss, G., Raghavan, M., Wu, F., Kleinberg, J., and Weinberger, K.Q., 2017, *On Fairness and Calibration*. NIPS: `https://arxiv.org/abs/1709.02012`

- Foster, D. and Syrgkanis, V., 2019, *Orthogonal Statistical Learning*. ICML: `http://arxiv.org/abs/1901.09036`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

12

Monotonic Constraints and Model Tuning for Interpretability

Most model classes have hyperparameters that can be tuned for faster execution speed, increasing predictive performance, and reducing overfitting. One way of reducing overfitting is by introducing regularization into the model training. In *Chapter 3*, *Interpretation Challenges*, we called regularization a remedial interpretability property, which reduces complexity with a penalty or limitation that forces the model to learn sparser representations of the inputs. Regularized models generalize better, which is why it is highly recommended to tune models with regularization to avoid overfitting to the training data. As a side effect, regularized models tend to have fewer features and interactions, making the model easier to interpret—*less noise means a clearer signal*!

And even though there are many hyperparameters, we will only focus on those that improve interpretability by controlling overfitting. Also, to a certain extent, we will revisit bias mitigation through the class imbalance-related hyperparameters explored in previous chapters.

Chapter 2, *Key Concepts of Interpretability*, explained three model properties that impact interpretability: non-linearity, interactivity, and non-monotonicity. Left to its own devices, a model can learn some spurious and counterintuitive non-linearities and interactivities. As discussed in *Chapter 10*, *Feature Selection and Engineering for Interpretability*, guardrails can be placed to prevent this through careful feature engineering. However, what can we do to place guardrails for monotonicity? In this chapter, we will learn how to do just this with monotonic constraints. And just as monotonic constraints can be the model counterpart to feature engineering, regularization can be the model counterpart to the feature selection methods we covered in *Chapter 10*!

These are the main topics we are going to cover in this chapter:

- Placing guardrails with feature engineering
- Tuning models for interpretability
- Implementing model constraints

Technical requirements

This chapter's example uses the `mldatasets`, `pandas`, `numpy`, `sklearn`, `xgboost`, `lightgbm`, `catboost`, `tensorflow`, `bayes_opt`, `tensorflow_lattice`, `matplotlib`, `seaborn`, `scipy`, `xai`, and `shap` libraries. Instructions on how to install these libraries are in the preface.

> The code for this chapter is located here: `https://packt.link/pKeAh`

The mission

The issue of algorithmic fairness is one with massive social implications, from the allocation of welfare resources to the prioritization of life-saving surgeries to screening job applications. These machine learning algorithms can determine a person's livelihood or life, and it's often the most marginalized and vulnerable populations that get the worst treatment from these algorithms because they perpetuate systemic biases learned from the data. Therefore, it's poorer families that get misclassified for child abuse; it's racial-minority people who get underprioritized for medical treatment; and it's women who get screened out of high-paying tech jobs. Even in cases involving less immediate and individualized risks such as online searches, Twitter/X bots, and social media profiles, societal prejudices such as elitism, racism, sexism, and ageism are reinforced.

This chapter will continue on the mission from *Chapter 6*, *Anchors and Counterfactual Explanations*. If you aren't familiar with these techniques, please go back and read *Chapter 6* to get a solid understanding of the problem. The recidivism case from *Chapter 6* is one of algorithmic bias. The co-founder of the company that developed the **COMPAS algorithm** (where **COMPAS** stands for **Correctional Offender Management Profiling Alternative Sanctions**) admitted that it's tough to make a score without questions that are correlated with race. This correlation is one of the main reasons that scores are biased against African Americans. The other reason is the likely overrepresentation of black defendants in the training data. We don't know for sure because we don't have the original training data, but we know that non-white minorities are overrepresented in the population of incarcerated individuals. We also know that black people are typically overrepresented in arrests because of codified discrimination in terms of minor drug-related offenses and over-policing in black communities.

So, what can we do to fix it?

In *Chapter 6*, *Anchors and Counterfactual Explanations*, we managed to demonstrate via a *proxy model* that the COMPAS algorithm was biased. For this chapter, let's say that the journalist published your findings, and an algorithmic justice advocacy group read the article and reached out. Companies that make criminal assessment tools are not taking responsibility for bias and claim that their tools simply reflect *reality*. The advocacy group has hired you to demonstrate that a machine learning model can be trained to be significantly less biased toward black defendants while ensuring that the model reflects only proven criminal justice *realities*.

These proven realities include the monotone decrease of recidivism risk with age, and a strong correlation with priors, which increases strongly with age. Another fact supported by the academic literature is how females are significantly less prone to recidivism and criminality in general.

Before we move on, we must recognize that supervised learning models face several impediments in capturing domain knowledge from data. For instance, consider the following:

- **Sample**, **exclusion**, **or prejudice bias:** What if your data doesn't truly represent the environment your model intends to generalize? If that's the case, the domain knowledge won't align with what you observe in the data. What if the environment that produced the data has a built-in systemic or institutional bias? Then, the data will reflect these biases.

- **Class imbalance:** As seen in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*, class imbalance could favor some groups over others. While taking the most effective route toward high accuracy, a model will learn from this imbalance, contradicting domain knowledge.

- **Non-monotonicity:** Sparse areas in a features histogram or high-leverage outliers could cause a model to learn non-monotonicity when domain knowledge calls for otherwise, and any of the previously mentioned problems could contribute to this as well.

- **Uninfluential features:** An unregularized model will, by default, try to learn from all features as long as they carry some information, but this stands in the way of learning from relevant features or overfitting to noise in the training data. A more parsimonious model is more likely to prop up features supported by domain knowledge.

- **Counterintuitive interactions:** As mentioned in *Chapter 10*, *Feature Selection and Engineering for Interpretability*, there could be counterintuitive interactions that a model favors over domain knowledge-supported interactions. As a side effect, these could end up favoring some groups that correlate with them. And in *Chapter 6, Anchors and Counterfactual Explanations*, we saw proof of this through an understanding of double standards.

- **Exceptions:** Our domain knowledge facts are based on an aggregate understanding, but when looking for patterns on a more granular scale, models will find exceptions such as pockets where female recidivism is of higher risk than that of males. Known phenomena might not support these models but they could be valid nonetheless, so we must be careful not to erase them with our tuning efforts.

The advocacy group has validated the data as adequately representative of only one county in Florida, and they have provided you with a balanced dataset. The first impediment is a tough one to ascertain and control. The second one has been taken care of. It's now up to you to deal with the remaining four!

The approach

You have decided to take a three-fold approach, as follows:

- **Placing guardrails with feature engineering:** Leveraging lessons learned in *Chapter 6*, *Anchors and Counterfactual Explanations*, as well as the domain knowledge we already have about priors and age, in particular, we will engineer some features.

- **Tuning models for interpretability:** Once the data is ready, we will tune many models with different class weighting and overfitting prevention techniques. These methods will ensure that the models not only generalize better but are also easier to interpret.

- **Implementing model constraints:** Last but not least, we will implement monotonic and interaction constraints on the best models to make sure that they don't stray from trusted and fair interactions.

In the last two sections, we will make sure the models perform accurately and fairly. We will also compare recidivism risk distributions between the data and the model to ensure that they align.

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/blob/main/12/Recidivism_part2.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `pandas` and `numpy` to manipulate it
- `sklearn` (scikit-learn), `xgboost`, `lightgbm`, `catboost`, `tensorflow`, `bayes_opt`, and `tensorflow_lattice` to split the data and fit the models
- `matplotlib`, `seaborn`, `scipy`, `xai`, and `shap` to visualize the interpretations

You should load all of them first, as follows:

```python
import math
import os
import copy
import mldatasets
import pandas as pd
import numpy as np
from sklearn import preprocessing, model_selection, metrics,\
    linear_model, svm, neural_network, ensemble
import xgboost as xgb
import lightgbm as lgb
import catboost as cb
import tensorflow as tf
from bayes_opt import BayesianOptimization
import tensorflow_lattice as tfl
from tensorflow.keras.wrappers.scikit_learn import\
                                         KerasClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import scipy
import xai
import shap
```

Let's check that `tensorflow` has loaded the right version with `print(tf.__version__)`. This should be 2.8 and above.

Understanding and preparing the data

We load the data like this into a DataFrame we call `recidivism_df`:

```
recidivism_df = mldatasets.load("recidivism-risk-balanced")
```

There should be over 11,000 records and 11 columns. We can verify this was the case with `info()`, as follows:

```
recidivism_df.info()
```

The preceding code outputs the following:

```
RangeIndex: 11142 entries, 0 to 11141
Data columns (total 12 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   sex                    11142 non-null  object
 1   age                    11142 non-null  int64
 2   race                   11142 non-null  object
 3   juv_fel_count          11142 non-null  int64
 4   juv_misd_count         11142 non-null  int64
 5   juv_other_count        11142 non-null  int64
 6   priors_count           11142 non-null  int64
 7   c_charge_degree        11142 non-null  object
 8   days_b_screening_arrest  11142 non-null  float64
 9   length_of_stay         11142 non-null  float64
 10  compas_score           11142 non-null  int64
 11  is_recid               11142 non-null  int64
dtypes: float64(2), int64(7), object(3)
```

The output checks out. There are no missing values, and all but three features are numeric (`sex`, `race`, and `charge_degree`). This is the same data we used in *Chapter 6*, *Anchors and Counterfactual Explanations*, so the data dictionary is exactly the same. However, the dataset has been balanced with sampling methods, and, this time, it hasn't been prepared for us so we will need to do this, but before this, let's gain an understanding of what the balancing did.

Verifying the sampling balance

We can check how `race` and `is_recid` are distributed with XAI's `imbalance_plot`. In other words, it will tally how many records exist for each `race-is_recid` combination. This plot will allow us to observe if there are imbalances in the number of defendants that recidivate for each `race`. The code can be seen in the following snippet:

```
categorical_cols_l = [
    'sex', 'race', 'c_charge_degree', 'is_recid', 'compas_score'
]
xai.imbalance_plot(
    recidivism_df,
    'race',
    'is_recid',
    categorical_cols=categorical_cols_l
)
```

The preceding code outputs *Figure 12.1*, which depicts how all races have equal amounts of `is_recid=0` and `is_recid=1`. However, **Other** is not at parity in numbers with the other races. Incidentally, this version of the dataset has bucketed all other races as **Other**, and the choice to not `upsample` **Other** or `downsample` the other two races to achieve total parity is made because they are less represented in the defendant population. This balancing choice is one of many that can be done in a situation such as this. Demographically, it all depends on what your data is supposed to represent. Defendants? Inmates? Civilians in the general population? And at what level? Of the county? The state? The country?

The output can be seen here:
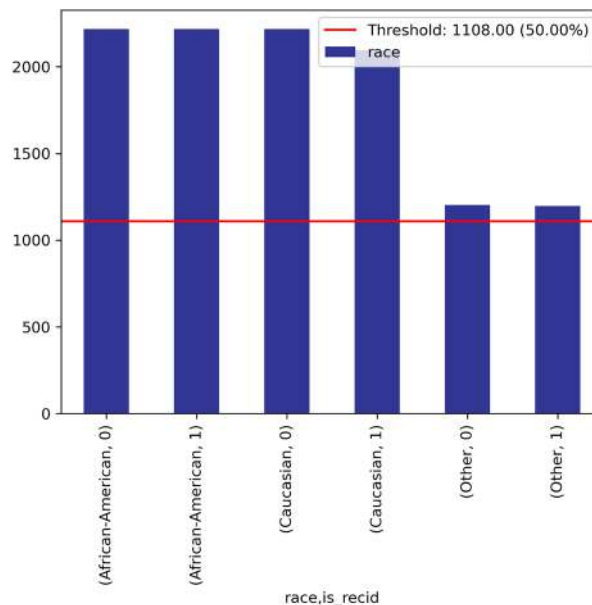
Figure 12.1: Distribution of 2-year recidivism (is_recid) by ethnicity

Next, let's compute how well each of our features monotonically correlates to the target. Spearman's rank-order correlation will be instrumental in this chapter because it measures the monotonicity between two features. After all, one of the technical topics of this chapter is monotonic constraints, and the primary mission is to produce a significantly less biased model.

We first create a new DataFrame without `compas_score` (`recidivism_corr_df`). Using this DataFrame, we output a color-coded DataFrame with a `feature` column with the first 10 features' names and another one with the Spearman coefficient (`correlation_to_target`) for all 10 features toward the 11th—the target variable. The code can be seen in the following snippet:

```python
recidivism_corr_df = recidivism_df.drop(
    ['compas_score'], axis=1
)
pd.DataFrame(
    {'feature': recidivism_corr_df.columns[:-1],
     'correlation_to_target':\
        scipy.stats.spearmanr(recidivism_corr_df).\
        correlation[10,:-1]
    }
).style.background_gradient(cmap='coolwarm')
```

The preceding code outputs the DataFrame shown in *Figure 12.2*. The most correlated features are `priors_count` followed by `age`, the three juvenile counts, and `sex`. The coefficients for `c_charge_degree`, `days_b_screening_arrest`, `length_of_stay`, and `race` are negligible.

The output can be seen here:

feature	correlation_to_target
sex	0.093255
age	-0.155838
race	-0.004598
juv_fel_count	0.082138
juv_misd_count	0.117976
juv_other_count	0.125797
priors_count	0.283640
c_charge_degree	-0.037764
days_b_screening_arrest	0.032485
length_of_stay	0.012530

Figure 12.2: Spearman coefficients of all features toward the target, prior to feature engineering

Next, we will learn how to use feature engineering to "bake in" some domain knowledge into the features.

Placing guardrails with feature engineering

In *Chapter 6, Anchors and Counterfactual Explanations,* we learned that besides `race`, the features most prominent in our explanations were age, `priors_count`, and `c_charge_degree`. Thankfully, the data is now balanced, so the racial bias attributed to this imbalance is now gone. However, through anchors and counterfactual explanations, we found some troubling inconsistencies. In the case of `age` and `priors_count`, these inconsistencies were due to how those features were distributed. We can correct issues with distribution through feature engineering, and, that way, ensure that a model doesn't learn from uneven distributions. In `c_charge_degree`'s case, being categorical, it lacked a discernible order, and this lack of order created unintuitive explanations.

In this section, we will study **ordinalization**, **discretization**, and **interaction terms**, three ways in which you can place guardrails through feature engineering.

Ordinalization

Let's first take a look in the following code snippet at how many observations we have for every `c_charge_degree` category:

```
recidivism_df.c_charge_degree.value_counts()
```

The preceding code produced the following output:

```
(F3)    6555
(M1)    2632
(F2)     857
(M2)     768
(F1)     131
(F7)     104
(MO3)     76
(F5)       7
(F6)       5
(NI0)      4
(CO3)      2
(TCX)      1
```

Each of the charge degrees corresponds to the charge's gravity. There's an order to these gravities, which is lost by using a categorical feature. We can easily fix this by replacing each category with a corresponding order.

We can put a lot of thought into what this order should be. For instance, we could look at sentencing laws or guidelines—there are minimum or maximum years of prison enforced for different degrees. We could also look at statistics on how violent these people are on average and assign this information to the charge degree. There's potential for bias in every decision such as this, and if we don't have substantial evidence to support it, it's best to use a sequence of integers. So, that's what we are going to do now. We will create a dictionary (`charge_degree_code_rank`) that maps the degrees to a number corresponding to a rank of gravity, from low to high. Then, we can use the pandas `replace` function to use the dictionary to perform the replacements. The code can be seen in the following snippet:

```
charge_degree_code_rank = {
    '(F10)': 15, '(F9)':14, '(F8)':13,\
    '(F7)':12, '(TCX)':11, '(F6)':10, '(F5)':9,\
    '(F4)':8, '(F3)':7, '(F2)':6, '(F1)':5, '(M1)':4,\
    '(NI0)':4, '(M2)':3, '(CO3)':2, '(MO3)':1, '(X)':0
}
recidivism_df.c_charge_degree.replace(
    charge_degree_code_rank, inplace=True
)
```

One way to assess how this order corresponds to recidivism probability is through a line plot that shows how it changes as the charge degree increases. We can use a function called `plot_prob_progression` for this, which takes a continuous feature in the first argument (`c_charge_degree`) to measure against probability for a binary feature in the second (`is_recid`). It can split the continuous feature by intervals (`x_intervals`), and even use quantiles (`use_quantiles`). Lastly, you can define axis labels and titles. The code can be seen in the following snippet:

```
mldatasets.plot_prob_progression(
    recidivism_df.c_charge_degree,
    recidivism_df.is_recid, x_intervals=12,
    use_quantiles=False,
    xlabel='Relative Charge Degree',
    title='Probability of Recidivism by Relative Charge Degree'
)
```

The preceding code generates the plot in *Figure 12.3*. As the now-ranked charge degree increases, the tendency is that the probability of 2-year recidivism decreases, except for rank 1. Below the probability, there are bar charts that show the distribution of the observations over every rank. Because it is so unevenly distributed, you should take the tendency with a grain of salt. You'll notice that some ranks, such as 0, 8, and 13–15, aren't in the plot because the charge-degree categories existed in the criminal justice system but weren't in the data.

The output can be seen here:
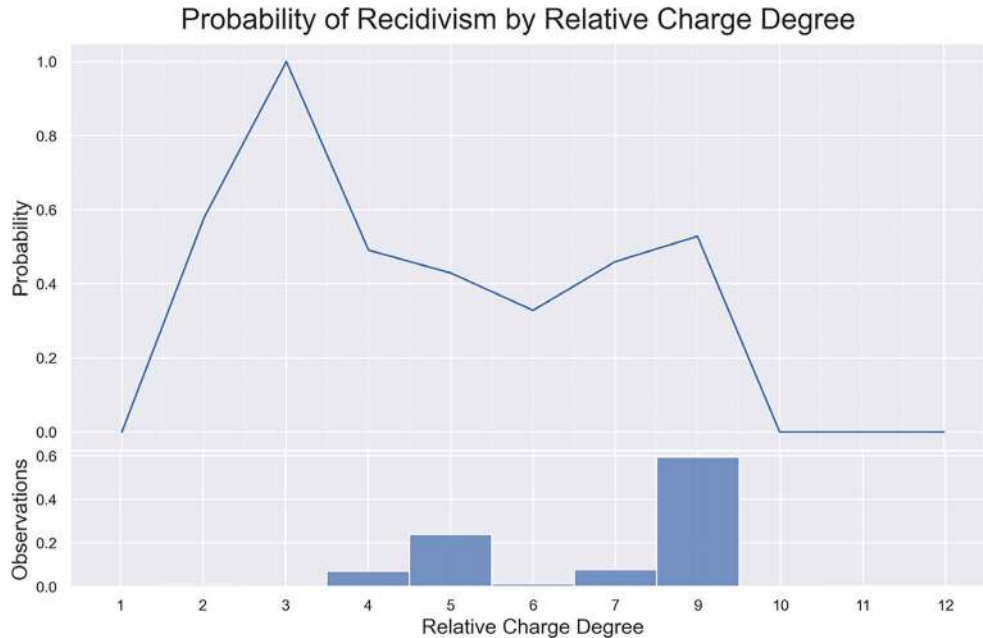
Figure 12.3: Probability progression plot by charge degree

Feature engineering-wise, we can't do much more to improve `c_charge_degree` because it already represents discrete categories now enhanced with an order. Any further transformations could produce a significant loss of information unless we have evidence to suggest otherwise. On the other hand, continuous features inherently have an order; however, a problem may arise from the level of precision they carry because small differences may not be meaningful but the data may tell the model otherwise. Uneven distributions and counterintuitive interactions only exacerbate this problem.

Discretization

To understand how to discretize our `age` continuous feature best, let's try two different approaches. We can use equal-sized discretization, also known as fixed-width bins or intervals, which means the size of the bin is determined by $(\max(x) - \min(x))/N$, where N is the number of bins. Another way to do this is with equal-frequency discretization, also known as quantiles, which ensures that each bin has approximately the same number of observations. Although, sometimes, given the histogram's skewed nature, it may be impossible to split them N ways, so you may end up with *N-1* or *N-2* quantiles.

It is easy to compare both approaches with `plot_prob_progression`, but this time, we produce two plots, one with fixed-width bins (`use_quantiles=False`) and another with quantiles (`use_quantiles=True`). The code can be seen in the following snippet:

```
mldatasets.plot_prob_progression(
    recidivism_df.age,
    recidivism_df.is_recid,
```

```
        x_intervals=7,
        use_quantiles=False,
        title='Probability of Recidivism by Age Discretized in Fix-Width \
        Bins',
        xlabel='Age'
)
mldatasets.plot_prob_progression(
        recidivism_df.age,
        recidivism_df.is_recid,
        x_intervals=7, use_quantiles=True,
        title='Probability of Recidivism by Age Discretized \
        in Quantiles',
        xlabel='Age'
)
```

The preceding snippet outputs *Figure 12.4*. By looking at the **Observations** portion of the fixed-width bin plot, you can tell that the histogram for the age feature is right-skewed, which causes the probability to shoot up for the last bin. The reason for this is that some outliers exist in this bin. On the other hand, the fixed-frequency (quantile) plot histogram is more even, and probability consistently decreases. In other words, it's monotonic—as it should be, according to our domain knowledge on the subject.

The output can be seen here:
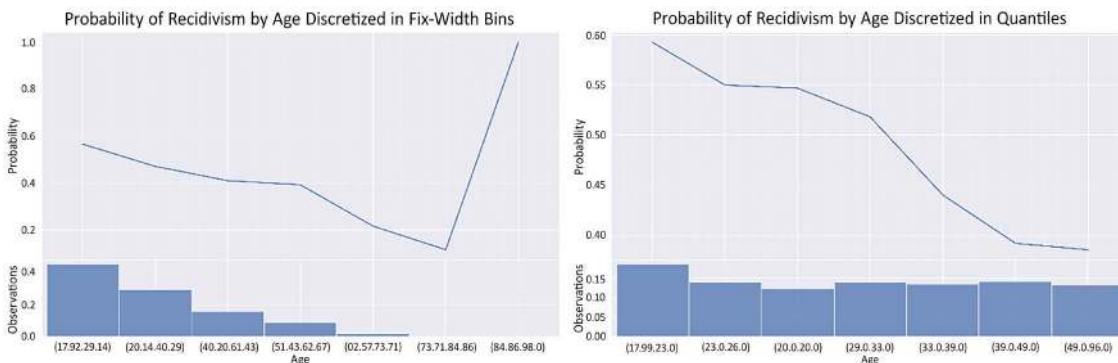
Figure 12.4: Comparing two discretization approaches for age

It is easy to observe why using quantiles to bin the feature is a better approach. We can take age and engineer a new feature called age_group. The qcut pandas function can perform quantile-based discretization. The code can be seen in the following snippet:

```
recidivism_df['age_group'] = pd.qcut(
        recidivism_df.age, 7, precision=0
).astype(str)
```

So, we now have discretized `age` into `age_group`. However, it must be noted that many model classes discretize automatically, so why bother? Because it allows you to control its effects. Otherwise, the model might decide on bins that don't ensure monotonicity. For instance, maybe the model might always use 10 quantiles whenever possible. Still, if you attempt this level of granularity on `age` (`x_intervals=10`), you'll end up with spikes in the probability progression. Our goal was to make sure that the models would learn that `age` and the incidence of `is_recid` have a monotonic relationship, and we cannot ascertain this if we allow the model to choose bins that may or may not achieve the same goal.

We will remove `age` because `age_group` has everything we need. But wait—you ask—won't we lose some important information by removing this variable? Yes, but only because of its interaction with `priors_count`. So, before we drop any features, let's examine this relationship and realize how, through creating an interaction term, we can retain some of the information lost through the removal of `age`, while keeping the interaction.

Interaction terms and non-linear transformations

We already know from *Chapter 6*, *Anchors and Counterfactual Explanations*, that `age` and `priors_count` are two of the most important predictors, and we can observe how, together, they impact the incidence of recidivism (`is_recid`) with `plot_prob_contour_map`. This function produces contour lines with color-coded contour regions, signifying different magnitudes. They are useful in topography, where they show elevation heights. In machine learning, they can show a two-dimensional plane representing feature interaction with a metric. In this case, the dimensions are `age` and `priors_count`, and the metric is the incidence of recidivism. The arguments received by this function are the same as `plot_prob_progression` except that it takes two features corresponding to the *x* axis and *y* axis. The code can be seen in the following snippet:

```
mldatasets.plot_prob_contour_map(
    recidivism_df.age,
    recidivism_df.priors_count,
    recidivism_df.is_recid,
    use_quantiles=True,
    xlabel='Age',
    ylabel='Priors Count',
    title='Probability of Recidivism by Age/Priors Discretized in \
    Quantiles'
)
```

The preceding snippet generated *Figure 12.5*, which shows how, when discretized by quantiles, the probability of 2-year recidivism increases, the lower the `age` and the higher the `priors_count`. It also shows histograms for both features. `priors_count` is very right-skewed, so discretization is challenging, and the contour map does not offer a perfectly diagonal progression between the bottom right and top left. And if this plot looks familiar, it's because it's just like the partial dependence interaction plots we produced in *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, except it's not measured against the predictions of a model but the ground truth (`is_recid`). We must distinguish between what the data can tell us directly and what the model has learned from it.

The output can be seen here:
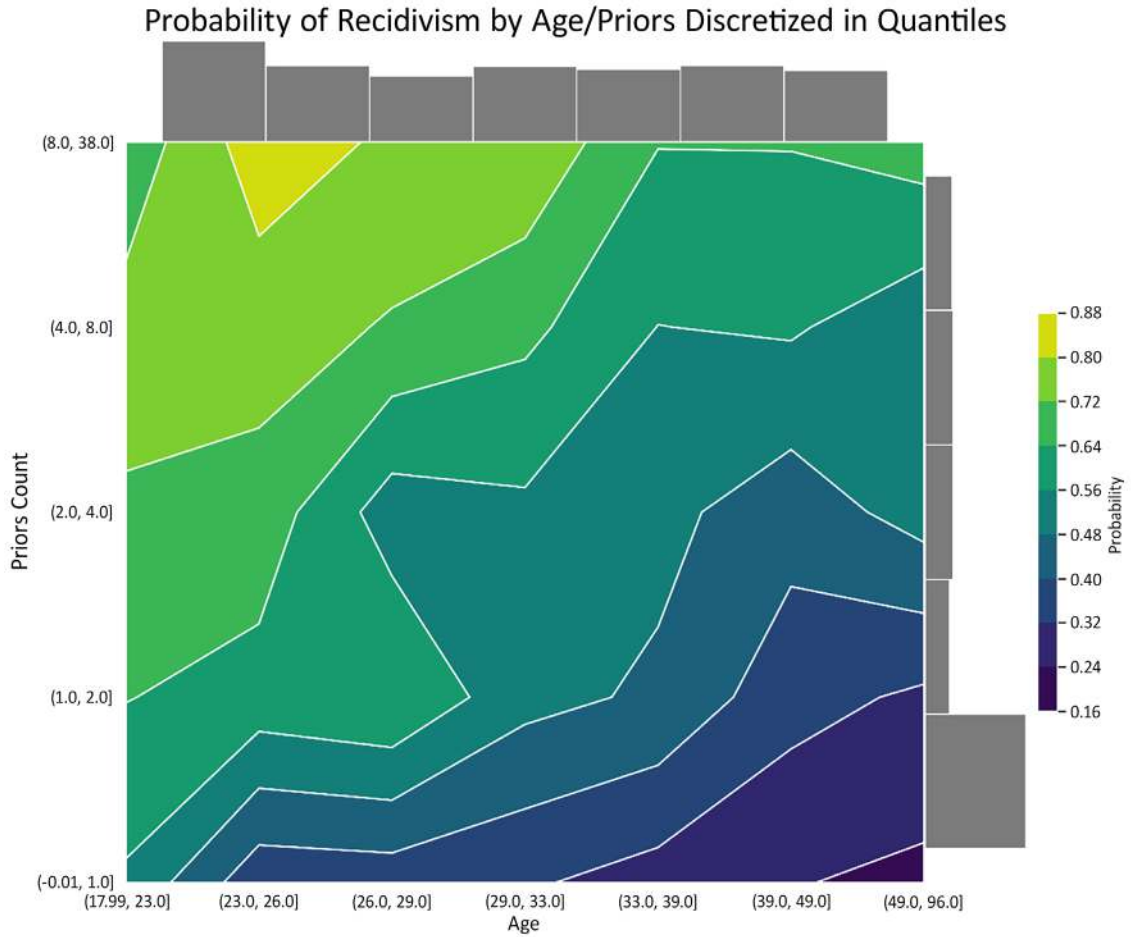
Figure 12.5: Recidivism probability contour map for age and priors_count

We can now engineer an interaction term that includes both features. Even though the contour map discretized the features to observe a smoother progression, we do not need to discretize this relationship. What makes sense is to make it a ratio of `priors_count` per year. But years since when? Years since the defendants were an adult, of course. But to obtain the years, we cannot use `age - 18` because this would lead to zero division, so we will use `17` instead. There are, of course, many ways to do this. The best way would be if we hypothetically had ages with decimals, and by deducting 18, we could compute a very precise `priors_per_year` ratio. Still, unfortunately, we don't have that. You can see the code in the following snippet:

```
recidivism_df['priors_per_year'] =\
            recidivism_df['priors_count']/(recidivism_df['age'] - 17)
```

Black-box models typically find interaction terms automatically. For instance, hidden layers in a neural network have all the first-order interactions, but because of the non-linear activations, it is not limited to linear combinations. However, "manually" defining interaction terms and even non-linear transformation allows us to interpret these better once the model has been fitted. Furthermore, we can also use monotonic constraints on them, precisely what we will do later with `priors_per_year`. For now, let's examine if its monotonicity holds with `plot_prob_progression`. Have a look at the following code snippet:

```
mldatasets.plot_prob_progression(
    recidivism_df.priors_per_year,
    recidivism_df.is_recid,
    x_intervals=8,
    xlabel='Priors Per Year',
    title='Probability of Recidivism by Priors per Year (\
according to data)'
)
```

The preceding snippet outputs the progression in the following screenshot, which shows how the new feature is almost monotonic:
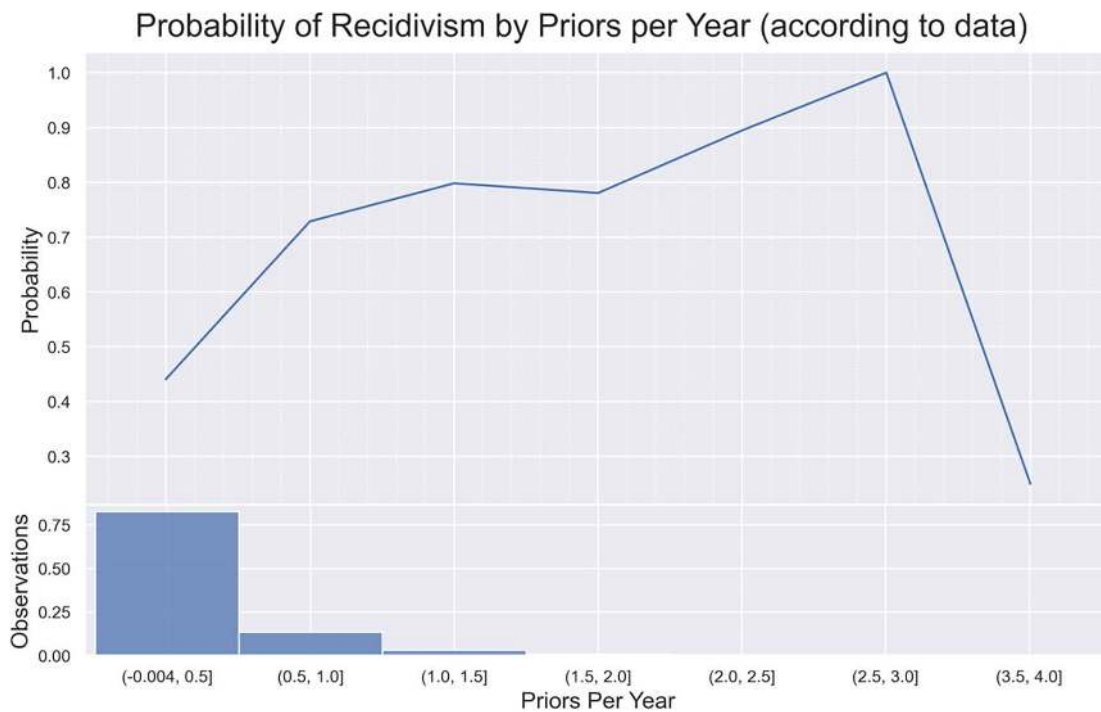
Figure 12.6: Probability progression for priors_per_year

The reason `priors_per_year` isn't more monotonic is how sparse the over-3.0 `priors_per_year` interval is. It would therefore be very unfair to these few defendants to enforce monotonicity on this feature because they present a 75% risk dip. One way to tackle this is to shift them over to the left, by setting `priors_per_year = -1` for these observations, as illustrated in the following code snippet:

```
recidivism_df.loc[recidivism_df.priors_per_year > 3,\
                  'priors_per_year'] = -1
```

Of course, this shift changes the interpretation of the feature ever so slightly, knowing that the few values of -1 really mean over 3. Now, let's generate another contour map, but this time, between `age_group` and `priors_per_year`. The latter will be discretized in quantiles (`y_intervals=6, use_quantiles=True`) so that the probability of recidivism is more easily observed. The code is shown in the following snippet:

```
mldatasets.plot_prob_contour_map(
    recidivism_df.age_group,
    recidivism_df.priors_per_year,
    recidivism_df.is_recid,
    y_intervals=6,
    use_quantiles=True,
    xlabel='Age Group',
    title='Probability of Recidivism by Age/Priors per Year \
    Discretized in Quantiles', ylabel='Priors Per Year'
)
```

The preceding snippet generates the contours in *Figure 12.7*. It shows that, for the most part, the plot moves in one direction. We were hoping to achieve this outcome because it allows us, through one interaction feature, to control the monotonicity of what used to involve two features.

The output can be seen here:
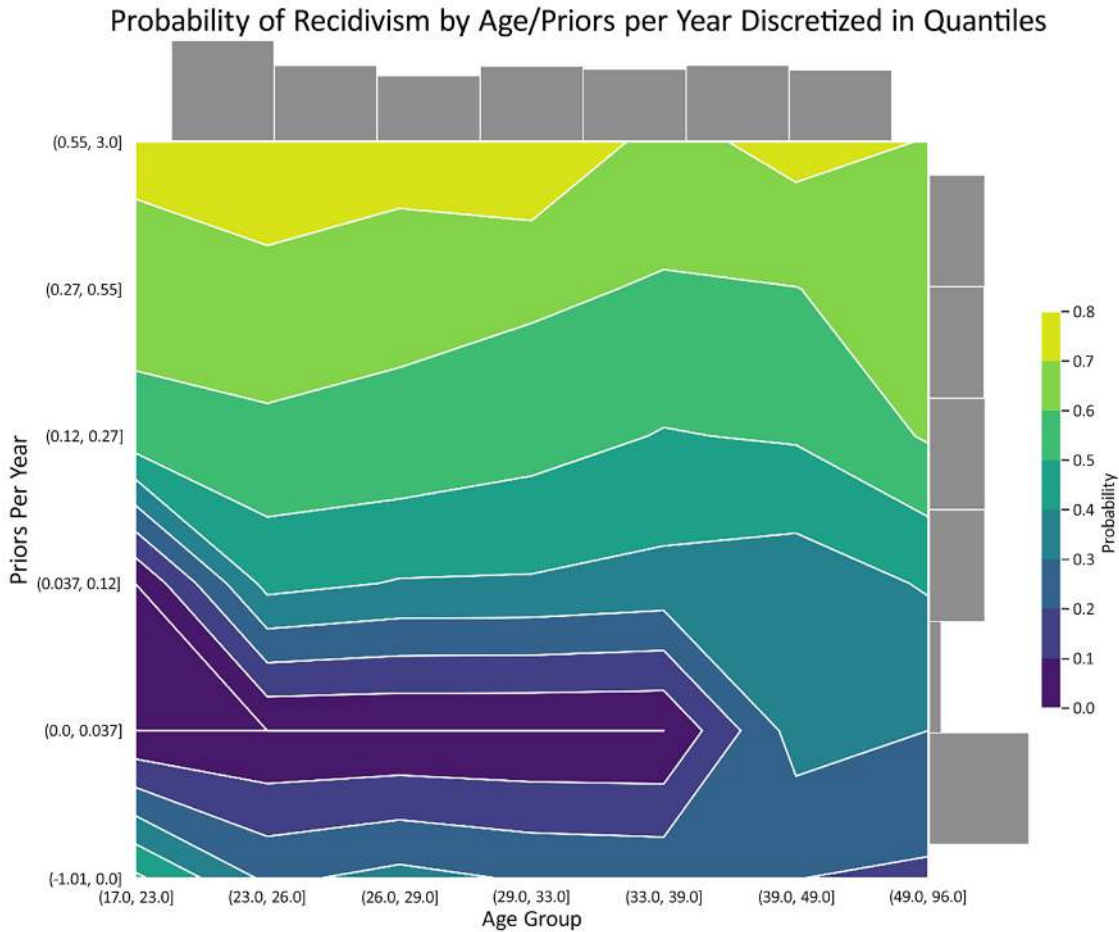
Figure 12.7: Recidivism probability contour map for age_group and priors_per_year

Almost everything is ready, but `age_group` is still categorical, so we have to encode it to take a numerical form.

Categorical encoding

The best categorical encoding method for `age_group` is **ordinal encoding**, also known as **label encoding**, because it will retain its order. We should also encode the other two categorical features in the dataset, `sex` and `race`. For `sex`, ordinal encoding converts it into binary form—equivalent to **dummy encoding**. On the other hand, `race` is a tougher call because it has three categories, and using ordinal encoding could lead to bias. However, whether to use **one-hot encoding** instead depends on which model classes you are using. Tree-based models have no bias issues with ordinal features but other models that operate with weights on a feature basis, such as neural networks and logistic regression, could be biased by this order.

Considering that the dataset has been balanced on `race`, there's a lower risk of this happening and we will remove this feature later anyway, so we will go ahead and ordinal-encode it.

To ordinal-encode the three features, we will use scikit-learn's `OrdinalEncoder`. We can use its `fit_transform` function to fit and transform the features in one fell swoop. Then, we should also delete unnecessary features while we are at it. Have a look at the following code snippet:

```python
cat_feat_l = ['sex', 'race', 'age_group']
ordenc = preprocessing.OrdinalEncoder(dtype=np.int8)
recidivism_df[cat_feat_l] =\
                    ordenc.fit_transform(recidivism_df[cat_feat_l])
recidivism_df.drop(['age', 'priors_count', 'compas_score'],\
                        axis=1, inplace=True)
```

Now, we aren't entirely done yet. We still ought to initialize our random seeds and train/test split our data.

Other preparations

The next preparations are straightforward. To ensure reproducibility, let's set a random seed everywhere it is needed, then set our y as `is_recid` and X as every other feature. We perform `train_test_split` on those two. Lastly, we reconstruct the `recidivism_df` DataFrame with the X followed by the y. The only reason for this is so that `is_recid` is the last column, which will help with the next step. The code can be seen here:

```python
rand = 9
os.environ['PYTHONHASHSEED'] = str(rand)
tf.random.set_seed(rand)
np.random.seed(rand)
y = recidivism_df['is_recid']
X = recidivism_df.drop(['is_recid'], axis=1).copy()
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size=0.2, random_state=rand
)
recidivism_df = X.join(y)
```

We will now verify that Spearman's correlations have improved where needed and stay the same otherwise. Have a look at the following code snippet:

```python
pd.DataFrame(
    {
        'feature': X.columns,
        'correlation_to_target':scipy.stats.spearmanr(recidivism_df).\
        correlation[10,:-1]
    }
).style.background_gradient(cmap='coolwarm')
```

The preceding code outputs the DataFrame shown in *Figure 12.8*. Please compare it with *Figure 12.2*. Note that discretized in quantiles, `age` is slightly less monotonically correlated with the target. Once ordinalized, `c_charge_degree` is also much more correlated, and `priors_per_year` has also improved over `priors_count`. No other features should have been affected, including those that have the lowest coefficients.

The output can be seen here:

feature	correlation_to_target
sex	0.093255
race	-0.004598
juv_fel_count	0.082138
juv_misd_count	0.117976
juv_other_count	0.125797
c_charge_degree	0.069803
days_b_screening_arrest	0.032485
length_of_stay	0.012530
age_group	-0.152131
priors_per_year	0.321885

Figure 12.8: Spearman correlation coefficients of all features toward the target (after feature engineering)

Features with the lowest coefficients are likely also unnecessary in a model, but we will let the model decide if they are useful through regularization. That's what we will do next.

Tuning models for interpretability

Traditionally, regularization was only achieved by imposing penalty terms such as **L1**, **L2**, or **elastic net** on the coefficients or weights, which shrink the impact of the least relevant features. As seen in the *Embedded methods* section of *Chapter 10*, *Feature Selection and Engineering for Interpretability*, this form of regularization results in feature selection while also reducing overfitting. And this brings us to another broader definition of regularization, which does not require a penalty term. Often, this comes as imposing a limitation, or a stopping criterion that forces the model to curb its complexity.

In addition to regularization, both in its narrow (penalty-based) and broad sense (overfitting methods), there are other methods that tune a model for interpretability—that is, improve the fairness, accountability, and transparency of a model through adjustments to the training process. For instance, the class imbalance hyperparameters we discussed in *Chapter 10*, *Feature Selection and Engineering for Interpretability*, and the adversarial debiasing in *Chapter 11*, *Bias Mitigation and Causal Inference Methods*, enhance fairness. Also, the constraints we will study further in this chapter have potential benefits for fairness, accountability, and transparency.

There are so many different tuning possibilities and model classes. As stated at the beginning of the chapter, we will focus on interpretability-related options, but will also limit the model classes to a popular deep learning library (Keras), a handful of popular tree ensembles (XGBoost, Random Forest, and so on), **Support Vector Machines** (**SVMs**), and logistic regression. Except for the last one, these are all considered black-box models.

Tuning a Keras neural network

For a Keras model, we will choose the best regularization parameters through hyperparameter tuning and **stratified K-fold cross-validation**. We will do this using the following steps:

1. First, we need to define the model and the parameters to tune.
2. Then, we run the tuning.
3. Next, we examine its results.
4. Finally, we extract the best model and evaluate its predictive performance.

Let's look at each of these steps in detail.

Defining the model and parameters to tune

The first thing we ought to do is create a function (`build_nn_mdl`) to build and compile a regularizable Keras model. The function takes arguments that will help tune it. It takes a tuple with the number of neurons in hidden layers (`hidden_layer_sizes`), and a value of L1 (`l1_reg`) and L2 (`l1_reg`) regularization to apply on the layer's kernel. Lastly, it takes the `dropout` parameter, which, unlike L1 and L2 penalties, is a **stochastic regularization method** because it employs random selection. Have a look at the following code snippet:

```python
def build_nn_mdl(hidden_layer_sizes, l1_reg=0, l2_reg=0, dropout=0):
    nn_model = tf.keras.Sequential([
        tf.keras.Input(shape=[len(X_train.keys())]),\
        tf.keras.layers.experimental.preprocessing.Normalization()
    ])
    reg_args = {}
    if (l1_reg > 0) or (l2_reg > 0):
        reg_args = {'kernel_regularizer':\
                    tf.keras.regularizers.l1_l2(l1=l1_reg, l2=l2_reg)}
    for hidden_layer_size in hidden_layer_sizes:
        nn_model.add(tf.keras.layers.Dense(hidden_layer_size,\
                        activation='relu', **reg_args))
    if dropout > 0:
        nn_model.add(tf.keras.layers.Dropout(dropout))
    nn_model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
    nn_model.compile(
        loss='binary_crossentropy',
        optimizer=tf.keras.optimizers.Adam(lr=0.0004),
```

```
            metrics=['accuracy',tf.keras.metrics.AUC(name='auc')]
    )

        return nn_model
```

The previous function initializes the model (nn_model) as a Sequential model with an input layer that corresponds to the number of features in training data, and a Normalization() layer that standardizes the input. Then, if either penalty term is over zero, it will set a dictionary (reg_args) with the kernel_regularizer assigned to tf.keras.regularizers.l1_l2 initialized with these penalties. Once it adds the hidden (Dense) layers with the corresponding hidden_layer_size, it will pass the reg_args dictionary as extra arguments to each layer. After all hidden layers have been added, it will optionally add the Dropout layer and the final Dense layer with the sigmoid activation for the output. The model is then compiled with binary_crossentropy and an Adam optimizer with a slow learning rate and is set to monitor accuracy and auc metrics.

Running the hyperparameter tuning

Now that we have defined the model and parameters to tune, we initialize the RepeatedStratifiedKFold cross-validator, which splits (n_splits) the training data in five a total of three times (n_repeats), using different randomization in each repetition. We then create a grid (nn_grid) for the grid-search hyperparameter tuning. It's testing only two possible options for three of the parameters (l1_reg, l2_reg, and dropout), which will result in $2^3 = 8$ combinations. We will use a scikit-learn wrapper (KerasClassifier) for our model to be compatible with the scikit-learn grid search. Speaking of which, we next initialize GridSearchCV, which, using the Keras model (estimator), performs a cross-validated (cv) grid search (param_grid). We want it to choose the best parameters based on precision (scoring) and not raise errors in the process (error_score=0). Finally, we fit GridSearchCV as we would with any Keras model, passing X_train, y_train, epochs, and batch_size. The code can be seen in the following snippet:

```
cv = model_selection.RepeatedStratifiedKFold(
    n_splits=5,
    n_repeats=3,
    random_state=rand
)
nn_grid = {
    'hidden_layer_sizes':[(80,)],
    'l1_reg':[0,0.005],
    'l2_reg':[0,0.01],
    'dropout':[0,0.05]
}
nn_model = KerasClassifier(build_fn=build_nn_mdl)
nn_grid_search = model_selection.GridSearchCV(
```

```
    estimator=nn_model,
    cv=cv,
    n_jobs=-1,
    param_grid=nn_grid,
    scoring='precision',
    error_score=0
)
nn_grid_result = nn_grid_search.fit(
    X_train.astype(float),
    y_train.astype(float),
    epochs=400,batch_size=128
)
```

Next, we can examine the results of our grid search.

Examining the results

Once the grid search has been completed, you can output the best parameters in a dictionary with this command: `print(nn_grid_result.best_params_)`. Or you can place all the results into a DataFrame, sort them by the highest precision (`sort_values`), and output them as follows:

```
pd.DataFrame(nn_grid_result.cv_results_)[
    [
        'param_hidden_layer_sizes',
        'param_l1_reg',
        'param_l2_reg',
        'param_dropout',
        'mean_test_score',
        'std_test_score',
        'rank_test_score'
    ]
].sort_values(by='rank_test_score')
```

The preceding snippet outputs the DataFrame shown in *Figure 12.9*. The unregularized model is dead last, showing that all regularized model combinations performed better. One thing to note is that given the 1.5–2% standard deviations (`std_test_score`) and that the top performer is only 2.2% from the lowest performer, in this case, the benefits are marginal from a precision standpoint, but you should use a regularized model nonetheless because of other benefits.

The output can be seen here:

param_hidden_layer_sizes	param_l1_reg	param_l2_reg	param_dropout	mean_test_score	std_test_score	rank_test_score
(80,)	0.005000	0.010000	0.050000	0.677700	0.018629	1
(80,)	0	0	0	0.670297	0.027577	2
(80,)	0.005000	0	0.050000	0.667625	0.021315	3
(80,)	0	0.010000	0.050000	0.667291	0.022757	4
(80,)	0.005000	0.010000	0	0.665553	0.017141	5
(80,)	0	0	0.050000	0.663555	0.011802	6
(80,)	0.005000	0	0	0.659114	0.026934	7
(80,)	0	0.010000	0	0.649437	0.019827	8

Figure 12.9: Results for cross-validated grid search for a neural net model

Evaluating the best model

Another important element that the grid search produced is the best-performing model (`nn_grid_result.best_estimator_`). We can create a dictionary to store all the models we will fit in this chapter (`fitted_class_mdls`) and then, using `evaluate_class_mdl`, evaluate this regularized Keras model and keep the evaluation in the dictionary at the same time. Have a look at the following code snippet:

```
fitted_class_mdls = {}
fitted_class_mdls['keras_reg'] = mldatasets.evaluate_class_mdl(
    nn_grid_result.best_estimator_,
    X_train.astype(float),
    X_test.astype(float),
    y_train.astype(float),
    y_test.astype(float),
    plot_roc=False,
    plot_conf_matrix=True,
    ret_eval_dict=True
)
```

The preceding snippet produced the confusion matrix and metrics shown in *Figure 12.10*. The accuracy is a little bit better than the original COMPAS model from *Chapter 6, Anchors and Counterfactual Explanations*, but the strategy to optimize for higher precision while regularizing yielded a model with nearly half as many false positives but 50% more false negatives.

The output can be seen here:

```
Accuracy_train:    0.6442      Accuracy_test:    0.6487
Precision_test:    0.7093      Recall_test:      0.5234
ROC-AUC_test:      0.6929      F1_test:          0.6023        MCC_test: 0.3115
```
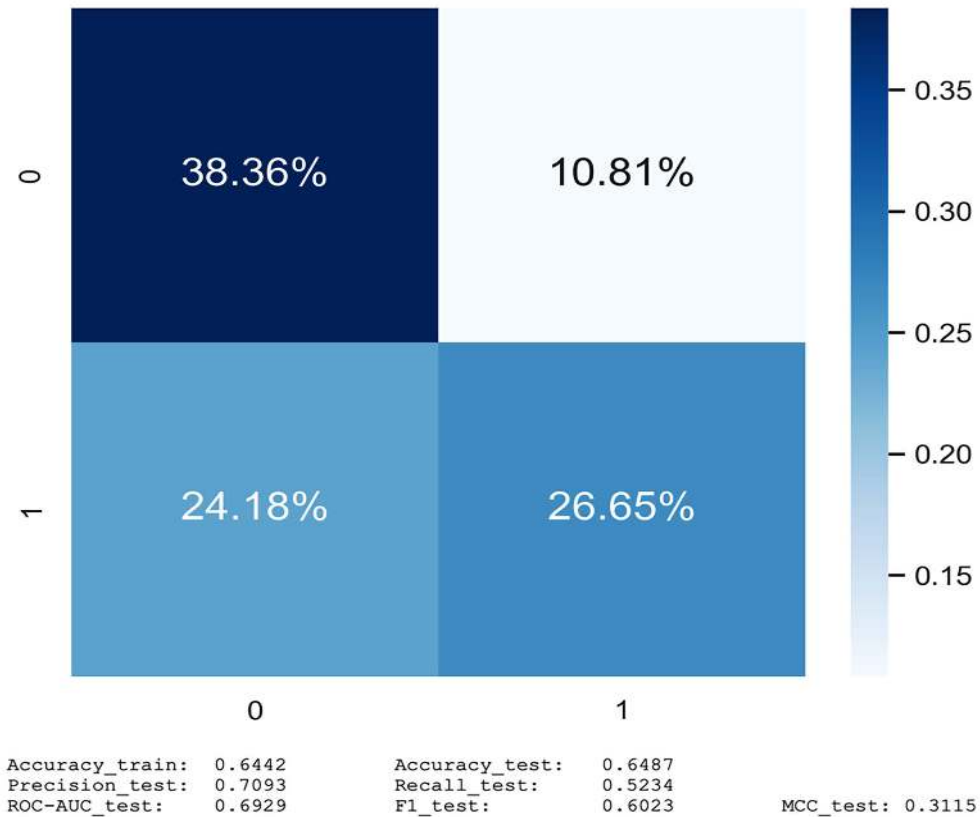
Figure 12.10: Evaluation of the regularized Keras model

Calibrating the class balance can be improved even further by employing a custom loss function or class weights, as we will do later. Next, we will cover how to tune other model classes.

Tuning other popular model classes

In this section, we will fit many different models, both unregularized and regularized. To this end, we will pick from a wide selection of parameters that perform penalized regularization, control overfitting through other means, and account for class imbalance.

A quick introduction to relevant model parameters

For your reference, there are two tables with parameters used to tune many popular models. These have been split into two parts. Part A (*Figure 12.11*) has five scikit-learn models with penalty regularization. Part B (*Figure 12.12*) shows all the tree ensembles, including scikit-learn's Random Forest models and models from the most popular boosted-tree libraries (XGBoost, LightGBM, and CatBoost).

Part A can be seen here:

	LogisticRegression		RidgeClassifier		SVC		NuSVC		MLPClassifier	
			Ridge		SVR		NuSVR		MLPRegressor	
algorithm	solver	"lbfgs"	solver	"auto"	kernel	"rbf"	kernel	"rbf"	solver	"adam"
regularization	penalty	"l2"								
	C	+/- 1	alpha	+/- 1	C	+/- 1	nu	+/- 0.5	alpha	+ 0.0001
	l1_ratio	None			gamma	"scale"	gamma	"scale"		
iterations	max_iter	+/- 100	max_iter	+ None	max_iter	+ -1	max_iter	+ -1	max_iter	+/- 200
learning rate									learning_rate_init	0.001
									learning_rate	"adaptive"
early stopping	tol	- 1e-4	tol	- 1e-3	tol	- 1e-3	tol	- 1e-3	tol	- 1e-4
									n_iter_no_change	- 10
									early_stopping	False
									validation_fraction	0.1
class imbalance	class_weight	None	class_weight	None	class_weight	None	class_weight	None		
sample weight	sample_weight*	None	sample_weight*	None	sample_weight*	None	sample_weight*	None		

(The rows iterations, learning rate, and early stopping are grouped under the label OVERFITTING on the left.)

Figure 12.11: Tuning parameters for penalty-regularized scikit-learn models

In *Figure 12.11*, you can observe models in the columns and corresponding parameter names in the rows with their default values to the right. In between the parameter name and default value, there's a plus or minus sign indicating whether changing the defaults in one direction or another should make the model more conservative. These parameters are also grouped by the following categories:

- **algorithm:** Some training algorithms are less prone to overfitting, but this often depends on the data.
- **regularization:** Only in the stricter sense. In other words, parameters that control penalty-based regularization.
- **iterations:** This controls how many training rounds, iterations, or epochs are performed. Adjusting this in one direction or another can impact overfitting. In tree-based models, the number of estimators or trees is what's analogous.
- **learning rate:** This controls how quickly the learning happens. It works in tandem with iterations. The lower the learning rate, the more iterations are needed to optimize the objective function.
- **early stopping:** These parameters control when to stop the training. This allows you to prevent your model from overfitting to training data.
- **class imbalance:** For most models, this penalizes misclassifications on smaller classes in the loss function, and for tree-based models, in particular, it is used to reweight the splitting criterion. Either way, it only works with classifiers.
- **sample weight:** We leveraged this one in *Chapter 11, Bias Mitigation and Causal Inference Methods*, to assign weights on a sample basis to mitigate bias.

There are both classification and regression models in the headings, and they share the same parameters. Please note that scikit-learn's `LinearRegression` isn't featured under `LogisticRegression` because it doesn't have built-in regularization. In any case, we will use only classification models in this section.

Part B can be seen here:

| | RandomForestClassifier | | XGBRFClassifier | | XGBClassifier | |
	RandomForestRegressor		**XGBRFRegressor**		**XGBRegressor**	
algorithm			booster	"gbtree"	booster	"gbtree"
regularization			reg_lambda	+ 1	reg_lambda	+ 1
			reg_alpha	+ 0	reg_alpha	+ 0
feature sampling	max_features	+/- "auto"			colsample_bytree	- 1
					colsample_bylevel	- 1
					colsample_bynode	- 1
learning rate			eta	+/- 1	eta	+/- 0.3
iterations / # trees	n_estimators	+/- 100	n_estimators	+/- 100	num_round	+/- 100
early stopping	oob_score	+ False	early_stopping_rounds*	None	early_stopping_rounds*	None
			eval_set*	None	eval_set*	None
			eval_metric*	None	eval_metric*	None
tree size	max_depth	- None	max_depth	- 6	max_depth	- 6
	max_leaf_nodes	- None			max_leaves	- 0
	min_samples_leaf	+ 1				
	min_weight_fraction_leaf	+ 0	min_child_weight	+ 1	min_child_weight	+ 1
splitting	min_samples_split	+ 2	gamma	+ 0	gamma	+ 0
	min_impurity_decrease	+ 0				
	criterion	"gini"				
bagging	max_samples	None	subsample	+ 1	subsample	+ 1
	bootstrap	True	sampling_method	"uniform"	sampling_method	"uniform"
class imbalance (classifiers only)	class_weight	None	scale_pos_weight	+/- 1	scale_pos_weight	+/- 1
sample weight	sample_weight*	None	sample_weight*	None	sample_weight*	None
constraints			monotone_constraints	+ None	monotone_constraints	+ None
			interaction_constraints	+ None	interaction_constraints	+ None

The left side rows from **feature sampling** through **bagging** are bracketed under **OVERFITTING**.

| | LGBMClassifier | | CatboostClassifier | |
	LGBMRegressor		**CatboostRegressor**	
algorithm	boosting	"gbdt"		
regularization	lambda_l2	+ 0	l2_leaf_reg	+ 3
	lambda_l1	+ 0		
feature sampling	feature_fraction	- 1		
learning rate	learning_rate	+/- 0.1	learning_rate	+/- 0.03
iterations / # trees	num_iterations	+ 100	iterations	+ 1000
early stopping	early_stopping_rounds*	0	early_stopping_rounds*	False
	eval_set*	None	eval_set*	None
	eval_metric*	None	eval_metric*	None
tree size	max_depth	- -1	depth	- 6
	num_leaves	- 31	max_leaves	- 31
	min_data_in_leaf	+ 20	min_data_in_leaf	+ 1
	min_sum_hessian_in_leaf	+ 1e-3		
splitting	min_split_gain	+ 0	grow_policy	SymmetricTree
			random_strength	+ 1
bagging	bagging_fraction	- 1	subsample	+ 0.66-1
	bagging_freq	+ 0		
class imbalance (classifiers only)	class_weight	None	class_weights	None
	scale_pos_weight	+/- 1	scale_pos_weight	+/- 1
	is_unbalance	+ False	auto_class_weights	+ False
sample weight	sample_weight*	None	sample_weight*	None
constraints	monotone_constraints	+ None	monotone_constraints	+ None
	interaction_constraints	+ None		

The left side rows from **feature sampling** through **bagging** are bracketed under **OVERFITTING**.

Figure 12.12: Tuning parameters for tree-ensemble models

Figure 12.12 is very similar to *Figure 12.11* except that it has a few more parameter categories that are only available in tree ensembles, such as the following:

- **feature sampling:** This works by considering fewer features in node splits, nodes, or tree training. It is a stochastic regularization method because features are randomly selected.
- **tree size:** This constrains the tree either by maximum depth or maximum leaves, or some other parameter that restricts its growth, which, in turn, curbs overfitting.
- **splitting:** Any parameter that controls how nodes in the tree are split can indirectly impact overfitting.
- **bagging:** Also known as **bootstrap aggregating**, this starts by bootstrapping, which involves randomly taking samples from the training data to fit weak learners. This method reduces variance and helps with overfitting, and by extension, the corresponding sampling parameters are usually prominent in hyperparameter tuning.
- **constraints:** We will explain these in further detail in the next section, but this maps how the features should be constrained to decrease or increase against the output. It can reduce overfitting in areas where data is very sparse. However, reducing overfitting is not usually the main goal, while interaction constraints can limit which features are allowed to interact.

Please note that parameters with an asterisk (*) in *Figure 12.12* denote those set in the `fit` function as opposed to those initialized with the model. Also, except for scikit-learn's `RandomForest` models, all other parameters typically have many aliases. For these, we are using the scikit-learn wrapper functions, but all the parameters also exist in the native versions. We can't possibly explain every model parameter here, but it is recommended that you go directly to the documentation for more insight into what each one does. The point of the section was to serve as a guide or reference.

Next, we will take steps similar to what we did with the Keras model but for many different models at once, and, lastly, we will assess the best model for fairness.

Batch hyperparameter tuning models

OK—so, now that we have taken a quick crash course on which levers we can pull to tune the models, let's define a dictionary with all the models, as we've done in other chapters. This time, we have included a `grid` with some parameter values for a grid search. Have a look at the following code snippet:

```
class_mdls = {
    'logistic':{
        'model':linear_model.LogisticRegression(random_state=rand,\
                                                max_iter=1000),
        'grid':{
            'C':np.linspace(0.01, 0.49, 25),
            'class_weight':[{0:6,1:5}],
            'solver':['lbfgs', 'liblinear', 'newton-cg']
        }
    },
    'svc':{
```

```python
        'model':svm.SVC(probability=True, random_state=rand),
        'grid':{'C':[15,25,40], 'class_weight':[{0:6,1:5}]}
    },
    'nu-svc':{
        'model':svm.NuSVC(
            probability=True,
            random_state=rand
        ),
        'grid':{
            'nu':[0.2,0.3], 'gamma':[0.6,0.7],\
            'class_weight':[{0:6,1:5}]}
    },
    'mlp':{
        'model':neural_network.MLPClassifier(
            random_state=rand,
            hidden_layer_sizes=(80,),
            early_stopping=True
        ),
        'grid':{
            'alpha':np.linspace(0.05, 0.15, 11),
            'activation':['relu','tanh','logistic']}
    },
    'rf':{
        'model':ensemble.RandomForestClassifier(
            random_state=rand, max_depth=7, oob_score=True, \
            bootstrap=True
        ),
        'grid':{
            'max_features':[6,7,8],
            'max_samples':[0.75,0.9,1],
            'class_weight':[{0:6,1:5}]}
    },
    'xgb-rf':{
        'model':xgb.XGBRFClassifier(
            seed=rand, eta=1, max_depth=7, n_estimators=200
        ),
        'grid':{
            'scale_pos_weight':[0.85],
            'reg_lambda':[1,1.5,2],
            'reg_alpha':[0,0.5,0.75,1]}
    },
```

```python
    'xgb':{
        'model':xgb.XGBClassifier(
            seed=rand, eta=1, max_depth=7
        ),
        'grid':{
            'scale_pos_weight':[0.7],
            'reg_lambda':[1,1.5,2],
            'reg_alpha':[0.5,0.75,1]}
    },
    'lgbm':{
        'model':lgb.LGBMClassifier(
            random_seed=rand,
            learning_rate=0.7,
            max_depth=5
        ),
        'grid':{
            'lambda_l2':[0,0.5,1],
            'lambda_l1':[0,0.5,1],
            'scale_pos_weight':[0.8]}
    },
    'catboost':{
        'model':cb.CatBoostClassifier(
            random_seed=rand,
            depth=5,
            learning_rate=0.5,
            verbose=0
        ),
        'grid':{
            'l2_leaf_reg':[2,2.5,3],
            'scale_pos_weight':[0.65]}
    }
}
```

The next step is to add a `for` loop to every model in the dictionary, then `deepcopy` it and `fit` it to produce a "base" unregularized model. Next, we produce an evaluation for it with `evaluate_class_mdl` and save it into the `fitted_class_mdls` dictionary we had previously created for the Keras model. Now, we need to produce the regularized version of the model. So, we do another `deepcopy` and follow the same steps we took with Keras to do the `RepeatedStratifiedKFold` cross-validated grid search with `GridSearchCV`, and we also evaluate in the same way, saving the results in the fitted model dictionary. The code is shown in the following snippet:

```python
for mdl_name in class_mdls:
    base_mdl = copy.deepcopy(class_mdls[mdl_name]['model'])
```

```
        base_mdl = base_mdl.fit(X_train, y_train)
        fitted_class_mdls[mdl_name+'_base'] = \
            mldatasets.evaluate_class_mdl(
                base_mdl, X_train, X_test,y_train, y_test,
                plot_roc=False, plot_conf_matrix=False,
                show_summary=False, ret_eval_dict=True
        )
        reg_mdl = copy.deepcopy(class_mdls[mdl_name]['model'])
        grid = class_mdls[mdl_name]['grid']
        cv = model_selection.RepeatedStratifiedKFold(
            n_splits=5, n_repeats=3, random_state=rand
        )
        grid_search = model_selection.GridSearchCV(
        estimator=reg_mdl, cv=cv, param_grid=grid,
        scoring='precision', n_jobs=-1, error_score=0, verbose=0
        )
        grid_result = grid_search.fit(X_train, y_train)
        fitted_class_mdls[mdl_name+'_reg'] =\
            mldatasets.evaluate_class_mdl(
                grid_result.best_estimator_, X_train, X_test, y_train,
                y_test, plot_roc=False,
                plot_conf_matrix=False, show_summary=False,
                ret_eval_dict=True
        )
        fitted_class_mdls[mdl_name+'_reg']['cv_best_params'] =\
            grid_result.best_params_
```

Once the code has finished, we can rank models by precision.

Evaluating models by precision

We can extract the fitted model dictionary's metrics and place them into a DataFrame with `from_dict`. We can then sort the models by their highest test precision and color code the two columns that matter the most, which are `precision_test` and `recall_test`. The code can be seen in the following snippet:

```
class_metrics = pd.DataFrame.from_dict(fitted_class_mdls, 'index')[
    [
        'accuracy_train',
        'accuracy_test',
        'precision_train',
        'precision_test',
        'recall_train',
        'recall_test',
        'roc-auc_test',
```

```
        'f1_test',
        'mcc_test'
    ]
]
with pd.option_context('display.precision', 3):
    html = class_metrics.sort_values(
        by='precision_test', ascending=False
    ).style.background_gradient(
        cmap='plasma',subset=['precision_test']
    ).background_gradient(
        cmap='viridis', subset=['recall_test'])
html
```

The preceding code will output the DataFrame shown in *Figure 12.13*. You can tell that regularized tree-ensemble models mostly rule the ranks, followed by their unregularized counterparts. The one exception is regularized Nu-SVC, which is number one, and its unregularized version is dead last!

The output can be seen here:

	accuracy_train	accuracy_test	precision_train	precision_test	recall_train	recall_test	roc-auc_test	f1_test	mcc_test
catboost_reg	0.968	0.826	0.991	0.836	0.943	0.818	0.885	0.827	0.652
nu-svc_reg	0.939	0.807	0.950	0.836	0.925	0.772	0.858	0.803	0.616
catboost_base	0.969	0.814	0.978	0.805	0.959	0.837	0.878	0.821	0.629
lgbm_reg	0.863	0.766	0.904	0.800	0.807	0.718	0.826	0.757	0.535
xgb_reg	0.807	0.727	0.885	0.800	0.697	0.618	0.811	0.697	0.469
lgbm_base	0.855	0.752	0.865	0.758	0.836	0.753	0.822	0.755	0.504
xgb_base	0.801	0.739	0.802	0.749	0.789	0.733	0.811	0.741	0.479
logistic_reg	0.643	0.638	0.721	0.745	0.445	0.437	0.701	0.551	0.309
svc_reg	0.623	0.648	0.741	0.728	0.356	0.491	0.716	0.586	0.317
mlp_reg	0.649	0.653	0.690	0.724	0.518	0.514	0.706	0.601	0.324
rf_reg	0.722	0.686	0.741	0.716	0.668	0.635	0.759	0.673	0.376
logistic_base	0.651	0.654	0.685	0.714	0.535	0.533	0.701	0.610	0.322
:	:	:	:	:	:	:	:	:	:
nu-svc_base	0.531	0.509	0.560	0.580	0.204	0.122	0.579	0.201	0.049

Figure 12.13: Top models according to the cross-validated grid search

You will find that the Keras regularized neural network model has lower precision than regularized logistic regression, but higher recall. It's true that we want to optimize for high precision because it impacts false positives, which we want to minimize, but precision can be at 100% and recall at 0%, and if that's the case, your model is no good. At the same time, there's fairness, which is about having a low false-positive rate but being equally distributed across races. So, there's a balancing act, and chasing one metric won't get us there.

Assessing fairness for the highest-performing model

To determine how to proceed, we must first assess how our highest-performing model does in terms of fairness. We can do this with `compare_confusion_matrices`. As you would do with scikit-learn's `confusion_matrix`, the first argument is the ground truth or target values (often known as `y_true`), and the second is the model's predictions (often known as `y_pred`). The difference here is it takes two sets of `y_true` and `y_pred`, one corresponding to one segment of the observations and one to another. After these first four arguments, you give each segment a name, so this is what the following two arguments tell you. Lastly, `compare_fpr=True` ensures that it will compare the **False Positive Rate** (**FPR**) between both confusion matrices. Have a look at the following code snippet:

```
y_test_pred = fitted_class_mdls['catboost_reg']['preds_test']
_ = mldatasets.compare_confusion_matrices(
    y_test[X_test.race==1],
    y_test_pred[X_test.race==1],
    y_test[X_test.race==0],
    y_test_pred[X_test.race==0],
    'Caucasian',
    'African-American',
    compare_fpr=True
)
y_test_pred =  fitted_class_mdls['catboost_base']['preds_test']
_ = mldatasets.compare_confusion_matrices(
    y_test[X_test.race==1],
    y_test_pred[X_test.race==1],
    y_test[X_test.race==0],
    y_test_pred[X_test.race==0],
    'Caucasian',
    'African-American',
    compare_fpr=True
)
```

The preceding snippet outputs *Figure 12.14* and *Figure 12.15*, corresponding to the regularized and base models, respectively. You can see *Figure 12.14* here:
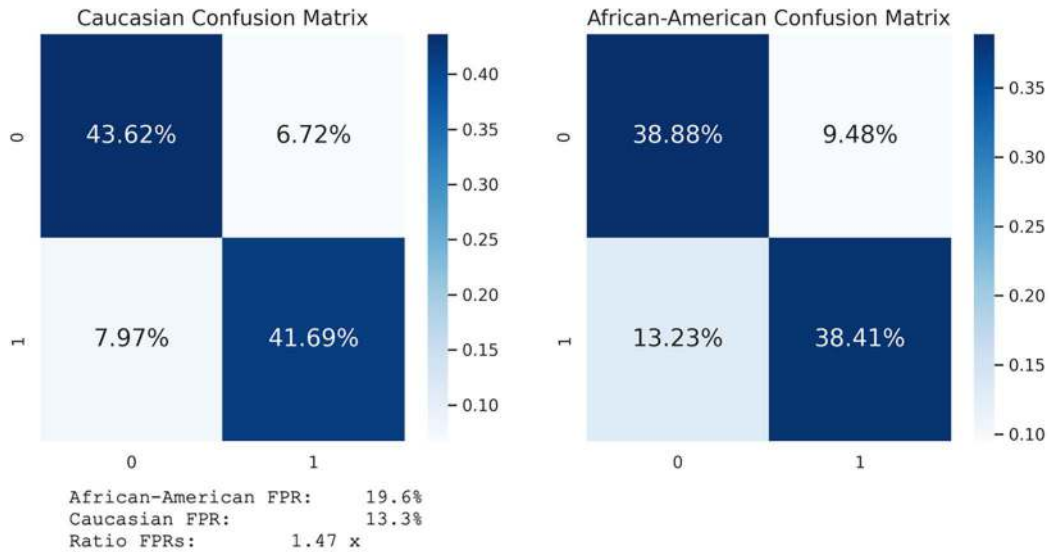
Figure 12.14: Confusion matrices between races for the regularized CatBoost model

Figure 12.15 tells us that the FPRs are significantly lower for the regularized model. You can see the output here:
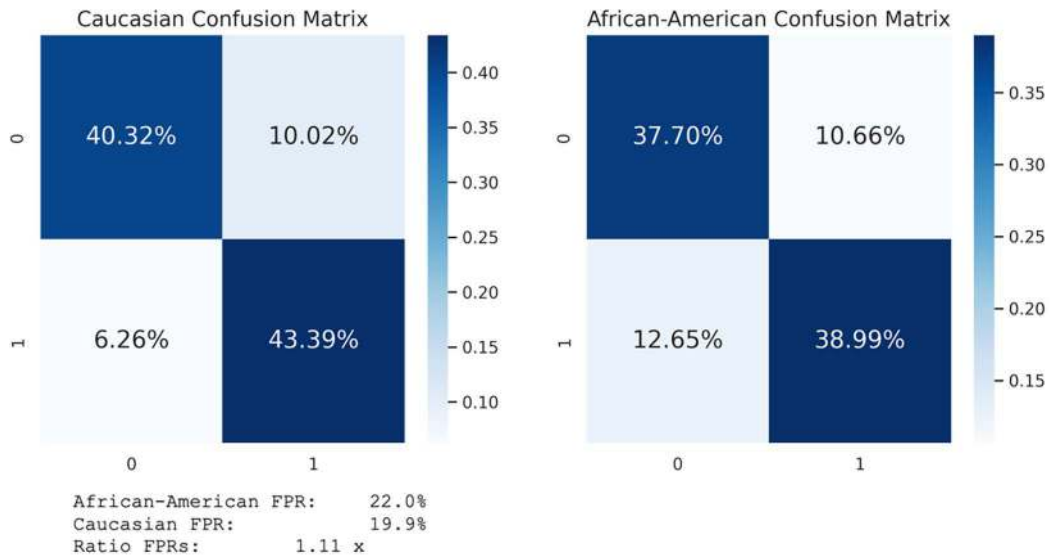
Figure 12.15: Confusion matrices between races for the base CatBoost model

However, the base model in *Figure 12.15* has an FPR ratio of 1.11 compared to 1.47 for the regularized model, which is significantly more despite the similar overall metrics. But when trying to achieve several goals at once, it's hard to evaluate and compare models, and that's what we will do in the next section.

Optimizing for fairness with Bayesian hyperparameter tuning and custom metrics

Our mission is to produce a model with high precision and good recall while maintaining fairness across different races. So, achieving this mission will require a custom metric to be designed.

Designing a custom metric

We could use the F1 score, but it treats precision and recall equally, so we will have to create a weighted metric. We can also factor in how precision and recall are distributed for each race. One way to do this is by using the standard deviation, which quantifies the variation in this distribution. To that end, we will penalize precision with half the intergroup standard deviation for precision, and we can call this penalized precision. The formula is shown here:

$$P_{penalized} = P - \frac{1}{2}\sigma_P$$

We can do the same for recall, as illustrated here:

$$R_{penalized} = R - \frac{1}{2}\sigma_R$$

Then, we make a weighted average for penalized precision and recall where precision is worth twice as much as recall, as illustrated here:

$$\text{custom\_metric} = \frac{2 \times P_{penalized} + R_{penalized}}{3}$$

To compute this new metric, we will need to create a function that we can call `weighted_penalized_pr_average`. It takes `y_true` and `y_pred` as the predictive performance metrics. However, it also includes `X_group` with a `pandas` series or array containing the values for the group, and `group_vals` with a list of values that it will subset the predictions by. In this case, the group is `race`, which can be values from 0 to 2. The function includes a `for` loop that iterates through these possible values, subsetting the predictions by each group. That way, it can compute precision and recall for each group. After this, the rest of the function simply performs the three mathematical operations outlined previously. The code can be seen in the following snippet:

```
def weighted_penalized_pr_average(y_true, y_pred, X_group,\
                    group_vals, penalty_mult=0.5,\
                    precision_mult=2,\
                    recall_mult=1):
    precision_all = metrics.precision_score(
```

```
        y_true, y_pred, zero_division=0
    )
    recall_all = metrics.recall_score(
        y_true, y_pred, zero_division=0
    )
    p_by_group = []
    r_by_group = []
    for group_val in group_vals:
        in_group = X_group==group_val
        p_by_group.append(metrics.precision_score(
            y_true[in_group], y_pred[in_group], zero_division=0
            )
        )
        r_by_group.append(metrics.recall_score(
            y_true[in_group], y_pred[in_group], zero_division=0
            )
        )
    precision_all = precision_all - \
                  (np.array(p_by_group).std()*penalty_mult)
    recall_all = recall_all -\
              (np.array(r_by_group).std()*penalty_mult)
    return ((precision_all*precision_mult)+
          (recall_all*recall_mult))/\
          (precision_mult+recall_mult)
```

Now, to put this function to work, we will need to run the tuning.

Running Bayesian hyperparameter tuning

Bayesian optimization is a *global optimization method* that uses the posterior distribution of black-box objective functions and their continuous parameters. In other words, it sequentially searches the best parameters to test next based on past results. Unlike grid search, it doesn't try fixed combinations of parameters on a grid but exploits what it already knows and explores the unknown.

The `bayesian-optimization` library is model-agnostic. All it needs is a function and parameters with their bounds. It will explore values for those parameters within those bounds. The function takes those parameters and returns a number. This is the number, or target, that the Bayesian optimization algorithm will maximize.

The following code is for the `objective` function, which initializes a `RepeatedStratifiedKFold` cross-validation with four splits and three repeats. It then iterates across the splits and fits the `CatBoostClassifier` with them. Lastly, it computes the `weighted_penalized_pr_average` custom metric for each model training and appends it to a list. Finally, the function returns the `median` of the custom metric for all 12 training samples. The code is shown in the following snippet:

```python
def hyp_catboost(l2_leaf_reg, scale_pos_weight):
    cv = model_selection.RepeatedStratifiedKFold(
        n_splits=4,n_repeats=3, random_state=rand
    )
    metric_l = []
    for train_index, val_index in cv.split(X_train, y_train):
        X_train_cv, X_val_cv = X_train.iloc[train_index],\
                               X_train.iloc[val_index]
        y_train_cv, y_val_cv = y_train.iloc[train_index],
                               y_train.iloc[val_index]
        mdl = cb.CatBoostClassifier(
            random_seed=rand, learning_rate=0.5, verbose=0, depth=5,\
            l2_leaf_reg=l2_leaf_reg, scale_pos_weight=scale_pos_weight
        )
        mdl = mdl.fit(X_train_cv, y_train_cv)
        y_val_pred = mdl.predict(X_val_cv)
        metric = weighted_penalized_pr_average(
            y_val_cv,y_val_pred, X_val_cv['race'], range(3)
        )
        metric_l.append(metric)
    return np.median(np.array(metric_l))
```

Now that the function has been defined, running the Bayesian optimization process is straightforward. First, set the parameter-bounds dictionary (pbounds), initialize BayesianOptimization with the hyp_ catboost function, and then run it with maximize. The maximize function takes init_points, which sets how many iterations it should run initially using random exploration. Then, n_iter is the number of optimization iterations it should perform to find the maximum value. We will set init_points and n_iter to 3 and 7, respectively, because it could take a long time, but the larger these numbers, the better. The code can be seen in the following snippet:

```python
pbounds = {
    'l2_leaf_reg': (2,4),
    'scale_pos_weight': (0.55,0.85)
    }
optimizer = BayesianOptimization(
    hyp_catboost,
    pbounds,
    random_state=rand
)
optimizer.maximize(init_points=3, n_iter=7)
```

Once it's finished, you can access the best parameters, like this:

```python
print(optimizer.max['params'])
```

It will return a dictionary with the parameters, as follows:

```
{'l2_leaf_reg': 2.0207483077713997, 'scale_pos_weight': 0.7005623776446217}
```

Now, let's fit a model with these parameters and evaluate it.

Fitting and evaluating a model with the best parameters

Initializing `CatBoostClassifier` with these parameters is as simple as passing the `best_params` dictionary as an argument. Then, all you need to do is `fit` the model and evaluate it (`evaluate_class_mdl`). The code is shown in the following snippet:

```
cb_opt = cb.CatBoostClassifier(
    random_seed=rand,
    depth=5,
    learning_rate=0.5,
    verbose=0,
    **optimizer.max['params']
)
cb_opt = cb_opt.fit(X_train, y_train)
fitted_class_mdls['catboost_opt'] = mldatasets.evaluate_class_mdl(
    cb_opt,
    X_train,
    X_test,
    y_train,
    y_test,
    plot_roc=False,
    plot_conf_matrix=True,
    ret_eval_dict=True
)
```

The preceding snippet outputs the following predictive performance metrics:

```
Accuracy_train:  0.9652        Accuracy_test:   0.8192
Precision_test:  0.8330        Recall_test:     0.8058
ROC-AUC_test:    0.8791        F1_test:         0.8192
```

They are the highest `Accuracy_test`, `Precision_test`, and `Recall_test` metrics we have achieved so far. Let's now see how the model fares with fairness using `compare_confusion_matrices`. Have a look at the following code snippet:

```
y_test_pred = fitted_class_mdls['catboost_opt']['preds_test']
_ = mldatasets.compare_confusion_matrices(
    y_test[X_test.race==1],
    y_test_pred[X_test.race==1],
    y_test[X_test.race==0],
```

```
        y_test_pred[X_test.race==0],
        'Caucasian',
        'African-American',
        compare_fpr=True
)
```

The preceding code outputs *Figure 12.16*, which shows some of the best fairness metrics we have obtained so far, as you can see here:
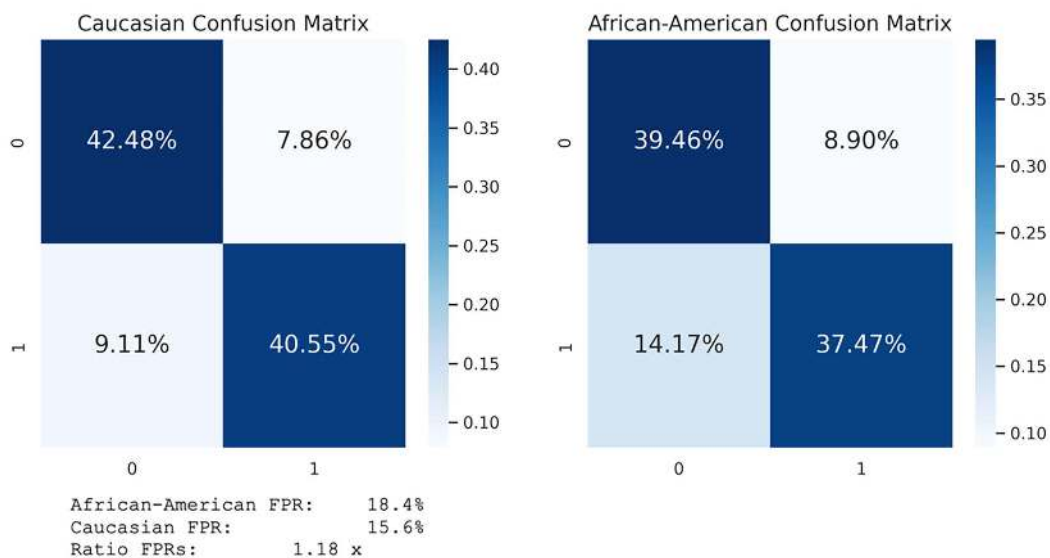
Figure 12.16: Comparison of confusion matrices between races for the optimized CatBoost model

These results are good, but we cannot be completely assured that the model is not racially biased because the feature is still there. One way to measure its impact is through feature importance methods.

Examining racial bias through feature importance

Although CatBoost is our best-performing model in most metrics, including accuracy, precision, and F1 score, we are moving forward with XGBoost because CatBoost doesn't support interaction constraints, which we will implement in the next section. But first, we will compare them both in terms of what they found important. Also, **SHapley Additive exPlanations** (**SHAP**) values provide a robust means to measure and visualize feature importance, so let's compute them for our optimized CatBoost and regularized XGBoost models. To do so, we need to initialize `TreeExplainer` with each model and then use `shap_values` to produce the values for each, as illustrated in the following code snippet:

```
fitted_cb_mdl = fitted_class_mdls['catboost_opt']['fitted']
shap_cb_explainer = shap.TreeExplainer(fitted_cb_mdl)
shap_cb_values = shap_cb_explainer.shap_values(X_test)
fitted_xgb_mdl = fitted_class_mdls['xgb_reg']['fitted']
shap_xgb_explainer = shap.TreeExplainer(fitted_xgb_mdl)
shap_xgb_values = shap_xgb_explainer.shap_values(X_test)
```

Next, we can generate two `summary_plot` plots side by side, using Matplotlib's `subplot`, as follows:

```python
ax0 = plt.subplot(1, 2, 1)
shap.summary_plot(
    shap_xgb_values,
    X_test,
    plot_type="dot",
    plot_size=None,
    show=False
)
ax0.set_title("XGBoost SHAP Summary")
ax1 = plt.subplot(1, 2, 2)
shap.summary_plot(
    shap_cb_values,
    X_test,
    plot_type="dot",
    plot_size=None,
    show=False
)
ax1.set_title("Catboost SHAP Summary")
```

The preceding snippet generates *Figure 12.17*, which shows how similar CatBoost and XGBoost are. This similarity shouldn't be surprising because, after all, they are both gradient-boosted decision trees. The bad news is that `race` is in the top four for both. However, the prevalence of the shade that corresponds to lower feature values on the right suggests that African American (`race=0`) negatively correlates with recidivism.

The output can be seen here:
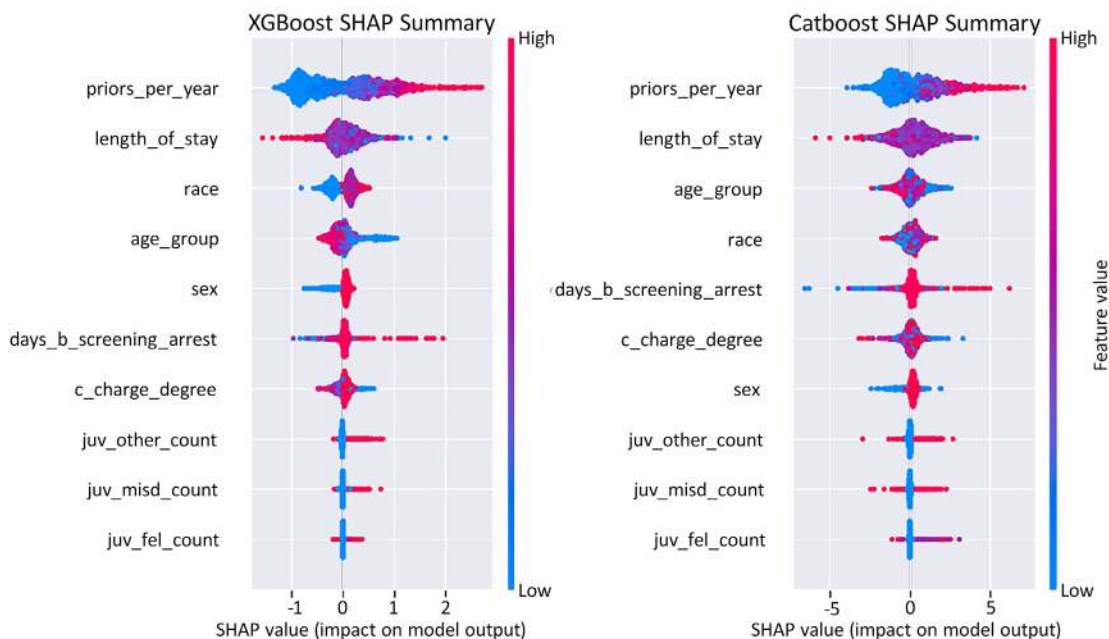
Figure 12.17: SHAP summary plot for the regularized XGBoost and optimized CatBoost models

In any case, it makes sense to remove race from the training data, but we must first ascertain why the model thinks this is a critical feature. Have a look at the following code snippet:

```
shap_xgb_interact_values =\
                shap_xgb_explainer.shap_interaction_values(X_test)
```

In *Chapter 4*, *Global Model-Agnostic Interpretation Methods*, we discussed assessing interaction effects. It's time to revisit this topic, but this time, we will extract SHAP's interaction values (shap_interaction_values) instead of using SHAP's dependence plots. We can easily rank SHAP interactions with a summary_plot plot. A SHAP summary plot is very informative, but it's not nearly as intuitive as a heatmap for interactions. To generate a heatmap with labels, we must place the shap_xgb_interact_values summed on the first axis in a DataFrame, then name the columns and rows (index) with the names of the features. The rest is simply using Seaborn's heatmap function to plot the DataFrame as a heatmap. The code can be seen in the following snippet:

```
shap_xgb_interact_avgs = np.abs(
    shap_xgb_interact_values
).mean(0)
np.fill_diagonal(shap_xgb_interact_avgs, 0)
shap_xgb_interact_df = pd.DataFrame(shap_xgb_interact_avgs)
shap_xgb_interact_df.columns = X_test.columns
shap_xgb_interact_df.index = X_test.columns
sns.heatmap(shap_xgb_interact_df, cmap='Blues', annot=True,\
            annot_kws={'size':13}, fmt='.2f', linewidths=.5)
```

The preceding code produced the heatmap shown in *Figure 12.18*. It demonstrates how `race` interacts most heavily with `length_of_stay`, `age_group`, and `priors per year`. These interactions would, of course, disappear once we removed `race`. However, given this finding, careful consideration ought to be given if these features don't have racial bias built in. Research supports the need for `age_group` and `priors_per_year`, which leaves `length_of_stay` as a candidate for scrutiny. We won't do this in this chapter, but it's certainly food for thought:
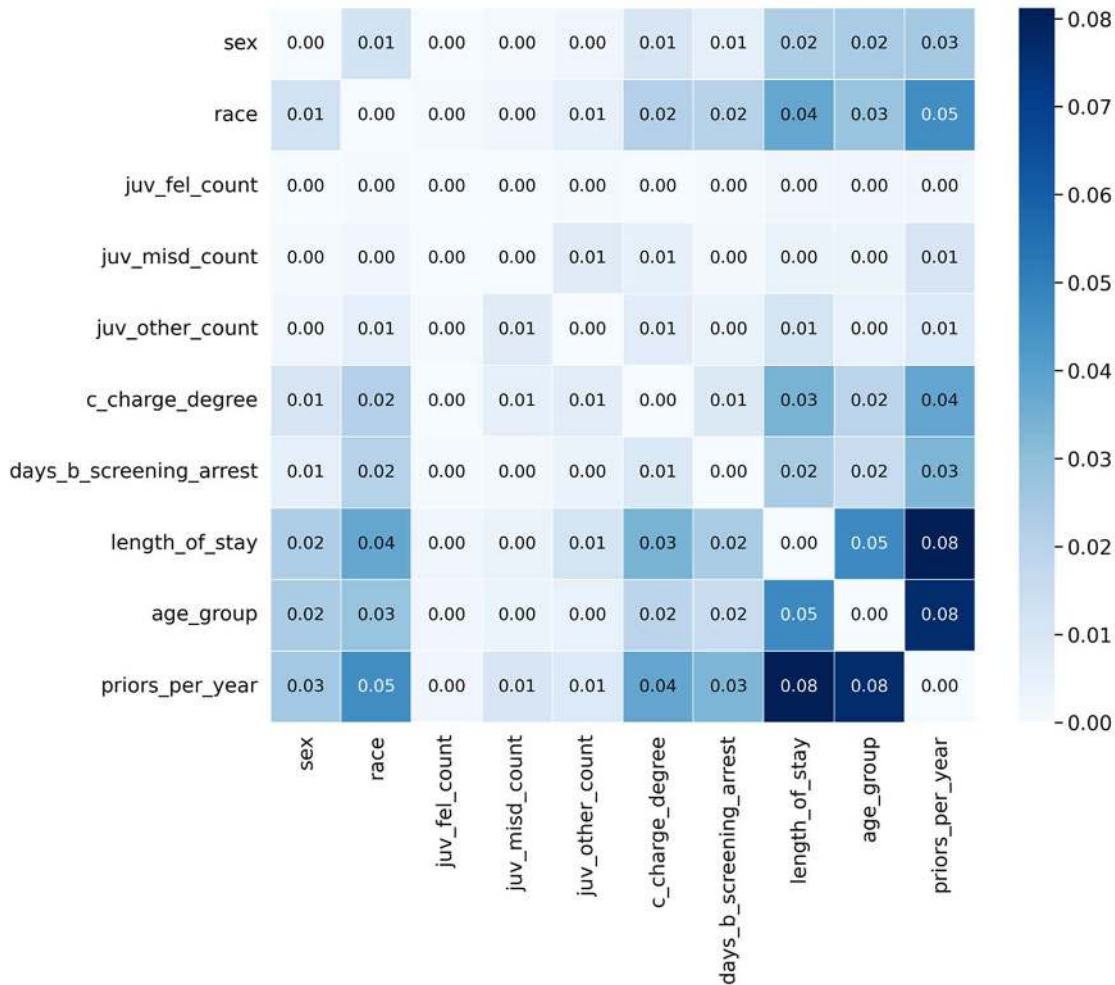
Figure 12.18: Heatmap with SHAP interaction values for the regularized XGBoost model

Another interesting insight from *Figure 12.18* is how features can be clustered. You can draw a box around the lower-right quadrant between `c_charge_degree` and `priors_per_year` because, once we remove `race`, most of the interaction will be located here. There are many benefits to limiting troubling interactions. For instance, why should all the juvenile delinquency features, such as `juv_fel_count`, interact with `age_group`? Why should `sex` interact with `length_of_stay`? Next, we will learn how to place a fence around the lower-right quadrant, limiting interactions between those features with **interaction constraints**. We will also ensure monotonicity for `priors_per_year` with **monotonic constraints**.

Implementing model constraints

We will discuss how to implement constraints first with XGBoost and all popular tree ensembles, for that matter, because the parameters are named the same (see *Figure 12.12*). Then, we will do so with TensorFlow Lattice. But before we move forward with any of that, let's remove `race` from the data, as follows:

```
X_train_con = X_train.drop(['race'], axis=1).copy()
X_test_con = X_test.drop(['race'], axis=1).copy()
```

Now, with `race` out of the picture, the model may still have some bias. However, the feature engineering we performed and the constraints we will place can help align the model against them, given the double standards we found in *Chapter 6, Anchors and Counterfactual Explanations*. That being said, the resulting model might perform worse against the test data. There are two reasons for this, outlined here:

- **Loss of information:** Race, especially through interaction with other features, impacted the outcome, so it unfortunately carried some information.
- **Misalignment between reality and policy-driven ideals:** This occurs when the main reason to enforce these constraints is to ensure that the model not only complies with domain knowledge but also ideals, and these might not be evident in the data. We must remember that a whole host of institutional racism could have tainted the ground truth. The model reflects the data, but the data reflects reality on the ground, which is itself biased.

With that in mind, let's get started with constraint implementation!

Constraints for XGBoost

We will take three simple steps in this section. We will first define our training parameters, then train and evaluate a constrained model, and, lastly, examine the effects of the constraints.

Setting regularization and constraint parameters

We take the best parameters for our regularized XGBoost model with `print(fitted_class_mdls['xgb_reg']['cv_best_params'])`. They are in the `best_xgb_params` dictionary, along with `eta` and `max_depth`. Then, to enforce monotonic constraints on `priors_per_year`, we must first know its position and the direction of the monotonic correlation. From *Figure 12.8*, we know the answers to both questions. It is the last feature, and the correlation is positive, so the `mono_con` tuple should have nine items, with the last one being a `1` and the rest `0`s. As for interaction constraints, we will only allow the last five features to interact with each other, and the same goes for the first four. The `interact_con` tuple is a list of lists that reflects these constraints. The code can be seen in the following snippet:

```
best_xgb_params = {'eta': 0.3, 'max_depth': 28,\
                   'reg_alpha': 0.2071, 'reg_lambda': 0.6534,\
                   'scale_pos_weight': 0.9114}
mono_con = (0,0,0,0,0,0,0,0,1)
interact_con = [[4, 5, 6, 7, 8],[0, 1, 2, 3]]
```

Next, we will train and evaluate the XGBoost model with these constraints.

Training and evaluating the constrained model

We will now train and evaluate our constrained model. First, we initialize the `XGBClassifier` model with our constraint and regularization parameters and then fit it using training data that lacks the `race` feature (`X_train_con`). We then evaluate the predictive performance with `evaluate_class_mdl` and compare fairness with `compare_confusion_matrices`, as we have done before. The code can be seen in the following snippet:

```python
xgb_con = xgb.XGBClassifier(
    seed=rand,monotone_constraints=mono_con,\
    interaction_constraints=interact_con, **best_xgb_params
)
xgb_con = xgb_con.fit(X_train_con, y_train)
fitted_class_mdls['xgb_con'] = mldatasets.evaluate_class_mdl(
    xgb_con, X_train_con, X_test_con, y_train, y_test,\
    plot_roc=False, ret_eval_dict=True
)
y_test_pred = fitted_class_mdls['xgb_con']['preds_test']
_ = mldatasets.compare_confusion_matrices(
    y_test[X_test.race==1],
    y_test_pred[X_test.race==1],
    y_test[X_test.race==0],
    y_test_pred[X_test.race==0],
    'Caucasian',
    'African-American',
    compare_fpr=True
)
```

The preceding snippet produces the confusion matrix pair in *Figure 12.19* and some predictive performance metrics. If we compare the matrices to those in *Figure 12.16*, racial disparities, as measured by our FPR ratio, took a hit. Also, predictive performance is lower than the optimized CatBoost model across the board, by 2–4%. We could likely increase these metrics a bit by performing the same *Bayesian hyperparameter tuning* on this model.

The confusion matrix output can be seen here:

African-American FPR: 23.0%
Caucasian FPR: 16.5%
Ratio FPRs: 1.39 x
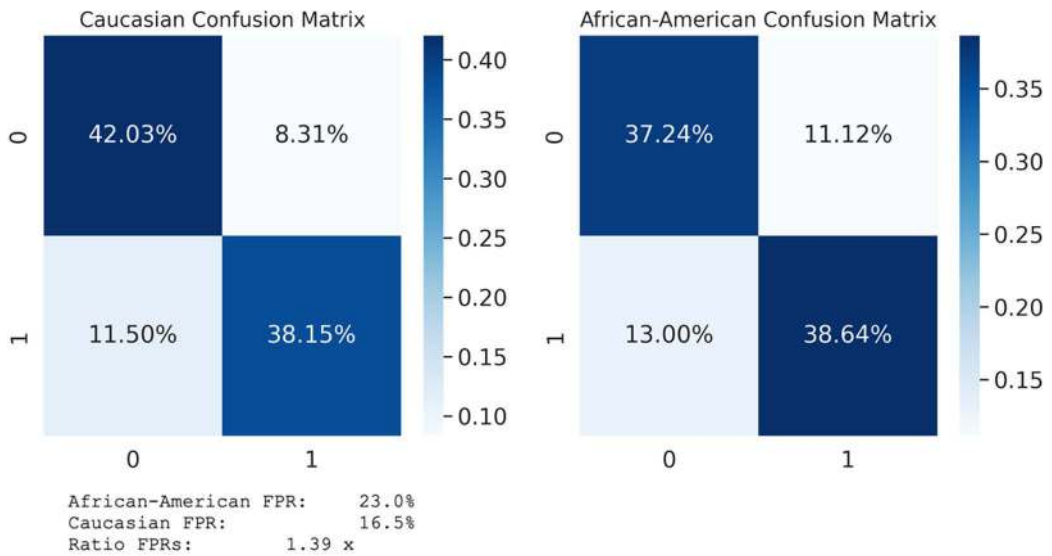
Figure 12.19: Comparison of confusion matrices between races for the constrained XGBoost model

One thing to consider is that, although racial inequity is a primary concern of this chapter, we also want to ensure that the model is optimal in other ways. As stated before, it's a balancing act. For instance, it's only fitting that defendants with the most `priors_per_year` are riskier than those with the least, and we ensured this with monotonic constraints. Let's verify these outcomes!

Examining constraints

An easy way to observe the constraints in action is to plot a SHAP `summary_plot`, as we did in *Figure 12.17*, but this time, we will only plot one. Have a look at the following ode snippet:

```
fitted_xgb_con_mdl = fitted_class_mdls['xgb_con']['fitted']
shap_xgb_con_explainer = shap.TreeExplainer(fitted_xgb_con_mdl)
shap_xgb_con_values = shap_xgb_con_explainer.shap_values(
    X_test_con
)
shap.summary_plot(
    shap_xgb_con_values, X_test_con, plot_type="dot"
)
```

The preceding code produces *Figure 12.20*. This demonstrates how `priors_per_year` from left to right is a cleaner gradient, which means that lower values are consistently having a negative impact, and the higher ones a positive one—as they should!

You can see the output here:
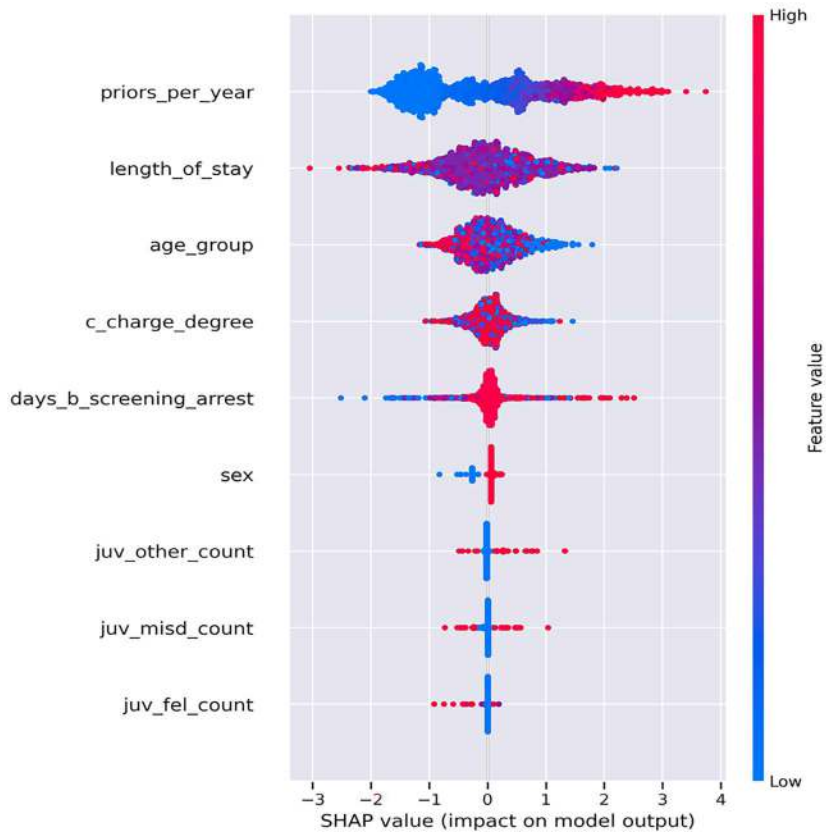
Figure 12.20: SHAP summary plot for the constrained XGBoost model

Next, let's examine the `age_group` versus `priors_per_year` interaction we saw through the lens of the data in *Figure 12.7*. We can also use `plot_prob_contour_map` for models by adding extra arguments, as follows:

- The fitted model (`fitted_xgb_con_mdl`)
- The DataFrame to use for inference with the model (`X_test_con`)
- The names of the two columns in the DataFrame to compare on each axis (`x_col` and `y_col`)

The outcome is an interaction partial dependence plot, like those shown in *Chapter 4, Global Model-Agnostic Interpretation Methods*, except that it uses the dataset (`recidivism_df`) to create the histograms for each axis. We will create two such plots right now for comparison—one for the regularized XGBoost model and another for the constrained one. The code for this can be seen in the following snippet:

```
mldatasets.plot_prob_contour_map(
    recidivism_df.age_group, recidivism_df.priors_per_year,
    recidivism_df.is_recid, x_intervals=ordenc.categories_[2],
    y_intervals=6, use_quantiles=True, xlabel='Age Group',
```

```
        ylabel='Priors Per Year', model=fitted_xgb_mdl,
        X_df=X_test,x_col='age_group',y_col='priors_per_year',
        title='Probability of Recidivism by Age/Priors per Year \
            (according to XGBoost Regularized Model)'
    )
mldatasets.plot_prob_contour_map(
        recidivism_df.age_group, recidivism_df.priors_per_year,
        recidivism_df.is_recid, x_intervals=ordenc.categories_[2],
        y_intervals=6, use_quantiles=True, xlabel='Age Group',
        ylabel='Priors Per Year', model=fitted_xgb_con_mdl,
        X_df=X_test_con,x_col='age_group',y_col='priors_per_year',
        title='(according to XGBoost Constrained Model)'
    )
```

The preceding code produces the plots shown in *Figure 12.21*. It shows that the regularized XGBoost model reflects the data (see *Figure 12.7*). On the other hand, the constrained XGBoost model smoothened and simplified the contours, as can be seen here:
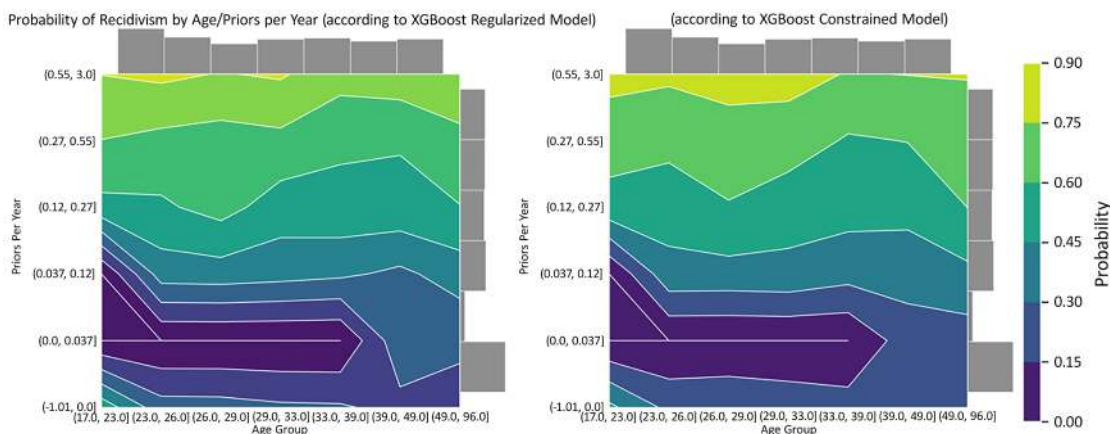
Figure 12.21: Recidivism probability contour map for age_group and priors_per_year according to XGBoost regularized and constrained models

Next, we can generate the SHAP interaction values heatmap from *Figure 12.18* but for the constrained model. The code is the same but uses the `shap_xgb_con_explainer` SHAP explainer and `X_test_con` data. The code can be seen in the following snippet:

```
shap_xgb_interact_values =\
        shap_xgb_con_explainer.shap_interaction_values(X_test_con)
shap_xgb_interact_df =\
        pd.DataFrame(np.sum(shap_xgb_interact_values, axis=0))
shap_xgb_interact_df.columns = X_test_con.columns
shap_xgb_interact_df.index = X_test_con.columns
sns.heatmap(
```

```
        shap_xgb_interact_df, cmap='RdBu', annot=True,
        annot_kws={'size':13}, fmt='.0f', linewidths=.5
    )
```

The preceding snippet outputs the heatmap shown in *Figure 12.22*. It shows how the interaction constraints were effective because of zeros in the lower-left and lower-right quadrants, which correspond to interactions between the two groups of features we separated. If we compare with *Figure 12.18*, we can also tell how the constraints shifted the most salient interactions, making `age_group` and `length_of_stay` by far the most important ones.

The output can be seen here:
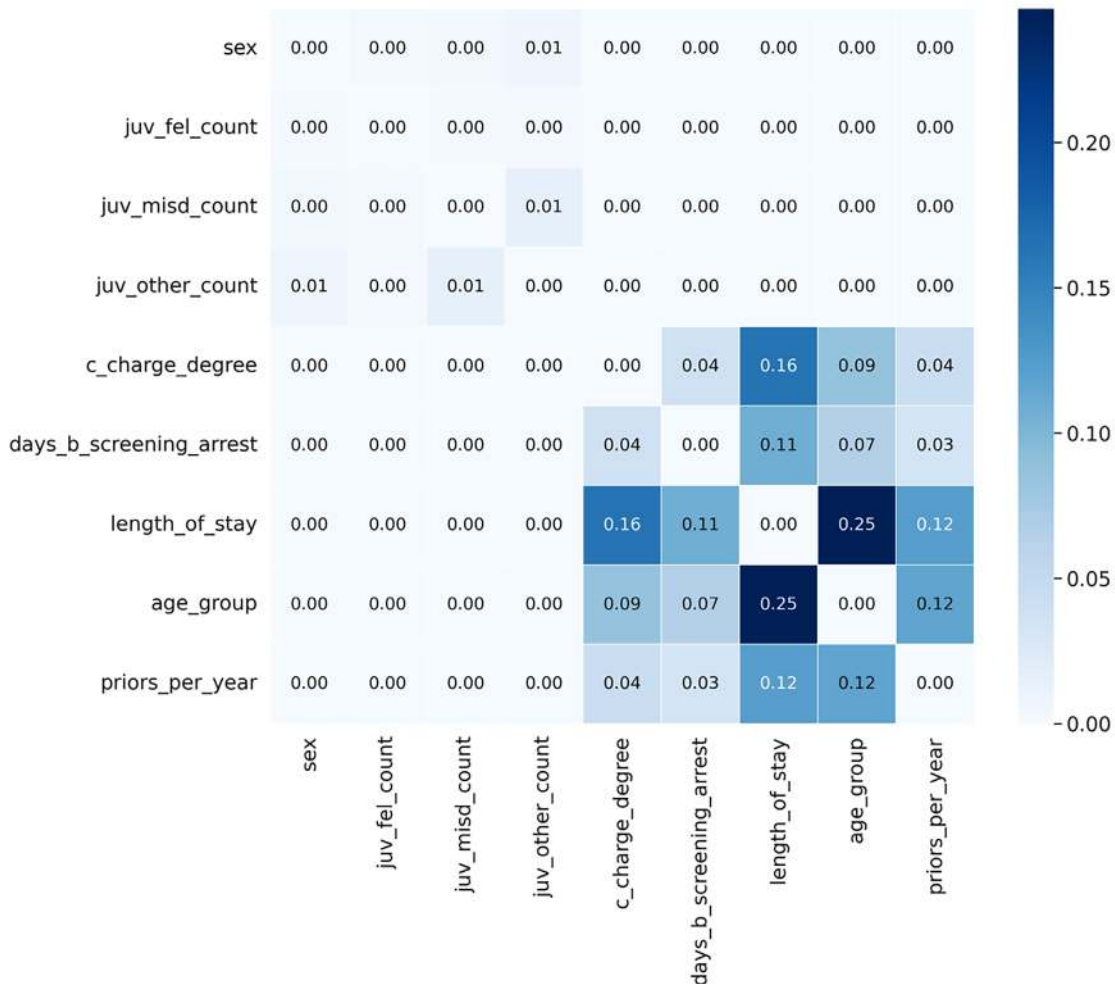
Figure 12.22: Heatmap with SHAP interaction values for the constrained XGBoost model

Now, let's see how TensorFlow implements monotonicity and other "shape constraints" via Tensor-Flow Lattice.

Constraints for TensorFlow Lattice

Neural networks can be very efficient in finding an optimal solution for the `loss` function. The loss is tied to a consequence we wish to predict. In this case, that would be 2-year recidivism. In ethics, a *utilitarian* (or *consequentialist*) view of fairness has no problem with this as long as the model's training data isn't biased. Yet a *deontological* view is that ethical principles or policies drive ethical questions and supersede consequences. Inspired by this, **TensorFlow Lattice** (**TFL**) can embody ethical principles in models as model shape constraints.

A lattice is an **interpolated lookup table**, which is a grid that approximates inputs to outputs through interpolation. In high-dimensional space, these grids become hypercubes. The mappings of each input to output are constrained through **calibration layers**, and they support many kinds of constraints—not just monotonicity. *Figure 12.23* shows this here:
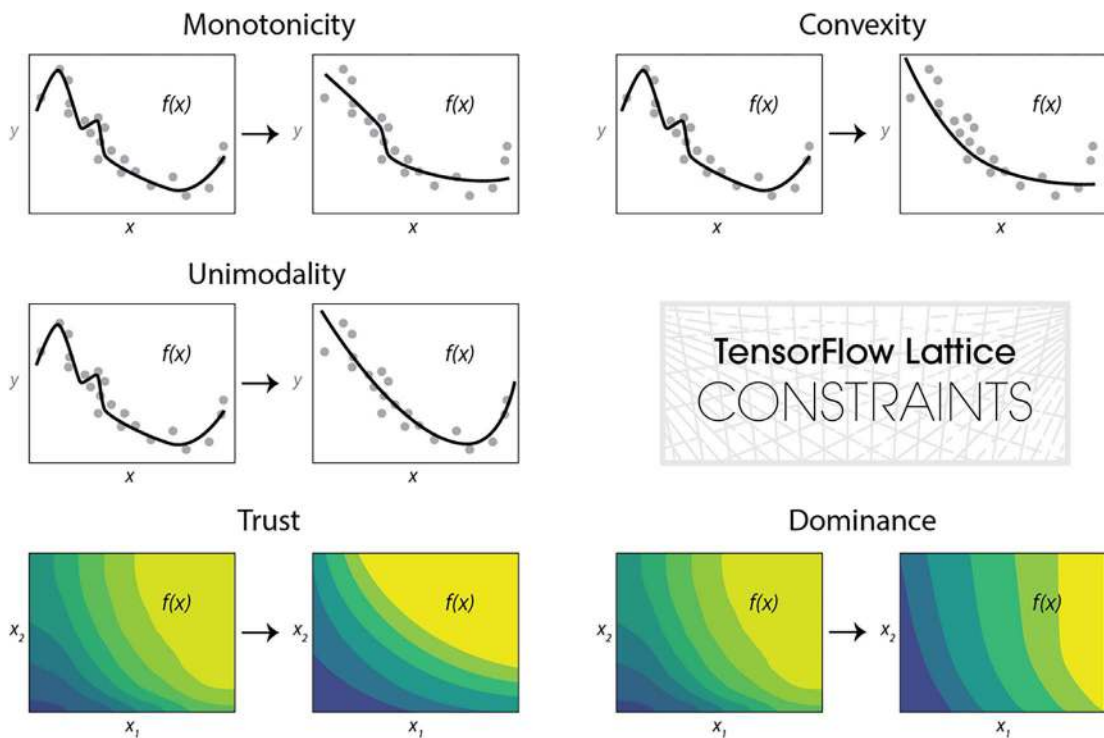
Figure 12.23: Some of the constraints supported by TensorFlow Lattice

Figure 12.23 shows several shape constraints. The first three are applied to a single feature (x) constraining the $f(x)$ line, representing the output. The last two are applied to a pair of features (x_1 and x_2) constraining the color-coded contour map ($f(x)$). A brief explanation for each follows:

- **Monotonicity:** This makes the function ($f(x)$) always increase (1) or decrease (-1) against the input (x).
- **Convexity:** This forces the function ($f(x)$) to be convex (1) or concave (-1) against the input (x). Convexity can be mixed with monotonicity to have an effect like the one in *Figure 12.23*.

- **Unimodality:** This is like monotonicity, except that it goes in both directions, allowing the function ($f(x)$) to have a single valley (1) or peak (-1).
- **Trust:** This forces one monotonic feature (x_1) to rely on another one (x_2). The example in *Figure 12.23* is **Edgeworth Trust**, but there's also a **Trapezoid Trust** variation with a different shape constraint.
- **Dominance:** Monotonic dominance constrains one monotonic (x_1) feature to define the direction of the slope or effects when compared to another (x_2). An alternative, range dominance, is similar, except both features are monotonic.

Neural networks are particularly prone to overfitting, and the levers for controlling it are comparably more difficult. For instance, exactly what combination of hidden nodes, dropout, weight regularization, and epochs will lead to an acceptable level of overfitting is challenging to tell. On the other hand, moving a single parameter in a tree-based model, tree depth, in one direction will likely lower overfitting to an acceptable level, albeit it might require many different parameters to make it optimal.

Enforcing shape constraints not only increases interpretability but also regularizes the model because it simplifies the function. TFL also supports different kinds of penalty-based regularization on a per-feature basis or to the calibration layer's kernel, leveraging L1 and L2 penalties via **Laplacian**, **Hessian**, **Torsion**, and **Wrinkle** regularizers. These regularizers have the effect of making functions more flat, linear, or smooth. We won't explain them but it suffices to say that there is regularization to cover many use cases.

There are also several ways to implement the framework—too many to elaborate here! Yet, it's important to point out that this example is just one of a handful of ways of implementing it. TFL comes with built-in **canned estimators** that abstract some of the configurations. You can also create a **custom estimator** using the TFL layers. For Keras, you can either use **premade models** or build a Keras model with TensorFlow Lattice layers. This last one is what we will do next!

Initializing the model and Lattice inputs

We will now create a series of *input layers*, which each include a single feature. These connect to *calibration layers*, which make each input fit into a **Piece-Wise Linear** (**PWL**) function that complies with individual constraints and regularizations, except for `sex`, which will use categorical calibration. The calibration layers all feed into a multidimensional *Lattice layer*, producing output via a *Dense layer* with *sigmoid* activation. This description can be a lot to take in, so feel free to skip ahead to *Figure 12.24* to get some visual aid.

Incidentally, there are many kinds of layers available that you can connect to produce a **Deep Lattice Network** (**DLN**), including the following:

- **Linear** for linear functions between more than one input, including those with dominance shape constraints.
- **Aggregation** to perform an aggregation function on more than one input.
- **Parallel combination** to place many calibration layers within a single function, making it compatible with Keras `Sequential` layers.

We won't use any of these layers in this example, but perhaps knowing this will inspire you to explore the TensorFlow Lattice library further. Anyway, back to this example!

The first thing to define is `lattice_sizes`, which is a tuple that corresponds to a number of vertices per dimension. We have one dimension per feature in the chosen architecture, so we need to choose nine numbers greater than or equal to two. Features with less cardinality for categorical features or inflection points for continuous ones warrant fewer vertices. However, we might also want to restrict a feature's expressiveness by purposely choosing an even smaller number of vertices. For instance, `juv_fel_count` has 10 unique values, but we will assign only two vertices to it. `lattice_sizes` is shown here:

```
lattice_sizes = [2, 2, 2, 2, 4, 5, 7, 7, 7]
```

Next, we initialize two lists, one to place all the input layers (`model_inputs`) and another for the calibration layers (`lattice_inputs`). Then, for each feature, one by one, we define an input layer with `tf.keras.layers.Input` and a calibration layer with either categorical calibration (`tfl.layers.CategoricalCalibration`) or PWL calibration (`tfl.layers.PWLCalibration`). Both input and calibration layers will be appended to their respective lists for each feature. What happens inside the calibration layer depends on the feature. All PWL calibrations use `input_keypoints`, which asks where the PWL function should be segmented. Sometimes, this is best answered with fixed widths (`np.linspace`), and other times with fixed frequency (`np.quantile`). Categorical calibration instead uses buckets (`num_buckets`) that correspond to the number of categories. All calibrators have the following arguments:

- `output_min`: The minimum output for the calibrator
- `output_max`: The maximum output for the calibrator—always has to match the output minimum + lattice size - 1
- `monotonicity`: Whether it should monotonically constrain the PWL function, and if so, how
- `kernel_regularizer`: How to regularize the function

In addition to these arguments, `convexity` and `is_cyclic` (for monotonic unimodal) can modify the constraint shape. Have a look at the following code snippet:

```python
model_inputs = []
lattice_inputs = []
sex_input = tf.keras.layers.Input(shape=[1], name='sex')
lattice_inputs.append(tfl.layers.CategoricalCalibration(
    name='sex_calib',
    num_buckets=2,
    output_min=0.0,
    output_max=lattice_sizes[0] - 1.0,
    kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.001),
    kernel_initializer='constant')(sex_input)
)
model_inputs.append(sex_input)
```

```python
juvf_input = tf.keras.layers.Input(shape=[1],\
                                   name='juv_fel_count')
lattice_inputs.append(tfl.layers.PWLCalibration(
    name='juvf_calib',
    monotonicity='none',
    input_keypoints=np.linspace(0, 20, num=5, dtype=np.float32),
    output_min=0.0,
    output_max=lattice_sizes[1] - 1.0,\
    kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.001),
    kernel_initializer='equal_slopes')(juvf_input)
)
model_inputs.append(juvf_input)
age_input = tf.keras.layers.Input(shape=[1], name='age_group')
lattice_inputs.append(tfl.layers.PWLCalibration(
    name='age_calib',
    monotonicity='none',
    input_keypoints=np.linspace(0, 6, num=7, dtype=np.float32),
    output_min=0.0,
    output_max=lattice_sizes[7] - 1.0,
    kernel_regularizer=('hessian', 0.0, 1e-4))(age_input)
)
model_inputs.append(age_input)
priors_input = tf.keras.layers.Input(shape=[1],\
                                     name='priors_per_year')
lattice_inputs.append(tfl.layers.PWLCalibration(
    name='priors_calib',
    monotonicity='increasing',
    input_keypoints=np.quantile(X_train_con['priors_per_year'],
                                np.linspace(0, 1, num=7)),
    output_min=0.0,
    output_max=lattice_sizes[8]-1.0)(priors_input))
model_inputs.append(priors_input)
```

So, we now have a list with `model_inputs` and another with calibration layers, which will be the input to the lattice (`lattice_inputs`). All we need to do now is tie these together to a lattice.

Building a Keras model with TensorFlow Lattice layers

We already have the first two building blocks of this model connected. Now, let's create the last two building blocks, starting with the lattice (`tfl.layers.Lattice`). As arguments, it takes `lattice_sizes`, output minimums and maximums, and `monotonicities` it should enforce. Note that the last item, `priors_per_year`, has monotonicity set as `increasing`. The lattice layer then feeds into the final piece, which is the `Dense` layer with `sigmoid` activation. The code can be seen in the following snippet:

```python
lattice = tfl.layers.Lattice(
    name='lattice',
    lattice_sizes=lattice_sizes,
    monotonicities=[
        'none', 'none', 'none', 'none', 'none',
        'none', 'none', 'none', 'increasing'
    ],
    output_min=0.0, output_max=1.0)(lattice_inputs)
model_output = tf.keras.layers.Dense(1, name='output',
                                     activation='sigmoid')(lattice)
```

The first two building blocks as `inputs` can now get connected with the last two as `outputs` with `tf.keras.models.Model`. And voilà! We now have a fully formed model, with the code shown here:

```python
tfl_mdl = tf.keras.models.Model(inputs=model_inputs,
                                outputs=model_output)
```

You can always run `tfl_mdl.summary()` to get an idea of how all the layers connect, but it's not as intuitive as using `tf.keras.utils.plot_model`, which is illustrated in the following code snippet:

```python
tf.keras.utils.plot_model(tfl_mdl, rankdir='LR')
```

The preceding code generates the model diagram shown here in *Figure 12.24*:
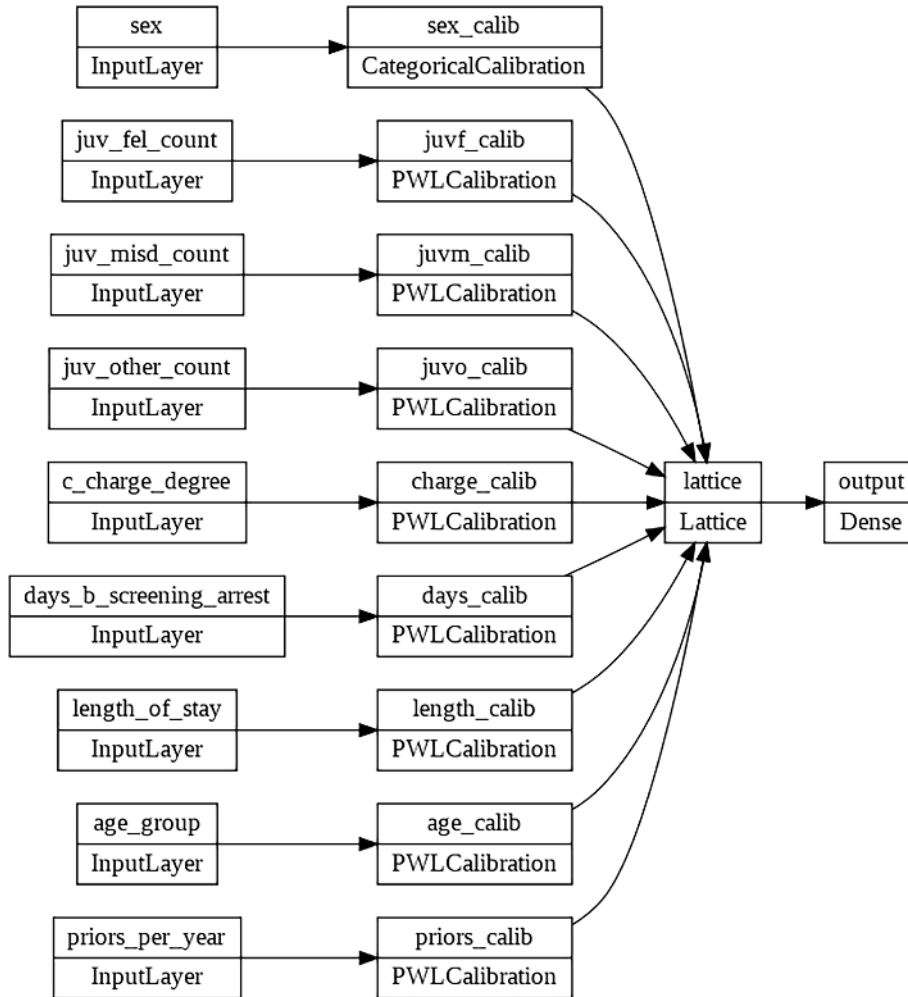
Figure 12.24: A diagram of the Keras model with TFL layers

Next, we need to compile the model. We will use a `binary_crossentropy` loss function and an `Adam` optimizer, and employ accuracy and **Area Under the Curve** (**AUC**) as metrics, as illustrated in the following code snippet:

```
tfl_mdl.compile(
    loss='binary_crossentropy',
    optimizer=tf.keras.optimizers.Adam(lr=0.001),
    metrics=['accuracy',tf.keras.metrics.AUC(name='auc')]
)
```

We are almost ready to go now! What follows next is the very last step.

Training and evaluating the model

If you take one hard look at *Figure 12.24*, you'll notice that the model doesn't have one input layer but nine, so this means that we must split our training and test data into nine parts. We can use `np.split` to do this, which will yield a list of nine NumPy arrays. As for the labels, TFL doesn't accept arrays with a single dimension. With `expand_dims`, we convert their shapes from `(N,)` to `(N,1)`, as illustrated in the following code snippet:

```python
X_train_expand = np.split(
    X_train_con.values.astype(np.float32),
    indices_or_sections=9,
    axis=1
)
y_train_expand = np.expand_dims(
    y_train.values.astype(np.float32),
    axis=1
)
X_test_expand = np.split(
    X_test_con.values.astype(np.float32),
    indices_or_sections=9,
    axis=1)
y_test_expand = np.expand_dims(
    y_test.values.astype(np.float32),
    axis=1
)
```

Now comes the training! To prevent overfitting, we can use `EarlyStopping` by monitoring the validation AUC (`val_auc`). And to account for class imbalance, in the `fit` function, we use `class_weight`, as illustrated in the following code snippet:

```python
es = tf.keras.callbacks.EarlyStopping(
    monitor='val_auc',
    mode='max',
    patience=40,
    restore_best_weights=True
)
tfl_history = tfl_mdl.fit(
    X_train_expand,
    y_train_expand,
    class_weight={0:18, 1:16},
    batch_size=128,
    epochs=300,
    validation_split=0.2,
```

```
        shuffle=True,
        callbacks=[es]
)
```

Once the model has been trained, we can use `evaluate_class_mdl` to output a quick summary of predictive performance, as we have before, and then `compare_confusion_matrices` to examine fairness, as we did previously. The code is shown in the following snippet:

```
fitted_class_mdls['tfl_con'] = mldatasets.evaluate_class_mdl(
    tfl_mdl,
    X_train_expand,
    X_test_expand,
    y_train.values.astype(np.float32),
    y_test.values.astype(np.float32),
    plot_roc=False,
    ret_eval_dict=True
)
y_test_pred = fitted_class_mdls['tfl_con']['preds_test']
_ = mldatasets.compare_confusion_matrices(
    y_test[X_test.race==1],
    y_test_pred[X_test.race==1],
    y_test[X_test.race==0],
    y_test_pred[X_test.race==0],
    'Caucasian',
    'African-American',
    compare_fpr=True
)
```

The preceding snippet produced the confusion matrices in *Figure 12.25*. The TensorFlow Lattice model performs much better than the regularized Keras model, yet the FPR ratio is better than the constrained XGBoost model. It must be noted that XGBoost's parameters were previously tuned. With TensorFlow Lattice, a lot could be done to improve FPR, including using a custom loss function or better early-stopping metrics that somehow account for racial disparities.

The output can be seen here:
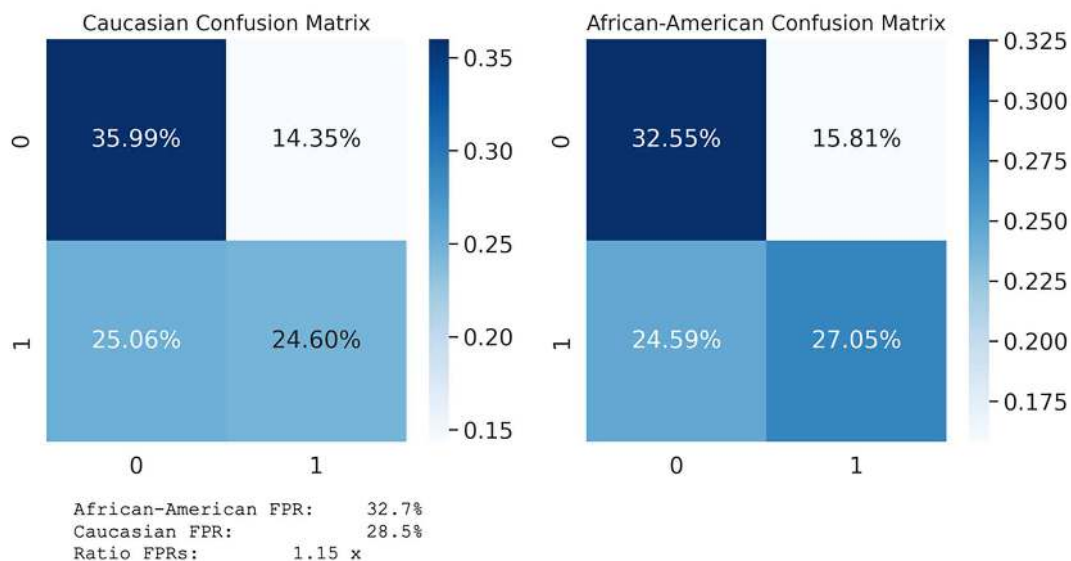
Figure 12.25: Comparison of confusion matrices between races for the constrained TensorFlow Lattice model

Next, we will make some conclusions based on what was learned in this chapter and determine if we accomplished the mission.

Mission accomplished

It's often the data that takes the blame for a poor-performing, uninterpretable, or biased model, and that can be true, but many different things can be done in the preparation and model development stages to improve it. To offer an analogy, it's like baking a cake. You need quality ingredients, yes. But seemingly small differences in the preparation of these ingredients and baking itself—such as the baking temperature, the container used, and time—can make a huge difference. Hell! Even things that are out of your control, such as atmospheric pressure or moisture, can impact baking! Even after it's all finished, how many different ways can you assess the quality of a cake?

This chapter is about these many details, and, as with baking, they are **part exact science** and **part art form.** The concepts discussed in this chapter also have far-reaching consequences, especially regarding how to optimize a problem that doesn't have a single goal and has profound societal implications. One possible approach is to combine metrics and account for imbalances. To that end, we have created a metric: a weighted average of precision recall that penalizes racial inequity, and we can efficiently compute it for all of our models and place it into the model dictionary (`fitted_class_mdls`). Then, as we have done before, we put it into a DataFrame and output it but, this time, sort by the custom metric (`wppra_test`). The code can be seen in the following snippet:

```
for mdl_name in fitted_class_mdls:
    fitted_class_mdls[mdl_name]['wppra_test'] =\
    weighted_penalized_pr_average(
        y_test,
```

```
            fitted_class_mdls[mdl_name]['preds_test'],
            X_test['race'],
            range(3)
        )
    class_metrics = pd.DataFrame.from_dict(fitted_class_mdls, 'index')[
        ['precision_test', 'recall_test', 'wppra_test']
    ]
    with pd.option_context('display.precision', 3):
        html = class_metrics.sort_values(
            by='wppra_test',
            ascending=False
        ).style.background_gradient(
            cmap='plasma',subset=['precision_test']
        ).background_gradient(
            cmap='viridis', subset=['recall_test'])
    html
```

The preceding code produced the DataFrame shown here in *Figure 12.26*:

	precision_test	recall_test	wppra_test
catboost_reg	0.836	0.818	0.810
catboost_opt	0.833	0.806	0.803
catboost_base	0.805	0.837	0.799
nu-svc_reg	0.836	0.772	0.791
xgb_con	0.810	0.777	0.783
lgbm_reg	0.798	0.709	0.747
lgbm_base	0.766	0.748	0.743
xgb_base	0.749	0.733	0.725
xgb_reg	0.800	0.618	0.717
tfl_con	0.646	0.540	0.591
nu-svc_base	0.580	0.122	0.406

Figure 12.26: Top models in this chapter when sorted by weighted penalized precision-recall average custom metric

In *Figure 12.26*, it's tempting to propose one of the models at the very top. However, they were trained with `race` as a feature and didn't account for proven criminal justice *realities*. However, the highest-performing constrained model—the XGBoost one (`xgb_con`)—lacked `race`, ensured that `priors_per_year` is monotonic and that `age_group` isn't allowed to interact with juvenile delinquency features, and it did all this while significantly improving predictive performance when compared to the original model. It is fairer, too, because it reduced the ratio of the FPR between the privileged and underprivileged groups from 1.84x (*Figure 6.2* from *Chapter 6, Anchors and Counterfactual Explanations*) to 1.39x (*Figure 12.19*). It's not perfect, but it's a massive improvement!

The mission was to prove that accuracy and domain knowledge could coexist with progress toward fairness, and we have completed it successfully. That being said, there's still room for improvement. Therefore, the plan of action would have to showcase the constrained XGBoost model to your client and continue improving and building more constrained models. The unconstrained ones should only serve as a benchmark.

You can make substantial fairness improvements if you combine the methods from this chapter with those learned in *Chapter 11, Bias Mitigation and Causal Inference Methods*. We didn't incorporate them into this chapter, to focus solely on model (or in-processing) methods that are typically not seen as part of the bias-mitigation toolkit, but they very much can assist to that end, not to mention model-tuning methods that serve to make a model more reliable.

Summary

After reading this chapter, you should now understand how to leverage data engineering to enhance interpretability, regularization to reduce overfitting, and constraints to comply with policies. The primary end goals are to place guardrails and curb the complexity that hinders interpretability.

In the next chapter, we will look at ways to enhance model reliability through adversarial robustness.

Dataset sources

- ProPublica Data Store (2019). *COMPAS Recidivism Risk Score Data and Analysis*. Originally retrieved from `https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis`

Further reading

- Hastie, T. J., Tibshirani, R. J. and Friedman, J. H. (2001). *The elements of statistical learning*. Springer-Verlag, New York, USA
- Wang, S. & Gupta, M. (2020). *Deontological Ethics By Monotonicity Shape Constraints*. AISTATS. `https://arxiv.org/abs/2001.11990`
- Cotter, A., Gupta, M., Jiang, H., Ilan, E. L., Muller, J., Narayan, T., Wang, S. & Zhu, T. (2019). *Shape Constraints for Set Functions*. ICML. `http://proceedings.mlr.press/v97/cotter19a.html`

- Gupta, M. R., Cotter A., Pfeifer, J., Voevodski, K., Canini, K., Mangylov, A., Moczydlowski, W. and van Esbroeck, A. (2016). *Monotonic Calibrated Interpolated Look-Up Tables. Journal of Machine Learning Research* 17(109):1–47. `https://arxiv.org/abs/1505.06378`

- Noble, S. (2018). *Algorithms of oppression: data discrimination in the age of Google*. NYU Press

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

13

Adversarial Robustness

Machine learning interpretation has many concerns, ranging from knowledge discovery to high-stakes ones with tangible ethical implications, like the fairness issues examined in the last two chapters. In this chapter, we will direct our attention to concerns involving reliability, safety, and security.

As we realized using the **contrastive explanation method** in *Chapter 7, Visualizing Convolutional Neural Networks*, we can easily trick an image classifier into making embarrassingly false predictions. This ability can have serious ramifications. For instance, a perpetrator can place a black sticker on a yield sign, and while most drivers would still recognize this as a yield sign, a self-driving car may no longer recognize it and, as a result, crash. A bank robber could wear a cooling suit designed to trick the bank vault's thermal imaging system, and while any human would notice it, the imaging system would fail to do so.

The risk is not limited to sophisticated image classifiers. Other models can be tricked! The **counterfactual examples** produced in *Chapter 6, Anchors and Counterfactual Explanations,* are like adversarial examples except with the goal of deception. An attacker could leverage any misclassification example, straddling the decision boundary adversarially. For instance, a spammer could realize that adjusting some email attributes increases the likelihood of circumventing spam filters.

Complex models are more vulnerable to adversarial attacks. So why would we trust them?! We can certainly make them more foolproof, and that's what adversarial robustness entails. An adversary can purposely thwart a model in many ways, but we will focus on evasion attacks and briefly explain other forms of attacks. Then we will explain two defense methods: spatial smoothing preprocessing and adversarial training. Lastly, we will demonstrate one robustness evaluation method.

These are the main topics we will cover:

- Learning about evasion attacks
- Defending against targeted attacks with preprocessing
- Shielding against any evasion attack through adversarial training of a robust classifier

Technical requirements

This chapter's example uses the `mldatasets`, `numpy`, `sklearn`, `tensorflow`, `keras`, `adversarial-robustness-toolbox`, `matplotlib`, and `seaborn` libraries. Instructions on how to install all of these libraries are in the *Preface*.

> The code for this chapter is located here: `https://packt.link/1MNrL`

The mission

The privately contracted security services industry market worldwide is valued at over USD 250 billion and is growing at around 5% annually. However, it faces many challenges, such as shortages of adequately trained guards and specialized security experts in many jurisdictions, and a whole host of unexpected security threats. These threats include widespread coordinated cybersecurity attacks, massive riots, social upheaval, and, last but not least, health risks brought on by pandemics. Indeed, 2020 tested the industry with a wave of ransomware, misinformation attacks, protests, and COVID-19 to boot.

In the wake of this, one of the largest hospital networks in the United States asked their contracted security company to monitor the correct use of masks by both visitors and personnel throughout the hospitals. The security company has struggled with this request because it diverts security personnel from tackling other threats, such as intruders, combative patients, and belligerent visitors. It has video surveillance in every hallway, operating room, waiting room, and hospital entrance. It's impossible to have eyes on every camera feed every time, so the security company thought they could assist guards with deep learning models.

These models already alert unusual activities, such as running in the hallways and brandishing weapons anywhere on the premises. They have proposed to the hospital network that they would like to add a new model that detects masks' correct usage. Before COVID-19, there were policies in place for mandatory mask usage in certain areas of each hospital, and during COVID-19, it was required everywhere. The hospital administrators would like to turn on and off this monitoring feature, depending on pandemic risk levels moving forward. They realize that personnel get fatigued and forget to put masks back on, or they partially slip off at times. Many visitors are also hostile toward using masks and may wear one when entering the hospital but take it off when no guard is around. This isn't always intentional, so they wouldn't want to dispatch guards on every alert, unlike other threats. Instead, they'd rather use awareness and a little bit of shame to modify behavior and only intervene with repeat offenders:

Figure 13.1: Radar speed signs like this one help curb speeding

Awareness is a very effective method with radar speed signs (see *Figure 13.1*), which make roads safer by only making drivers aware that they are driving too fast. Likewise, having a screen at the end of heavily trafficked hallways showing snapshots of those who have recently either mistakenly or purposely not complied with mandatory mask usage potentially creates some embarrassment for offenders. The system will log repeat offenders so that security guards can look for them and either make them comply or ask them to vacate the premises.

There's some concern with visitors trying to trick the model into evading compliance, so the security company has hired you to ensure that the model is robust in the face of this kind of adversarial attack. Security officers have noticed some low-tech trickery before, such as people momentarily covering their faces with their hands or a part of their sweater when they realize cameras monitor them. Also, in one disturbing incident, a visitor dimmed the lights and sprayed some gel on a camera, and in another, an individual painted their mouth. However, there are concerns about higher-tech attacks, such as jamming the camera's wireless signal or shining high-powered lasers directly into cameras. Devices that perform these attacks are increasingly easier to obtain and could impact other surveillance functions on a larger scale, like preventing theft. The security company hopes this robustness exercise can inform their efforts to improve every surveillance system and model.

Eventually, the security company would like to produce its own dataset with face images from the hospitals they monitor. Meanwhile, synthetically masked faces from external sources are the best they can do to productionize a model in the short term. To this end, you have been provided a large dataset of synthetically correctly and incorrectly masked faces and their unmasked counterparts. The two datasets were combined into a single one, and the original dimensions of 1,024 × 1,024 were reduced to the thumbnail size of 124 × 124. Also, for efficiency's sake, 21,000 images were sampled from roughly 210,000 in these datasets.

The approach

You've decided to take a four-fold approach:

- Exploring several possible evasion attacks to understand how vulnerable the model is to them and how credible they are as threats
- Using a preprocessing method to protect a model against these attacks
- Leveraging adversarial retraining to produce a robust classifier that is intrinsically less prone to many of these attacks
- Evaluating robustness with state-of-the-art methods to assure hospital administrators that the model is adversarially robust

Let's get started!

The preparations

You will find the code for this example here: `https://github.com/PacktPublishing/Interpretable-Machine-Learning-with-Python-2E/tree/main/13/Masks.ipynb`

Loading the libraries

To run this example, you need to install the following libraries:

- `mldatasets` to load the dataset
- `numpy` and `sklearn` (scikit-learn) to manipulate it
- `tensorflow` to fit the models
- `matplotlib` and `seaborn` to visualize the interpretations

You should load all of them first:

```
import math
import os
import warnings
warnings.filterwarnings("ignore")
import mldatasets
import numpy as np
from sklearn import preprocessing
import tensorflow as tf
from tensorflow.keras.utils import get_file
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import FastGradientMethod,\
```

```
                         ProjectedGradientDescent, BasicIterativeMethod
from art.attacks.evasion import CarliniLInfMethod
from art.attacks.evasion import AdversarialPatchNumpy
from art.defences.preprocessor import SpatialSmoothing
from art.defences.trainer import AdversarialTrainer
from tqdm.notebook import tqdm
```

Let's check that TensorFlow has loaded the right version with `print(tf.__version__)`. The version should be 2.0 and above.

We should also disable eager execution and verify that it worked with the following command:

```
tf.compat.v1.disable_eager_execution()
print('Eager execution enabled:', tf.executing_eagerly())
```

The output should say that it's `False`.

> In TensorFlow, turning eager execution mode on means that it doesn't require a computational graph or a session. It's the default for TensorFlow 2.x and later but not in prior versions, so you need to disable it to avoid incompatibilities with code that was optimized for prior versions of TensorFlow.

Understanding and preparing the data

We will load the data into four NumPy arrays, corresponding to the training/test datasets. While we are at it, we will divide `X` face images by 255 because, that way, they will be of values between zero and one, which is better for deep learning models. We call this feature scaling. We will need to record the `min_` and `max_` for the training data because we will need these later:

```
X_train, X_test, y_train, y_test = mldatasets.load(
    "maskedface-net_thumbs_sampled", prepare=True
)
X_train, X_test = X_train / 255.0, X_test / 255.0
min_ = X_train.min()
max_ = X_train.max()
```

It's always important to verify our data when we load it to make sure it didn't get corrupted:

```
print('X_train dim:\t%s' % (X_train.shape,))
print('X_test dim:\t%s' % (X_test.shape,))
print('y_train dim:\t%s' % (y_train.shape,))
print('y_test dim:\t%s' % (y_test.shape,))
print('X_train min:\t%s' % (min_))
print('X_train max:\t%s' % (max_))
print('y_train labels:\t%s' % (np.unique(y_train)))
```

The preceding snippet will produce the following output, which tells us that the images have dimensions of 128 × 128 pixels and three channels (color). There are 16,800 training images and 4,200 test images. The labels only have a 1 in the second value, which indicates that they are not one-hot encoded. Indeed, by printing the unique values (np.unique(y_train)), we can tell that labels are represented as text: Correct for correctly masked, Incorrect for incorrectly masked, and None for no mask:

```
X_train dim:    (16800, 128, 128, 3)
X_test dim: (4200, 128, 128, 3)
y_train dim:    (16800, 1)
y_test dim: (4200, 1)
X_train min:    0.0
X_train max:    1.0
y_train labels: ['Correct' 'Incorrect' 'None']
```

Therefore, a preprocessing step we will need to perform is to **One-Hot Encode** (**OHE**) the y labels because we will need the OHE form to evaluate the model's predictive performance. Once we initialize the OneHotEncoder, we will need to fit it into the training data (y_train). We can also extract the categories from the encoder into a list (labels_l) to verify that it has all three:

```
ohe = preprocessing.OneHotEncoder(sparse=False)
ohe.fit(y_train)
labels_l = ohe.categories_[0].tolist()
print(labels_l)
```

For reproducibility's sake, always initialize your random seeds like this:

```
rand = 9
os.environ['PYTHONHASHSEED'] = str(rand)
tf.random.set_seed(rand)
np.random.seed(rand)
```

Making machine learning truly reproducible means also making it deterministic, which means that training with the same data will produce a model with the same parameters. Determinism is very difficult with deep learning and is often session-, platform-, and architecture-dependent. If you use an NVIDIA GPU, you can install a library called framework-reproducibility.

Many of the adversarial attack, defense, and evaluation methods we will study in this chapter are very resource-intensive, so if we used the entire test dataset with them, they could likely take many hours on a single method! For efficiency, it is strongly suggested to use samples of the test dataset. Therefore, we will create a medium 200-image sample (X_test_mdsample, y_test_mdsample) and a small 20-image sample (X_test_smsample, y_test_smsample) using np.random.choice:

```
sampl_md_idxs = np.random.choice(X_test.shape[0], 200, replace=False)
X_test_mdsample = X_test[sampl_md_idxs]
y_test_mdsample = y_test[sampl_md_idxs]
sampl_sm_idxs = np.random.choice(X_test.shape[0], 20, replace=False)
```

```
X_test_smsample = X_test[sampl_sm_idxs]
y_test_smsample = y_test[sampl_sm_idxs]
```

We have two sample sizes because some methods could take too long with a larger sample size. Now, let's take a peek at what images are in our datasets. In the preceding code, we have taken a medium and small sample of our test dataset. We will place each image of our small sample in a 4 × 5 grid with the class label above it, with the following code:

```python
plt.subplots(figsize=(15,12))
for s in range(20):
    plt.subplot(4, 5, s+1)
    plt.title(y_test_smsample[s][0], fontsize=12)
    plt.imshow(X_test_smsample[s], interpolation='spline16')
    plt.axis('off')
plt.show()
```

The preceding code plots the grid of images in *Figure 13.2*:

Figure 13.2: A small test dataset sample of masked and unmasked faces

Figure 13.2 depicts a variety of correctly and incorrectly masked and unmasked faces of all ages, genders, and ethnicities. Despite the variety, one thing to note about this dataset is that it only has light blue surgical masks represented, and images are mostly at a front-facing angle. Ideally, we would generate an even larger dataset with all colors and types of masks and augment it further with random rotations, shears, and brightness adjustments, either before or during training. These augmentations would make for a much more robust model. Nevertheless, we must differentiate between this general type of robustness and adversarial robustness.

Loading the CNN base model

You don't have to train the CNN base model, but the code to do so is provided nonetheless in the GitHub repository. The pretrained model has also been stored there. We can quickly load the model and output its summary like this:

```
model_path = get_file('CNN_Base_MaskedFace_Net.hdf5',\
    'https://github.com/PacktPublishing/Interpretable-Machine- \
    Learning-with-Python/blob/master/models/ \
    CNN_Base_MaskedFace_Net.hdf5?raw=true')
base_model = tf.keras.models.load_model(model_path)
base_model.summary()
```

The preceding snippet outputs the following summary:

```
Model: "CNN_Base_MaskedFaceNet_Model"

_____
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 16)      448            _____

_____
maxpool2d_1 (MaxPooling2D)   (None, 63, 63, 16)        0

_____
conv2d_2 (Conv2D)            (None, 61, 61, 32)        4640

_____
maxpool2d_2 (MaxPooling2D)   (None, 30, 30, 32)        0

_____
conv2d_3 (Conv2D)            (None, 28, 28, 64)        18496

_____
maxpool2d_3 (MaxPooling2D)   (None, 14, 14, 64)        0

_____
```

```
conv2d_4 (Conv2D)              (None, 12, 12, 128)        73856
_____
maxpool2d_4 (MaxPooling2D)     (None, 6, 6, 128)          0
_____
flatten_6 (Flatten)            (None, 4608)               0
_____
dense_1 (Dense)                (None, 768)                3539712
_____
dropout_6 (Dropout)            (None, 768)                0
_____
dense_2 (Dense)                (None, 3)                  2307
================================================================
Total params: 3,639,459
Trainable params: 3,639,459
Non-trainable params: 0
_____
```

The summary has pretty much everything we need to know about the model. It has four convolutional layers (Conv2D), each followed by a max pool layer (MaxPooling2D). It then has a Flatten layer and a fully connected layer (Dense). Then, there's more Dropout before the second Dense layer. Naturally, three neurons are in this final layer, corresponding to each class.

Assessing the CNN base classifier

We can evaluate the model using the test dataset with the evaluate_multiclass_mdl function. The arguments include the model (base_model), our test data (X_test), and the corresponding labels (y_test), as well as the class names (labels_l) and the encoder (ohe). Lastly, we don't need to plot the ROC curves since, given the high accuracy, they won't be very informative (plot_roc=False). This function returns the predicted labels and probabilities, which we can store as variables for later use:

```
y_test_pred, y_test_prob = mldatasets.evaluate_multiclass_mdl(
    base_model,
    X_test,
    y_test,
    labels_l,
    ohe,
    plot_conf_matrix=True,
    predopts={"verbose":1}
)
```

The preceding code generates *Figure 13.3*, with a confusion matrix and performance metrics for each class:


```
             precision    recall   f1-score    support

  Correct        0.998     0.998      0.998       1400
Incorrect        0.999     0.995      0.997       1400
     None        0.996     0.999      0.998       1400

 accuracy                             0.997       4200
macro avg        0.997     0.997      0.997       4200
weighted avg     0.997     0.997      0.997       4200
```

Figure 13.3: The confusion matrix and predictive performance metrics for the base classifier, evaluated on the test dataset

Even though the confusion matrix in *Figure 13.3* seems to suggest a perfect classification, pay attention to the circled areas. We can tell the model had some issues with misclassifying incorrectly masked faces once we see the recall (99.5%) breakdown.

Now, we can start attacking this model to assess how robust it actually is!

Learning about evasion attacks

There are six broad categories of adversarial attacks:

- **Evasion**: designing an input that can cause a model to make an incorrect prediction, especially when it wouldn't fool a human observer. It can either be targeted or untargeted, depending on the attacker's intention to fool the model into misclassifying a specific class (targeted) or, rather, misclassifying any class (untargeted). The attack methods can be white-box if the attacker has full access to the model and its training dataset, or black-box with only inference access. Gray-box sits in the middle. Black-box is always model-agnostic, whereas white and gray-box methods might be.

- **Poisoning**: injecting faulty training data or parameters into a model can come in many forms, depending on the attacker's capabilities and access. For instance, for systems with user-generated data, the attacker may be capable of adding faulty data or labels. If they have more access, perhaps they can modify large amounts of data. They can also adjust the learning algorithm, hyperparameters, or data augmentation schemes. Like evasion, poisoning can also be targeted and untargeted.

- **Inference:** extracting the training dataset through model inference. Inference attacks also come in many forms and can be used for espionage (privacy attacks) through membership inference, which confirms if one example (for instance, a specific person) was in the training dataset. Attribute inference ascertains if an example category (for instance, an ethnicity) was represented in the training data. Input inference (also known as model inversion) has attack methods to extract a training dataset from a model rather than guessing and confirming. These have broad privacy and regulatory implications, especially in medical and legal applications.

- **Trojaning:** this implants malicious functionality activated with a trigger during inference but requires retraining the model.

- **Backdooring:** similar to trojans but a backdoor remains, even when a model is retrained from scratch.

- **Reprogramming:** remote sabotaging of a model during training by sneaking in examples that are specifically designed to produce specific outputs. For instance, if you provide enough examples labeled as tiger shark where four small black squares are always in the same place, the model will learn that that is a tiger shark, regardless of what it is, thus intentionally forcing the model to overfit.

The first three are the most studied forms of adversarial attacks. Attacks can be further subcategorized once we split them by stage and goal (see *Figure 13.4*). The stage is when the attack is perpetrated because it can impact the model training or its inference, and the goal is what the attacker hopes to gain from it. This chapter will only deal with evasion sabotage attacks because we expect hospital visitors, patients, and personnel to occasionally sabotage the production model:

Stage		Goal		
		Espionage	Sabotage	Fraud
	Training	Inference (by poisoning)	Trojaning	
			Poisoning	
			Backdooring	
	Production	Inference	Reprogramming	
			Evasion	

Figure 13.4: Table of adversarial attack category methods by stage and goal

Even though we use white-box methods to attack, defend, and evaluate a model's robustness, we don't expect attackers to have this level of access. We will only use white-box methods because we have full access to the model. In other circumstances, such as a bank surveillance system with a thermal imaging system and a corresponding model to detect perpetrators, we could expect professional attackers to use black-box methods to find vulnerabilities! So, as defenders of this system, we would be wise to try the very same attack methods.

The library we will use for adversarial robustness is called the **Adversarial Robustness Toolbox** (**ART**), and it's supported by the **LF AI & Data Foundation** – the same folks that support other open-source projects such as AIX360 and AIF360, explored in *Chapter 11, Bias Mitigation and Causal Inference Methods*. ART requires that attacked models are abstracted in an estimator or classifier, even if it's a black-box one. We will use `KerasClassifier` for most of this chapter except for the last section, in which we will use `TensorFlowV2Classifier`. Initializing an ART classifier is simple. You must specify the `model`, and sometimes there are other required attributes. For `KerasClassifier`, all remaining attributes are optional, but it is recommended you use `clip_values` to specify the range of the features. Many attacks are input permutations, so knowing what input values are allowed or feasible is essential:

```
base_classifier = KerasClassifier(
    model=base_model, clip_values=(min_, max_)
)
y_test_mdsample_prob = np.max(
    y_test_prob[sampl_md_idxs], axis=1
)
y_test_smsample_prob = np.max(
    y_test_prob[sampl_sm_idxs], axis=1
)
```

In the preceding code, we also prepare two arrays with probabilities for the predicted class of the medium and small samples. It is entirely optional, but these assist in placing the predicted probability next to the predicted label when plotting some examples.

Fast gradient sign method attack

One of the most popular attack methods is the **Fast Gradient Sign Method** (**FSGM** or **FGM**). As the name implies, it leverages a deep learning model's gradient to find adversarial examples. It performs small perturbations on the pixels of the input image, either additions or subtractions. Which one to use depends on the gradient's sign, which indicates what direction would increase or decrease the loss according to the pixel's intensity.

As with all ART attack methods, you first initialize it by providing the ART estimator or classifier. `FastGradientMethod` also requires an attack step size `eps`, which will condition the attack strength. Incidentally, `eps` stands for epsilon (ϵ), which represents error margins or infinitesimal approximation errors. A small step size will cause pixel intensity changes to be less visible, but it will also misclassify fewer examples. A larger step size will cause more examples to be misclassified with more visible changes:

```
attack_fgsm = FastGradientMethod(base_classifier, eps=0.1)
```

After initializing, the next step is to `generate` the adversarial examples. The only required attribute is original examples (`X_test_mdsample`). Please note that FSGM can be targeted, so there's an optional `targeted` attribute in the initialization, but you would also need to provide corresponding labels in the generation. This attack is untargeted because the attacker's intent is to sabotage the model:

```
X_test_fgsm = attack_fgsm.generate(X_test_mdsample)
```

Generating the adversarial examples with FSGM is quick, unlike other methods, hence the "Fast"!

Now, we will do two things in one swoop. First, evaluate the adversarial examples (`X_test_fgsm`) against our base classifier's model (`base_classifier.model`) with `evaluate_multiclass_mdl`. Then we can employ `compare_image_predictions` to plot a grid of images, contrasting the randomly selected adversarial examples (`X_test_fgsm`) against the original ones (`X_test_mdsample`) and their corresponding predicted labels (`y_test_fgsm_pred, y_test_mdsample`) and probabilities (`y_test_fgsm_prob, y_test_mdsample_prob`). We customize the titles and limit the grid to four examples (`num_samples`). By default, `compare_image_predictions` only compares misclassifications but an optional attribute, `use_misclass`, can be set to `False` to compare correct classifications:

```
y_test_fgsm_pred, y_test_fgsm_prob =\
    mldatasets.evaluate_multiclass_mdl(\
        base_classifier.model, X_test_fgsm, y_test_mdsample,\
        labels_l, ohe, plot_conf_matrix=False, plot_roc=False
    )
y_test_fgsm_prob = np.max(y_test_fgsm_prob, axis=1)
mldatasets.compare_image_predictions(
    X_test_fgsm, X_test_mdsample, y_test_fgsm_pred,\
    y_test_mdsample.flatten(), y_test_fgsm_prob,\
    y_test_mdsample_prob, title_mod_prefix="Attacked:",\
    title_difference_prefix="FSGM Attack Average Perturbation:",\
    num_samples=4
)
```

The preceding code outputs a table first, which shows that the model has only 44% accuracy with FSGM-attacked examples! And even though it wasn't a targeted attack, it was most effective toward correctly masked faces. So hypothetically, if perpetrators managed to cause this level of signal distortion or interference, they would severely undermine the security companies' ability to monitor mask compliance.

The code also outputs *Figure 13.5*, which shows some misclassifications caused by the FSGM attack. The attack pretty much evenly distributed noise throughout the images. It also shows that the image was only modified by a mean absolute error of 0.092, and since pixel values range between 0 and 1, this means 9.2%. If you were to calibrate attacks so that they are less detectable but still impactful, you must note that an eps of 0.1 causes a 9.2% mean absolute perturbation, which reduces accuracy to 44%:
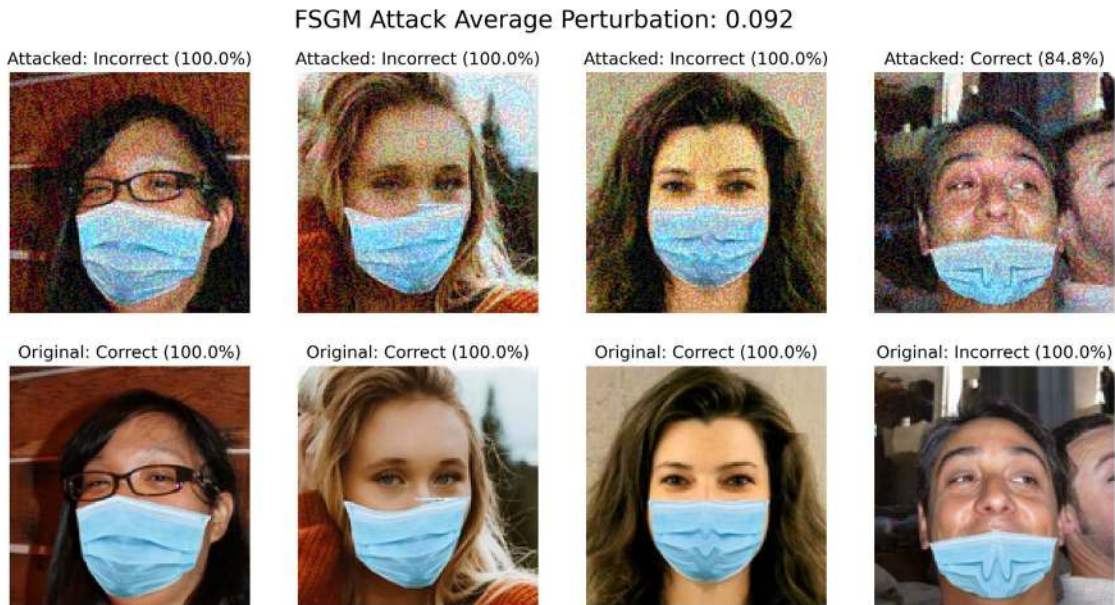
Figure 13.5: Plot comparing FSGM-attacked versus the original images for the base classifier

Speaking of less detectable attacks, we will now learn about Carlini and Wagner attacks.

Carlini and Wagner infinity norm attack

In 2017, **Carlini and Wagner** (**C&W**) employed three **norm-based distance metrics:** L_0, L_2, and L_∞, measuring the difference between the original and adversarial example. In other papers, these metrics had already been discussed, including the FSGM one. The innovation introduced by C&W was how these metrics were leveraged, using a gradient descent-based optimization algorithm designed to approximate a loss function minima. Specifically, to avoid getting stuck at a local minimum, they use multiple starting points in the gradient descent. And so that the process "yields a valid image," it evaluates three methods to box-constrain the optimization problem. In this case, we want to find an adversarial example where the distances between that example and the original image are minimal, while also remaining realistic.

All three C&W attacks (L_0, L_2, and L_∞) use the Adam optimizer to quickly converge. Their main difference is the distance metric, of which L_∞ is arguably the best one. It's defined as follows:

$$L_\infty = ||x - x'||_\infty = max(|x_1 - x'_1|, \ldots, |x_n - x'_n|)$$

And because it's the maximum distance to any coordinate, you make sure that the adversarial example is not just "on average" minimally different but also not too different anywhere in the feature space. That's what would make an attack less detectable!

Initializing C&W infinity norm attacks and generating adversarial examples with them is similar to FSGM. To initialize `CarliniLInfMethod`, we define optionally a `batch_size` (the default is `128`). Then, to `generate` an untargeted adversarial attack, the same applies as with FSGM. Only X is needed when untargeted, and y when targeted:

```
attack_cw = CarliniLInfMethod(
    base_classifier, batch_size=40
)
X_test_cw = attack_cw.generate(X_test_mdsample)
```

We will now evaluate the C&W adversarial examples (`X_test_cw`) just as we did with FSGM. It's exactly the same code, only with `fsgm` replaced with `cw` and different titles in `compare_image_predictions`. Just as with FSGM, the following code will yield a classification report and grid of images (*Figure 13.6*):

```
y_test_cw_pred, y_test_cw_prob =\
    mldatasets.evaluate_multiclass_mdl(
        base_classifier.model, X_test_cw, y_test_mdsample, labels_l,\
        ohe, plot_conf_matrix=False, plot_roc=False
    )
y_test_cw_prob = np.max(y_test_cw_prob, axis=1)
mldatasets.compare_image_predictions(
    X_test_cw,\
    X_test_mdsample, y_test_cw_pred,\
    y_test_mdsample.flatten(), y_test_cw_prob,\
    y_test_mdsample_prob, title_mod_prefix="Attacked:",\
    title_difference_prefix="C&W Inf Attack Average Perturbation",\
    num_samples=4
)
```

As outputted by the preceding code, the C&W adversarial examples have a 92% accuracy with our base model. The drop is sufficient to render the model useless for its intended purpose. If the attacker disturbed a camera's signal just enough, they could achieve the same results. And, as you can tell by *Figure 13.6*, the perturbation of 0.3% is tiny compared to FSGM, but it was sufficient to misclassify 8%, including the four in the grid that seem apparent to the naked eye:
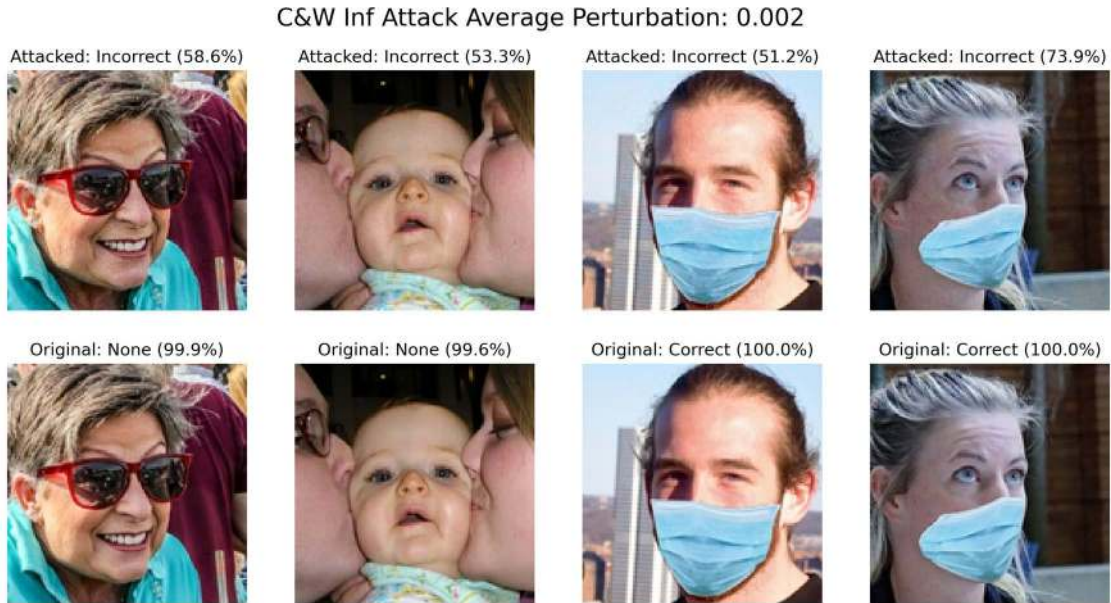
Figure 13.6: Plot comparing C&W infinity norm-attacked versus the original images for the base classifier

Sometimes it doesn't matter if an attack goes undetected or not. The point of it is to make a statement, and that's what adversarial patches can do.

Targeted adversarial patch attack

Adversarial Patches (APs) are a robust, universal, and targeted method. You generate a patch you can either superimpose on an image, or print and physically place in a scene to trick a classifier into ignoring everything else in the scene. It is designed to work under a wide variety of conditions and transformations. Unlike other adversarial example generation approaches, there's no intention of camouflaging the attack because, essentially, you replace a detectable portion of the scene with the patch. The method works by leveraging a variant of **Expectation Over Transformation** (**EOT**), which trains images over transformations of a given patch on different locations of an image. What it learns is the patch that fools the classifier the most, given the training examples.

This method requires more parameters and steps than FSGM and C&W. For starters, we will use `AdversarialPatchNumpy`, which is the variant that works with any neural network image or video classifier. There's also one for TensorFlow v2, but our base classifier is a `KerasClassifier`. The first argument is the classifier (`base_classifier`), and the other ones we will define are optional but highly recommended. The scaling ranges `scale_min` and `scale_max` are particularly important because they define how big can patches be in relation to the images – in this case, we want to test no smaller than 40% and no larger than 70%. Besides that, it makes sense to define a target class (`target`). In this case, we want the patch to target the "Correct" class. For the `learning_rate` and max iterations (`max_iter`), we use the defaults but note that these can be tuned to improve patch adversarial effectiveness:

```python
attack_ap = AdversarialPatchNumpy(
    base_classifier, scale_min=0.4, scale_max=0.7,\
    learning_rate=5., max_iter=500,\
    batch_size=40, target=0
)
```

We don't want the patch generation algorithm to waste time testing patches everywhere in images, so we can direct this effort by using a Boolean mask. This mask tells it where it can center the patch. To make the mask, we start by creating an array of zeros, 128 × 128. Then we place ones in the rectangular area between pixels 80–93 and 45–84, which loosely corresponds to cover the center of the mouth area in most of the images. Lastly, we expand the array's dimensions so that it's (`1, W, H`) and convert it to a Boolean. Then we can proceed to `generate` patches using the small-size test dataset samples and the mask:

```python
placement_mask = np.zeros((128,128))
placement_mask[80:93,45:83] = 1
placement_mask = np.expand_dims(placement_mask, axis=0).astype(bool)
patch, patch_mask = attack_ap.generate(
    x=X_test_smsample,
    y=ohe.transform(y_test_smsample),
    mask=placement_mask
)
```

We can now plot the patch with the following snippet:

```python
plt.imshow(patch * patch_mask)
```

The preceding code produced the image in *Figure 13.7*. As expected, it has plenty of the shades of blue found in masks. It also has bright red and yellow hues, mostly missing from training examples, which confuse the classifier:
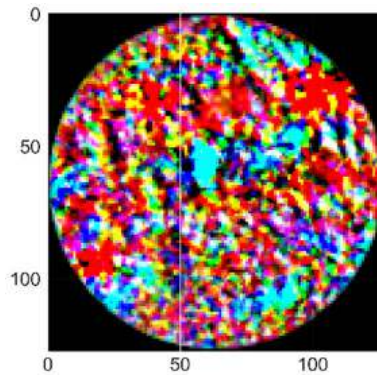
Figure 13.7: AP generated image to misclassify as correctly masked

Unlike other methods, `generate` didn't produce adversarial examples but a single patch, which is an image we can then place on top of images to create adversarial examples. This task is performed with `apply_patch`, which takes the original examples `X_test_smsample` and a scale; we will use 55%. It is also recommended to use a `mask` that will make sure the patch is applied where it makes more sense – in this case, in the area around the mouth:

```
X_test_ap = attack_ap.apply_patch(
    X_test_smsample,
    scale=0.55,
    mask=placement_mask
)
```

Now it's time to evaluate our attack and examine some misclassifications. We will do exactly as before and reuse the code that produced *Figure 13.5* and *Figure 13.7*, except we replace the variables so that they have `ap` and a corresponding title:

```
y_test_ap_pred, y_test_ap_prob =\
    mldatasets.evaluate_multiclass_mdl(
        base_classifier.model, X_test_ap, y_test_smsample,
        labels_l, ohe, plot_conf_matrix=False, plot_roc=False
    )
y_test_ap_prob = np.max(y_test_ap_prob, axis=1)
mldatasets.compare_image_predictions(
    X_test_ap, X_test_smsample, y_test_ap_pred,\
    y_test_smsample.flatten(), y_test_ap_prob,
    y_test_smsample_prob, title_mod_prefix="Attacked:",\
    title_difference_prefix="AP Attack Average Perturbation:", num_samples=4
)
```

The preceding code yields the accuracy result of our attack at 65%, which is quite good considering how few examples it was trained on. AP needs more data than other methods. Targeted attacks, in general, need more examples to understand how to best target one class. The preceding code also produced the grid of images in *Figure 13.8*, which demonstrates how, hypothetically, if people walked around holding a cardboard patch in front of their face, they could easily fool the model:
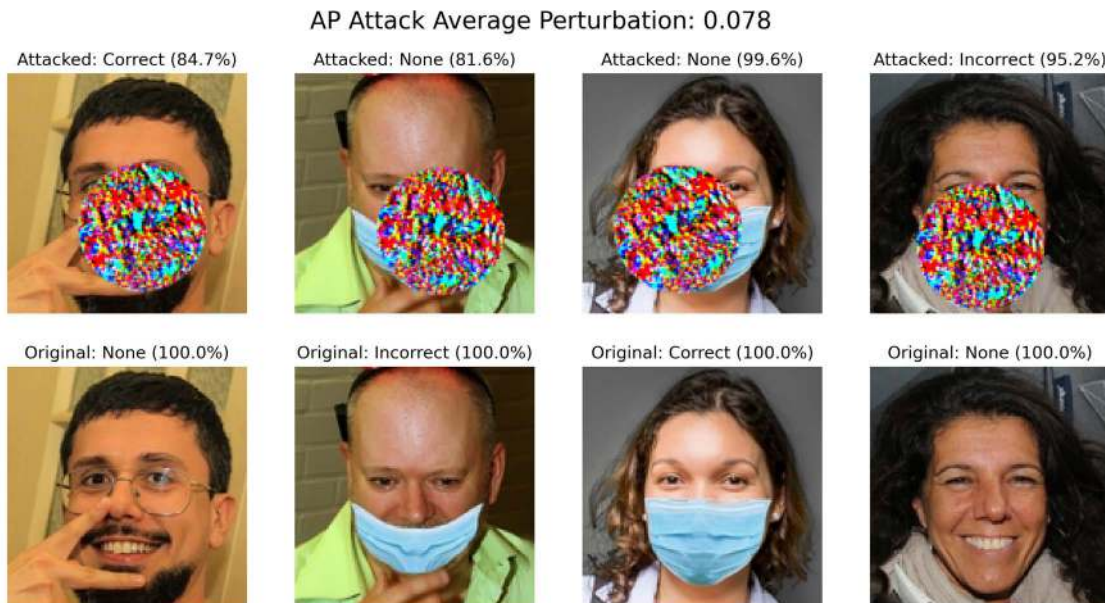
Figure 13.8: Plot comparing AP-attacked versus the original images for base classifier

So far, we have studied three attack methods but haven't yet tackled how to defend against these attacks. We will explore a couple of solutions next.

Defending against targeted attacks with preprocessing

There are five broad categories of adversarial defenses:

- **Preprocessing**: changing the model's inputs so that they are harder to attack.
- **Training**: training a new robust model that is designed to overcome attacks.
- **Detection**: detecting attacks. For instance, you can train a model to detect adversarial examples.
- **Transformer**: modifying model architecture and training so that it's more robust – this may include techniques such as distillation, input filters, neuron pruning, and unlearning.
- **Postprocessing**: changing model outputs to overcome production inference or model extraction attacks.

Only the first four defenses work with evasion attacks, and in this chapter, we will only cover the first two: **preprocessing** and **adversarial training**. FGSM and C&W can be defended easily with either of these, but an AP is tougher to defend against, so it might require a stronger **detection** or **transformer** method.

Before we defend, we must create a targeted attack. We will employ **Projected Gradient Descent** (**PGD**), which is a strong attack very similar in output to FSGM – that is, it produces noisy images. We won't explain PGD in detail here but what is important to note is, like FSGM, it is regarded as a **first-order adversary** because it leverages first-order information about a network (due to gradient descent). Also, experiments prove that robustness against PGD ensures robustness against any first-order adversary. Specifically, PGD is a strong attack, so it makes for conclusive benchmarks.

To create a targeted attack against the correctly masked class, it's best that we only select examples that aren't correctly masked (`x_test_notmasked`) and their corresponding labels (`y_test_notmasked`) and predicted probabilities (`y_test_notmasked_prob`). Then, we want to create an array with the class (`Correct`) that we want to generate adversarial examples for (`y_test_masked`):

```python
not_masked_idxs = np.where(y_test_smsample != 'Correct')[0]
X_test_notmasked = X_test_smsample[not_masked_idxs]
y_test_notmasked = y_test_smsample[not_masked_idxs]
y_test_notmasked_prob = y_test_smsample_prob[not_masked_idxs]
y_test_masked = np.array(
    ['Correct'] * X_test_notmasked.shape[0]
).reshape(-1,1)
```

We initialize `ProjectedGradientDescent` as we did with FSGM, except we will set the maximum perturbation (`eps`), attack step size (`eps_step`), maximum iterations (`max_iter`), and `targeted=True`. Precisely because it is targeted, we will set both `X` and `y`:

```python
attack_pgd = ProjectedGradientDescent(
    base_classifier, eps=0.3, eps_step=0.01,\
    max_iter=40, targeted=True
)
X_test_pgd = attack_pgd.generate(
    X_test_notmasked, y=ohe.transform(y_test_masked)
)
```

Now let's evaluate the PGD attack as we did before, but this time, let's plot the confusion matrix (`plot_conf_matrix=True`):

```python
y_test_pgd_pred, y_test_pgd_prob =\
    mldatasets.evaluate_multiclass_mdl(
        base_classifier.model, X_test_pgd, y_test_notmasked,\
        labels_l, ohe, plot_conf_matrix=True, plot_roc=False
    )
y_test_pgd_prob = np.max(y_test_pgd_prob, axis=1)
```

The preceding snippet produces the confusion matrix in *Figure 13.9*. The PGD attack was so effective that it produced an accuracy of 0%, making all unmasked and incorrectly masked examples appear to be masked:
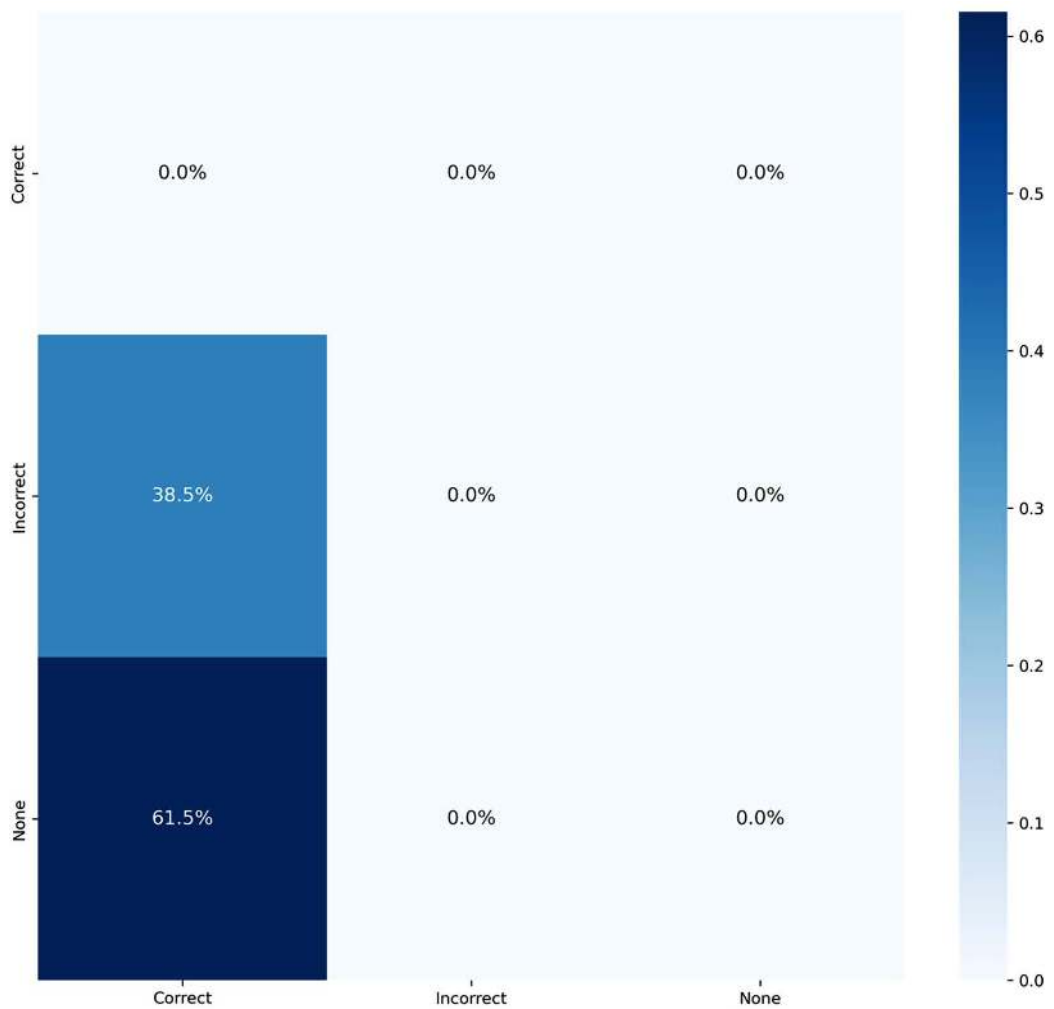
Figure 13.9: Confusion matrix for PGD attacked examples evaluated against the base classifier

Next, let's run `compare_image_prediction` to see some random misclassifications:

```
mldatasets.compare_image_predictions(
    X_test_pgd, X_test_notmasked, y_test_pgd_pred,\
    y_test_notmasked.flatten(), y_test_pgd_prob,\
    y_test_smsample_prob, title_mod_prefix="Attacked:",\
    num_samples=4, title_difference_prefix="PGD Attack Average Perturbation:"
)
```

The preceding code plots the grid of images in *Figure 13.10*. The mean absolute perturbation is the highest we've seen so far at 14.7%, and all unmasked faces in the grid are classified as correctly masked:
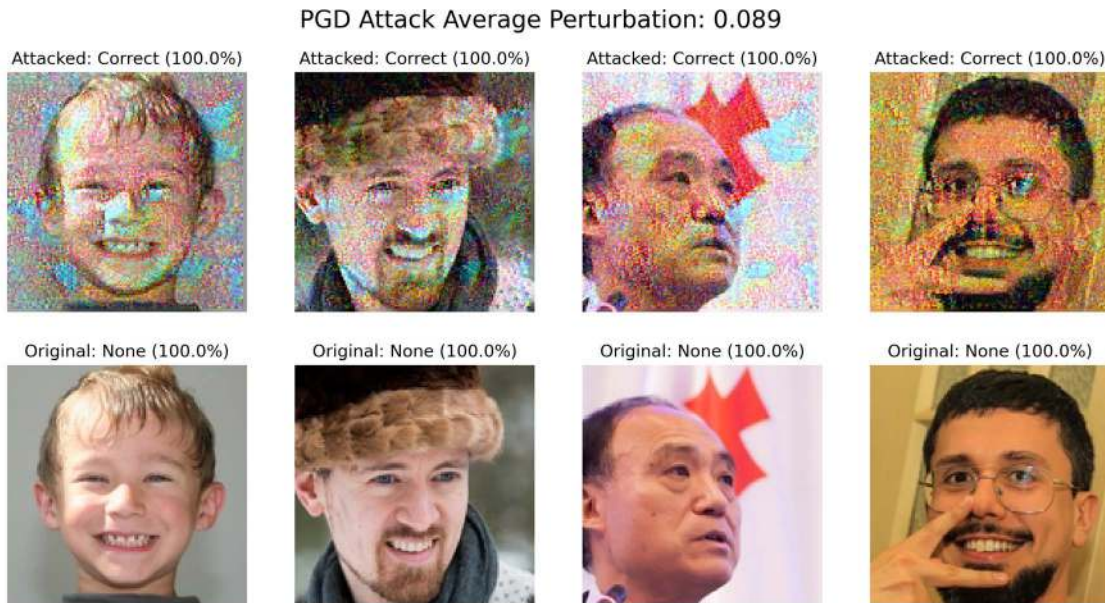
Figure 13.10: Plot comparing PGD-attacked versus original images for the base classifier

The accuracy cannot get worse, and the images are grainy beyond repair. So how can we combat noise? If you recall, we have dealt with this problem before. In *Chapter 7, Visualizing Convolutional Neural Networks*, **SmoothGrad** improved saliency maps by averaging the gradients. It's a different application but the same principle – just as with a human, a noisy saliency map is more challenging to interpret than a smooth one, and a grainy image is much more challenging for a model to interpret than a smooth one.

Spatial smoothing is just a fancy way of saying blur! However, what's novel about it being introduced as an adversarial defense method is that the proposed implementation (SpatialSmoothing) calls for using the median and not the mean in a sliding window. The window_size is configurable, and it is recommended to adjust it where it is most useful as a defense. Once the defense has been initialized, you plug in the adversarial examples (X_test_pgd). It will output spatially smoothed adversarial examples (X_test_pgd_ss):

```
defence_ss = SpatialSmoothing(window_size=11)
X_test_pgd_ss, _ = defence_ss(X_test_pgd)
```

Now we can take the blurred adversarial examples produced and evaluate them as we did before – first, with evaluate_multiclass_mdl to get predicted labels (y_test_pgd_ss_pred) and probabilities (y_test_pgd_ss_prob) and the output of some predictive performance metrics. With compare_image_predictions to plot a grid of images, let's use use_misclass=False to compare properly classified images – in other words, the adversarial examples that were defended successfully:

```
y_test_pgd_ss_pred, y_test_pgd_ss_prob =\
    mldatasets.evaluate_multiclass_mdl(
        base_classifier.model, X_test_pgd_ss,\
        y_test_notmasked, labels_l, ohe,\
        plot_conf_matrix=False, plot_roc=False
)
y_test_pgd_ss_prob = np.max(y_test_pgd_ss_prob, axis=1)

mldatasets.compare_image_predictions(
    X_test_pgd_ss, X_test_notmasked, y_test_pgd_ss_pred,\
    y_test_notmasked.flatten(), y_test_pgd_ss_prob,\
    y_test_notmasked_prob, use_misclass=False,\
    title_mod_prefix="Attacked+Defended:", num_samples=4,\
    title_difference_prefix="PGD Attack & Defended Average:"
)
```

The preceding code yields an accuracy of 54%, which is much better than 0% before the spatial smoothing defense. It also produces *Figure 13.11*, which demonstrates how blur effectively thwarted the PGD attack. It even halved the mean absolute perturbation!
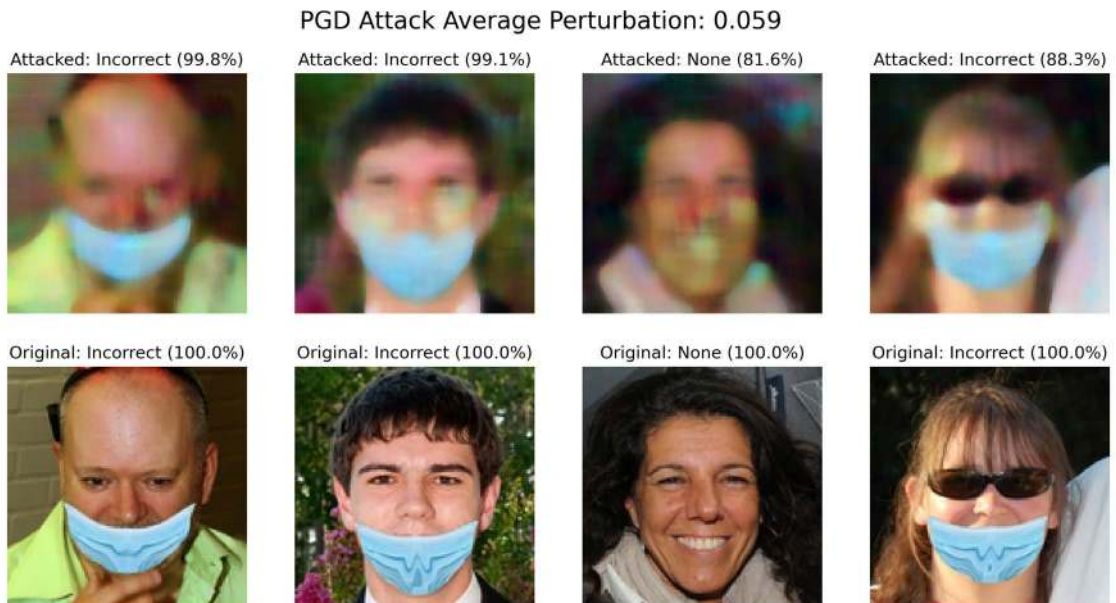
Figure 13.11: Plot comparing spatially smoothed PGD-attacked images versus the original images for the base classifier

Next, we will try another defense method in our toolbox: adversarial training!

Shielding against any evasion attack by adversarial training of a robust classifier

In *Chapter 7*, *Visualizing Convolutional Neural Networks*, we identified a garbage image classifier that would likely perform poorly in the intended environment of a municipal recycling plant. The abysmal performance on out-of-sample data was due to the classifier being trained on a large variety of publicly available images that don't match the expected conditions, or the characteristics of materials that are processed by a recycling plant. The chapter's conclusion called for training a network with images that represent their intended environment to make for a more robust model.

For model robustness, training data variety is critical, but only if it represents the intended environment. In statistical terms, it's a question of using samples for training that accurately depict the population so that a model learns to classify them correctly. For adversarial robustness, the same principles apply. If you augment data to include plausible examples of adversarial attacks, the model will learn to classify them. That's what adversarial training is in a nutshell.

Machine learning researchers in the adversarial robustness field suggest this form of defense is very effective against any kind of evasion attack, essentially shielding it. That being said, it's not impervious. Its effectiveness is contingent on using the right kind of adversarial examples in training, the optimal hyperparameters, and so forth. There are some guidelines outlined by researchers, such as increasing the number of neurons in the hidden layers and using PGD or BIM to produce adversarial examples for the training. **BIM** stands for **Basic Iterative Method**. It's like FSGM but not fast because it iterates to approximate the best adversarial example within a ϵ-neighborhood for the original image. The eps attribute bounds this neighborhood.

Training a robust model can be very resource-intensive. It is not required because we can download one already trained for us, but it's important to understand how to perform this with ART. We will explain these steps to give the option of completing the model training with ART. Otherwise, just skip the steps and download the trained model. The robust_model is very much like the base_model except we use equal-sized filters in the four convolutional (Conv2D) layers. We do this to decrease complexity to counter the complexity we add by quadrupling the neurons in the first hidden (Dense) layer, as suggested by the machine learning researchers:

```
robust_model = tf.keras.models.Sequential([
    tf.keras.layers.InputLayer(input_shape=X_train.shape[1:]),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
```

```python
    tf.keras.layers.Dense(3072, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(3, activation='softmax')
], name='CNN_Robust_MaskedFaceNet_Model')

robust_model.compile(
    optimizer=tf.keras.optimizers.Adam(lr=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy'])
robust_model.summary()
```

The summary() in the preceding code outputs the following. You can see that trainable parameters total around 3.6 million – similar to the base model:

```
Model: "CNN_Robust_MaskedFaceNet_Model"
_____
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 32)      896

maxpool2d_1 (MaxPooling2D)   (None, 63, 63, 32)        0

conv2d_2 (Conv2D)            (None, 61, 61, 32)        9248

maxpool2d_2 (MaxPooling2D)   (None, 30, 30, 32)        0

conv2d_3 (Conv2D)            (None, 28, 28, 32)        9248

maxpool2d_3 (MaxPooling2D)   (None, 14, 14, 32)        0

conv2d_4 (Conv2D)            (None, 12, 12, 32)        9248

maxpool2d_4 (MaxPooling2D)   (None, 6, 6, 32)          0

flatten (Flatten)            (None, 1152)              0

dense_1 (Dense)              (None, 3072)              3542016

dropout (Dropout)            (None, 3072)              0

dense_2 (Dense)              (None, 3)                 9219
=================================================================
```

```
 Total params: 3,579,875
 Trainable params: 3,579,875
 Non-trainable params: 0
 _____
```

Next, we can adversarially train the model by first initializing a new `KerasClassifier` with the `robust_model`. Then, we initialize a `BasicIterativeMethod` attack on this classifier. Lastly, we initialize `AdversarialTrainer` with the `robust_classifier` and the BIM attack and `fit` it. Please note that we saved the BIM attack into a variable called `attacks` because this could be a list of ART attacks instead of a single one. Also, note that `AdversarialTrainer` has an attribute called `ratio`. This attribute determines what percentage of the training examples are adversarial examples. This percentage dramatically impacts the effectiveness of adversarial attacks. If it's too low, it might not perform well with adversarial examples, and if it's too high, it might perform less effectively with non-adversarial examples. If we run the `trainer`, it will likely take many hours to complete, so don't get alarmed:

```python
robust_classifier = KerasClassifier(
    model=robust_model, clip_values=(min_, max_)
)
attacks = BasicIterativeMethod(
    robust_classifier, eps=0.3, eps_step=0.01, max_iter=20
)
trainer = AdversarialTrainer(
    robust_classifier, attacks, ratio=0.5
)
trainer.fit(
    X_train, ohe.transform(y_train), nb_epochs=30, batch_size=128
)
```

If you didn't train the `robust_classifier`, you can download a pretrained `robust_model` and initialize the `robust_classifier` with it like this:

```python
model_path = get_file(
    'CNN_Robust_MaskedFace_Net.hdf5',
    'https://github.com/PacktPublishing/Interpretable-Machine- \
    Learning-with-Python/blob/master/models/ \
    CNN_Robust_MaskedFace_Net.hdf5?raw=true'
)
robust_model = tf.keras.models.load_model(model_path)
robust_classifier = KerasClassifier(
    model=robust_model, clip_values=(min_, max_)
)
```

Now, let's evaluate the `robust_classifier` against the original test dataset using `evaluate_multiclass_mdl`. We set `plot_conf_matrix=True` to see the confusion matrix:

```
y_test_robust_pred, y_test_robust_prob =\
mldatasets.evaluate_multiclass_mdl(
    robust_classifier.model, X_test, y_test, labels_l, ohe,\
    plot_conf_matrix=True, predopts={"verbose":1}
)
```

The preceding code outputs the confusion matrix and performance metrics in *Figure 13.12*. It's 1.8% less accurate than the base classifier. Most of the misclassifications are with correctly masked faces getting classified as incorrectly masked. There's certainly a trade-off when choosing a 50% adversarial example ratio, or perhaps we can tune the hyperparameters or the model architecture to improve this:


```
              precision    recall  f1-score   support

     Correct      0.962     0.996     0.979      1400
   Incorrect      0.980     0.957     0.969      1400
        None      0.994     0.983     0.989      1400

    accuracy                          0.979      4200
   macro avg      0.979     0.979     0.979      4200
weighted avg      0.979     0.979     0.979      4200
```
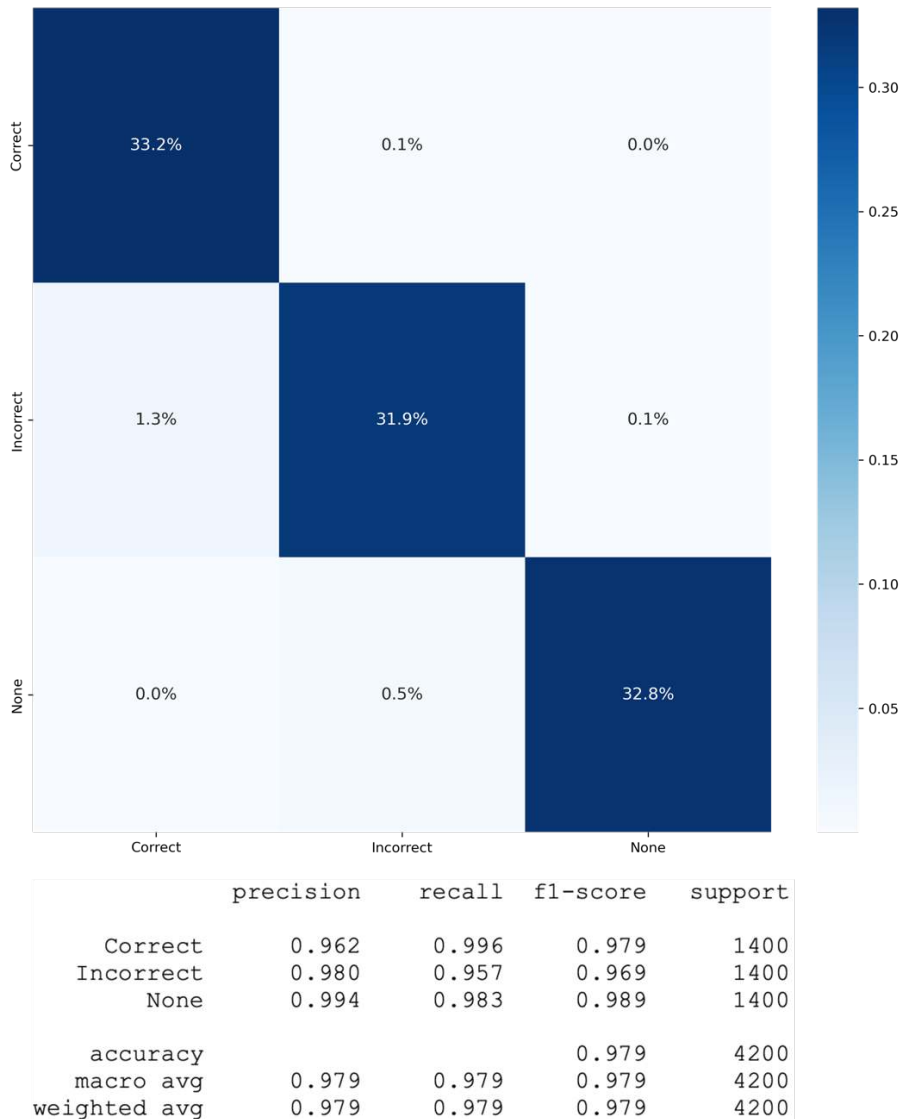
Figure 13.12: Robust classifier confusion metrics and performance metrics

Let's see how the robust model fares against adversarial attacks. Let's use `FastGradientMethod` again, but this time, replace `base_classifier` with `robust_classifier`:

```
attack_fgsm_robust = FastGradientMethod(
    robust_classifier, eps=0.1
)
X_test_fgsm_robust = attack_fgsm_robust.generate(X_test_mdsample)
```

Next, we can employ `evaluate_multiclass_mdl` and `compare_image_predictions` to measure and observe the effectiveness of our attack, but this time against the `robust_classifier`:

```
y_test_fgsm_robust_pred, y_test_fgsm_robust_prob =\
    mldatasets.evaluate_multiclass_mdl(
        robust_classifier.model, X_test_fgsm_robust,\
        y_test_mdsample, labels_l, ohe,\
        plot_conf_matrix=False, plot_roc=False
    )
y_test_fgsm_robust_prob = np.max(
    y_test_fgsm_robust_prob, axis=1
)
mldatasets.compare_image_predictions(
    X_test_fgsm_robust, X_test_mdsample,
    y_test_fgsm_robust_pred, num_samples=4,\
    y_test_mdsample.flatten(), y_test_fgsm_robust_prob,\
    y_test_mdsample_prob, title_mod_prefix="Attacked:",\
    title_difference_prefix="FSGM Attack Average Perturbation:"
)
```

The preceding snippet outputs some performance metrics, which evidenced a 95.5% accuracy. If you compare how an equally strengthened FSGM attack fared against the `base_classifier`, it yielded a 44% accuracy. That was quite an improvement! The preceding code also produces the image grid in *Figure 13.13*. You can tell how the FSGM attack against the robust model makes less grainy and more patchy images. On average, they are less perturbed than they were against the base model because so few of them were successful, but those that were significantly degraded. It appears as if the FSGM reduced their color depth from millions of possible colors (24+ bits) to 256 (8-bit) or 16 (4-bit) colors. Of course, an evasion attack can't actually do that, but what happened was that the FSGM algorithm converged at the same shades of blue, brown, red, and orange that could fool the classifier! Other shades remain unaltered:
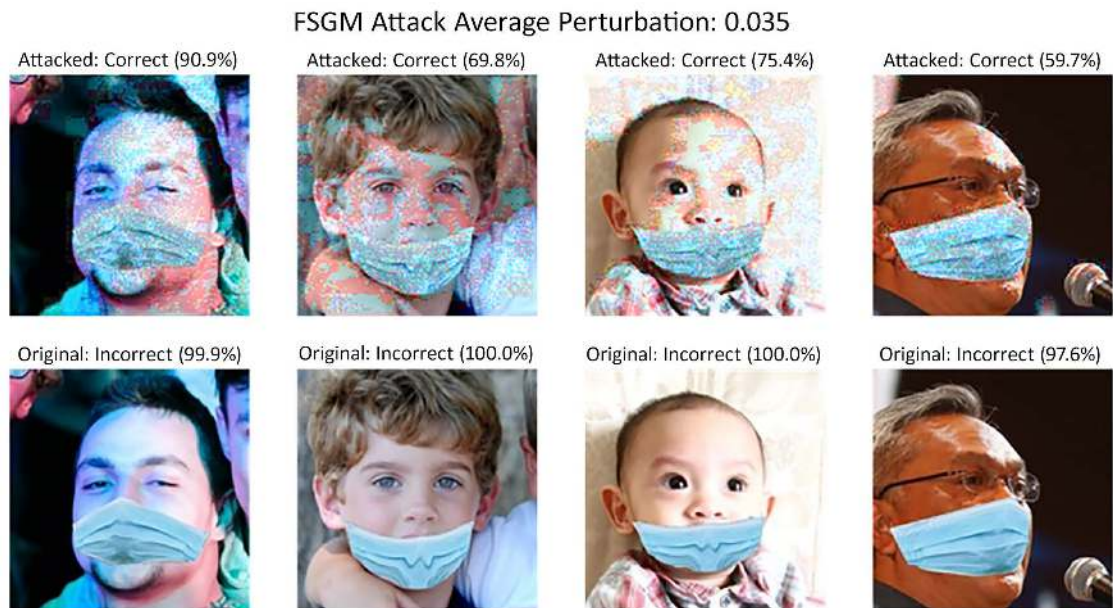
Figure 13.13: Plot comparing FSGM-attacked versus the original images for the robust classifier

So far, we have evaluated the robustness of models but only against one attack strength, without factoring in possible defenses, thus evaluating its robustness. In the next section, we will study a method that does this.

Evaluating adversarial robustness

It's necessary to test your systems in any engineering endeavor to see how vulnerable they are to attacks or accidental failures. However, security is a domain where you must stress-test your solutions to ascertain what level of attacks are needed to make your system break down beyond an acceptable threshold. Furthermore, figuring out what level of defense is needed to curtail an attack is useful information too.

Comparing model robustness with attack strength

We now have two classifiers we can compare against an equally strengthened attack, and we try different attack strengths to see how they fare across all of them. We will use FSGM because it's fast, but you could use any method!

The first attack strength we can assess is no attack strength. In other words, what is the classification accuracy against the test dataset with no attack? We already had stored the predicted labels for both the base (`y_test_pred`) and robust (`y_test_robust_pred`) models, so this is easy to obtain with scikit-learn's `accuracy_score` metric:

```python
accuracy_base_0 = metrics.accuracy_score(
    y_test, y_test_pred
)
accuracy_robust_0 = metrics.accuracy_score(
    y_test, y_test_robust_pred
)
```

We can now iterate across a range of attack strengths (eps_range) between 0.01 and 0.9. Using `linspace`, we can generate 9 values between 0.01 and 0.09 and 9 values between 0.1 and 0.9, and `concatenate` them into a single array. We will test attacks for these 18 eps values by `for`-looping through all of them, attacking each model, and retrieving the post-attack accuracies with `evaluate`. The respective accuracies are appended to two lists (`accuracy_base`, `accuracy_robust`). And after the `for` loop, we prepend 0 to the `eps_range` to account for the accuracies prior to any attacks:

```python
eps_range = np.concatenate(
    (np.linspace(0.01, 0.09, 9), np.linspace(0.1, 0.9, 9)), axis=0
).tolist()
accuracy_base = [accuracy_base_0]
accuracy_robust = [accuracy_robust_0]
for eps in tqdm(eps_range, desc='EPS'):
    attack_fgsm.set_params(**{'eps': eps})
    X_test_fgsm_base_i =attack_fgsm.generate(X_test_mdsample)
    _, accuracy_base_i =\
    base_classifier.model.evaluate(
        X_test_fgsm_base_i, ohe.transform(y_test_mdsample)
    )
    attack_fgsm_robust.set_params(**{'eps': eps})
    X_test_fgsm_robust_i=attack_fgsm_robust.generate(
        X_test_mdsample
    )
    _, accuracy_robust_i =\
        robust_classifier.model.evaluate(
            X_test_fgsm_robust_i, ohe.transform(y_test_mdsample)
        )
    accuracy_base.append(accuracy_base_i)
    accuracy_robust.append(accuracy_robust_i)
eps_range = [0] + eps_range
```

Now, we can plot the accuracies for both classifiers across all attack strengths with the following code:

```python
fig, ax = plt.subplots(figsize=(14,7))
ax.plot(
    np.array(eps_range), np.array(accuracy_base),\
    'b-', label='Base classifier'
)
ax.plot(
    np.array(eps_range), np.array(accuracy_robust),\
    'r-', label='Robust classifier'
)
legend = ax.legend(loc='upper center')
plt.xlabel('Attack strength (eps)')
plt.ylabel('Accuracy')
```

The preceding code generates *Figure 13.14*, which demonstrates that the robust model performs better between attack strengths of 0.02 and 0.3 but then does consistently about 10% worse:
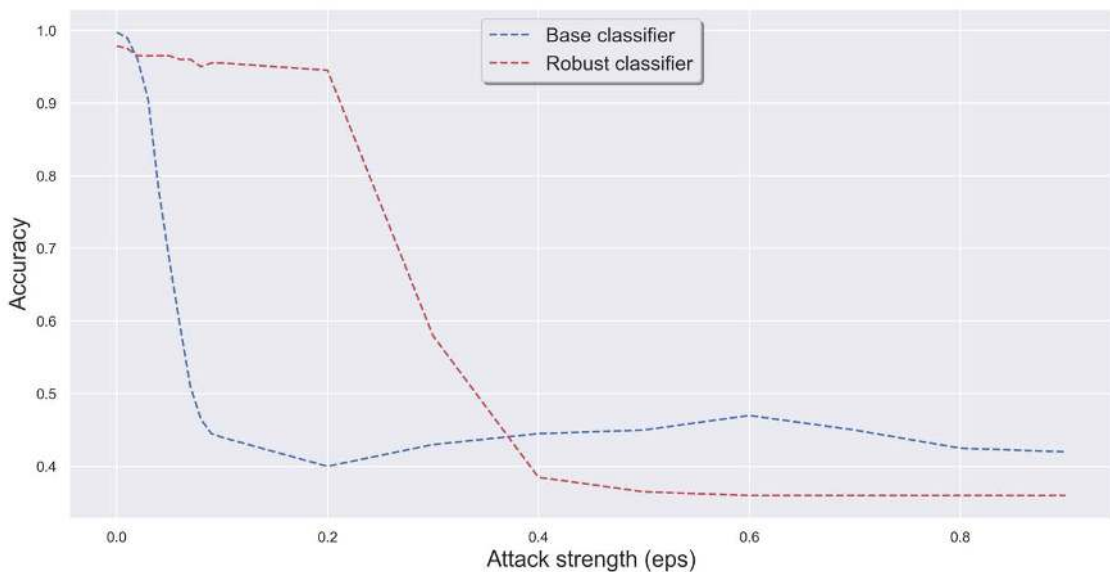
Figure 13.14: Accuracy measured for the robust and base classifiers at different FSGM attack strengths

One thing that *Figure 13.14* fails to account for is defenses. For example, if hospital cameras were constantly jammed or tampered with, the security company would be remiss not to defend their models. The easiest way to do so for this kind of attack is with some sort of smoothing.

Adversarial training also produces an empirically robust classifier that you cannot guarantee will work under certain pre-defined circumstances, which is why there's a need for certifiable defenses.

Mission accomplished

The mission was to perform some adversarial robustness tests on their face mask model to determine if hospital visitors and staff can evade mandatory mask compliance. The base model performed very poorly on many evasion attacks, from the most aggressive to the most subtle.

You also looked at possible defenses to these attacks, such as spatial smoothing and adversarial retraining. And then, you explored ways to evaluate the robustness of your proposed defenses. You can now provide an end-to-end framework to defend against this kind of attack. That being said, what you did was only a proof of concept.

Now, you can propose training a certifiably robust model against the attacks the hospitals expect to encounter the most. But first, you need the ingredients for a generally robust model. To this end, you will need to take all 210,000 images in the original dataset, make many variations on mask colors and types with them, and augment them even further with reasonable brightness, shear, and rotation transformations. Lastly, the robust model needs to be trained with several kinds of attacks, including several kinds of APs. These are important because they mimic the most common compliance evasion behavior of concealing faces with body parts or clothing items.

Summary

After reading this chapter, you should understand how attacks can be perpetrated on machine learning models and evasion attacks in particular. You should know how to perform FSGM, BIM, PGD, C&W, and AP attacks, as well as how to defend against them with spatial smoothing and adversarial training. Last but not least, you know how to evaluate adversarial robustness.

The next chapter is the last one, and it outlines some ideas on what's next for machine learning interpretation.

Dataset sources

- Adnane Cabani, Karim Hammoudi, Halim Benhabiles, and Mahmoud Melkemi, 2020, *Masked-Face-Net - A dataset of correctly/incorrectly masked face images in the context of COVID-19*, Smart Health, ISSN 2352–6483, Elsevier: `https://doi.org/10.1016/j.smhl.2020.100144` (Creative Commons BY-NC-SA 4.0 license by NVIDIA Corporation)
- Karras, T., Laine, S., and Aila, T., 2019, *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 4396–4405: `https://arxiv.org/abs/1812.04948` (Creative Commons BY-NC-SA 4.0 license by NVIDIA Corporation)

Further reading

- Polyakov, A., 2019, Aug 6, *How to attack Machine Learning (Evasion, Poisoning, Inference, Trojans, Backdoors)* [blog post]: `https://towardsdatascience.com/how-to-attack-machine-learning-evasion-poisoning-inference-trojans-backdoors-a7cb5832595c`

- Carlini, N., & Wagner, D., 2017, *Towards Evaluating the Robustness of Neural Networks*. 2017 IEEE Symposium on Security and Privacy (SP), 39–57: `https://arxiv.org/abs/1608.04644`
- Brown, T., Mané, D., Roy, A., Abadi, M., and Gilmer, J., 2017, *Adversarial Patch*. ArXiv: `https://arxiv.org/abs/1712.09665`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

14

What's Next for Machine Learning Interpretability?

Over the last thirteen chapters, we have explored the field of **Machine Learning** (**ML**) interpretability. As stated in the preface, it's a broad area of research, most of which hasn't even left the lab and become widely used yet, and this book has no intention of covering absolutely all of it. Instead, the objective is to present various interpretability tools in sufficient depth to be useful as a starting point for beginners and even complement the knowledge of more advanced readers. This chapter will summarize what we've learned in the context of the ecosystem of ML interpretability methods, and then speculate on what's to come next!

These are the main topics we are going to cover in this chapter:

- Understanding the current landscape of ML interpretability
- Speculating on the future of ML interpretability

Understanding the current landscape of ML interpretability

First, we will provide some context on how the book relates to the main goals of ML interpretability and how practitioners can start applying the methods to achieve those broad goals. Then, we'll discuss the current areas of growth in research.

Tying everything together!

As discussed in *Chapter 1*, *Interpretation, Interpretability, and Explainability; and Why Does It All Matter?*, there are three main themes when talking about ML interpretability: **Fairness, Accountability, and Transparency** (**FAT**), and each of these presents a series of concerns (see *Figure 14.1*). I think we can all agree these are all desirable properties for a model! Indeed, these concerns all present opportunities for the improvement of **Artificial Intelligence** (**AI**) systems. These improvements start by leveraging model interpretation methods to evaluate models, confirm or dispute assumptions, and find problems.

What your aim is will depend on what stage you are at in the ML workflow. If the model is already in production, the objective might be to evaluate it with a whole suite of metrics, but if the model is still in early development, the aim may be to find deeper problems that a metric won't discover. Perhaps you are also just using black-box models for knowledge discovery as we did in *Chapter 4* – in other words, leveraging the models to learn from the data with no plan to take it to production. If this is the case, you might confirm or dispute the assumptions you had about the data, and by extension, the model:
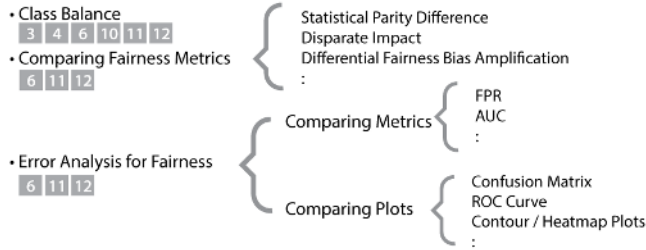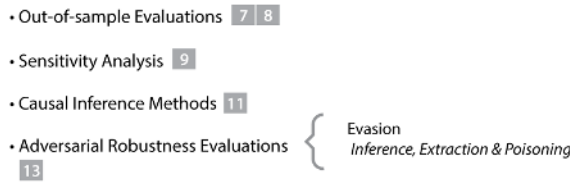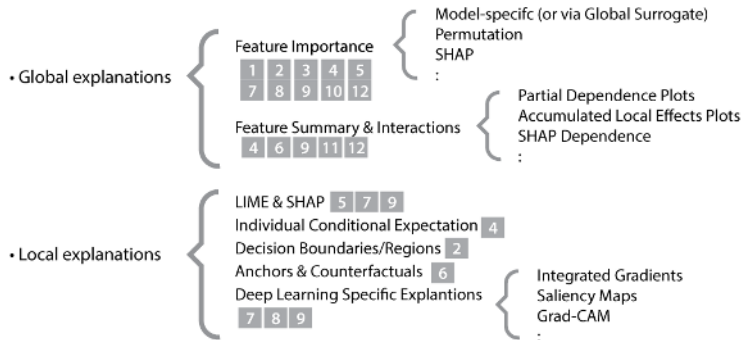
Figure 14.1: ML interpretation methods

In any case, none of these aims are mutually exclusive, and you should probably always be looking for problems and disputing assumptions, even when the model appears to be performing well!

And regardless of the aim and primary concern, it is recommended that you use many interpretation methods, not only because no technique is perfect but also because all problems and aims are interrelated. In other words, there's no justice without consistency and no reliability without transparency.

In fact, you can read *Figure 14.1* from bottom to top as if it were a pyramid, because transparency is foundational, followed by accountability in the second tier, and, ultimately, fairness is the cherry on top.

Therefore, even when the goal is to assess model fairness, the model should be stress-tested for robustness. Most relevant feature importances and interactions should be understood. Otherwise, it won't matter if predictions aren't robust and transparent.

There are many interpretation methods covered in *Figure 14.1*, and these are by no means every interpretation method available. They represent the most popular methods with well-maintained open-source libraries behind them. In this book, we have touched on most of them, albeit some of them only briefly. Those that weren't discussed are in *italics* and those that were discussed have the relevant chapter numbers provided next to them. There's been a focus on **model-agnostic** methods for **black-box supervised learning models**. Still, outside of this realm, there are also many other interpretation methods, such as those found in reinforcement learning, generative models, or the many statistical methods used strictly for linear regression. And even within the supervised learning black-box model realm, there are hundreds of application-specific model interpretation methods used for applications ranging from chemistry graph CNNs to customer churn classifiers.

That being said, many of the methods discussed in this book can be tailored to a wide variety of applications. Integrated gradients can be used to interpret audio classifiers and weather forecasting models. Sensitivity analysis can be employed in financial modeling and infectious disease risk models. Causal inference methods can be leveraged to improve user experience and drug trials.

Improve is the operative word here because interpretation methods have a flip side!

In this book, that flip side has been referred to as *tuning for interpretability*, which means creating solutions to problems with FAT. Those solutions can be appreciated in *Figure 14.2*:

INTERPRETABILITY SOLUTIONS BY CHAPTER
Mitigating bias, placing guardrails, enhancing reliability, reducing complexity, and ensuring privacy

	DATA	MODEL	PREDICTION
FAIRNESS	Resampling / Reweighting `11` Feature Engineering `10` `12` Data Augmentation `8` `11` `13` Feature Selection `10` (Filter, Embedded, Wrapper)	Cost-sensitive Learning `10` `11` `12` Monotonic Constraints `12` Adversarial Debiasing `11` Regularization `3` `12`	Calibrating/Equalizing Odds `6` `11` Prediction Abstention `13` *Fairness Model Certification*
ACCOUNTABILITY	*Feature Drift Detection* Data Augmentation `8` `11` `13` Adv. Preproc. Defenses `13` Feature Selection `10` (Filter, Embedded, Wrapper) Feature Engineering `10` `12` *Data Anonymization* *Differential Privacy*	*Uncertainty Estimation /* *Conformal Prediction* Adversarial Robustness Certified Training & Inference `13` Adversarial Training `13` Regularization `3` `12` (plus other under-fitting tuning) Monotonic Constraints `12` *Federated Learning* *Other Adversarial Defenses* *(for espionage attacks)*	Adv. Postprocessing Def `13` Calibrating/Equalizing Odds `7` `11` *Privacy-Preserving Inference*
TRANSPARENCY	Feature Selection `10` (Filter, Embedded, Wrapper) Feature Engineering `10` `12`	Regularization `3` `12` (plus other under-fitting tuning) Model Constraints `12` White & Glass-Box Models `3` `4`	Local Interpretation `6` `7` `8` `9`

Figure 14.2: Toolset to treat FAT issues

I have observed five approaches to interpretability solutions:

- **Mitigating Bias:** Any corrective measure that is taken to account for bias. Please note that this bias refers to the sampling, exclusion, prejudice, and measurement biases in the data, along with any other bias introduced into the ML workflow.
- **Placing Guardrails:** Any solution that ensures that the model is constrained so that it doesn't contradict the domain knowledge and predict without confidence.
- **Enhancing Reliability:** Any fix that increases the confidence and consistency of predictions, excluding those that do so by reducing complexity.
- **Reducing Complexity:** Any means by which sparsity is introduced. As a side effect, this generally enhances reliability by generalizing better.
- **Ensuring Privacy:** Any effort to secure private data and model architecture from third parties. We didn't cover this approach in this book.

There are also three areas in which these approaches can be applied:

- **Data ("preprocessing"):** By modifying the training data
- **Model ("in-processing"):** By modifying the model, its parameters, or training procedure
- **Prediction ("postprocessing"):** By intervening in the inference of the model

There's a fourth area that can impact the other three – namely, data and algorithmic governance. This includes regulations and standards that dictate a certain methodology or framework. It's a missing column because very few industries and jurisdictions have laws dictating what methods and approaches should be applied to comply with FAT. For instance, governance could impose a standard for explaining algorithmic decisions, data provenance, or a robustness certification threshold. We will discuss this further in the next section.

You can tell in *Figure 14.2* that many of the methods repeat themselves for FAT. **Feature Selection and Engineering, Monotonic Constraints**, and **Regularization** benefit all three but are not always leveraged by the same approach. **Data Augmentation** also can enhance reliability for fairness and accountability. As with *Figure 14.1*, the items in italics were not covered in the book, of which three topics stand out: **Uncertainty Estimation**, **Adversarial Robustness**, and **Privacy Preservation** are fascinating topics and deserve books of their own.

Current trends

One of the most significant deterrents of AI adoption is a lack of interpretability, which is partially the reason why 50-90% of AI projects never take off (see the *Further reading* section for articles about this), and the other is the ethical transgressions that happen as a result of not complying with FAT. In this aspect, **Interpretable Machine Learning** (**iML**) has the power to lead ML as a whole because it can help with both goals with the corresponding methods in *Figure 14.1* and *Figure 14.2*.

Thankfully, we are witnessing an increase in interest and production in iML, mostly under **Explainable Artificial Intelligence** (**XAI**) – see *Figure 14.3*. In the scientific community, iML is still the most popular term, but XAI dominates in public settings:

Academic Publications
(Source: Dimesions.ai)

Book Publications
% of yearly total
(Source: Google Books Ngram Viewer)

Search Trends
yearly avg % of historical max
(Source: Google Trends)
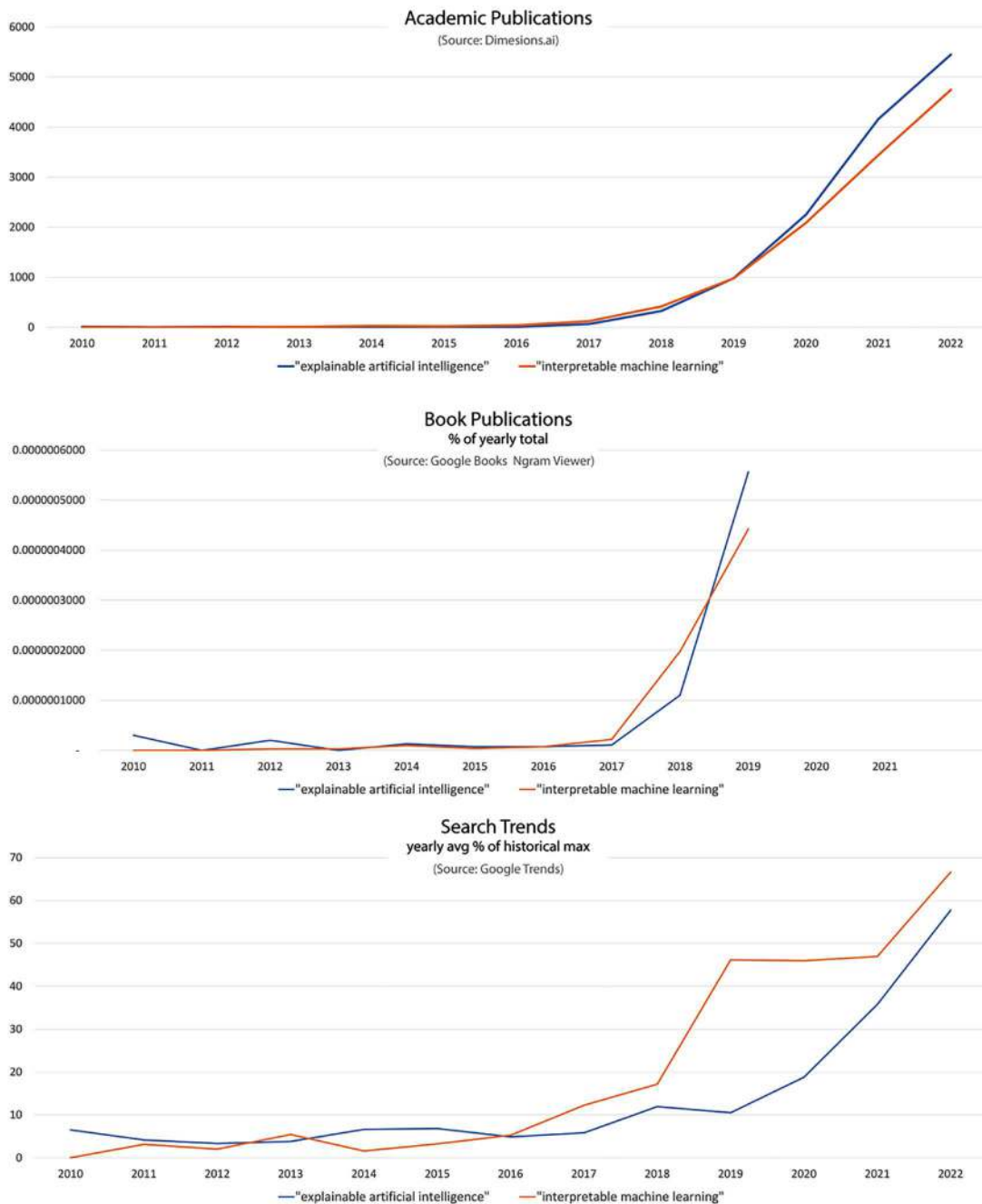
Figure 14.3: Publication and search trends for iML and XAI

This means that just as ML is starting to get standardized, regulated, consolidated, and integrated into a whole host of other disciplines, interpretation will soon get a seat at the table.

ML is replacing software in all industries. And as more is getting automated, more models are deployed to the cloud, and it will get worse with the **Artificial Intelligence of Things** (**AIoT**). Deployment is not traditionally in the ML practitioner's wheelhouse. That is why ML increasingly depends on **Machine Learning Operations** (**MLOps**). And the pace of automation means more tools are needed to build, test, deploy, and monitor these models. At the same time, there's a need for the standardization of tools, methods, and metrics. Slowly but surely, this is happening. Since 2017, we have had the **Open Neural Network Exchange** (**ONNX**), an open standard for interoperability. And at the time of writing, the **International Organization for Standardization** (**ISO**) has over two dozen AI standards being written (and one published), several of which involve interpretability. Naturally, some things will get standardized because of common use, due to the consolidation of ML model classes, methods, libraries, service providers, and practices. Over time, one or a few in each area will emerge. Lastly, given ML's outsized role in algorithmic decision-making, it's only a matter of time before it is regulated. Only some financial markets regulate trading algorithms, such as the **Securities and Exchange Commission** (**SEC**) in the United States and the **Financial Conduct Authority** (**FCA**) in the UK. Besides that, only data privacy and provenance regulations are widely enforced, such as the HIPAA in the US and the LGPD in Brazil. The GDPR in the European Union takes this a bit further with the "right to an explanation" for algorithmic decisions but the intended scope and methodology are still unclear.

> **XAI versus IML – which one to use?**
>
> My take: although they are understood as synonyms in industry and *iML* is regarded as more of an academic term, ML practitioners, even those in industry, should be wary about using the term *XAI*. Words can have outsized suggestive power. *Explainable* presumes full understanding, but *interpretable* leaves room for error, as there always should be when talking about models, and extraordinarily complex black-box ones at that. Furthermore, AI has captured the public imagination as a panacea or has been vilified as dangerous. Either way, along with the term *explainable*, it serves to make it even more filled with hubris for those who think it's a panacea and perhaps calm some concerns for those who think it's dangerous. XAI term might be serving a purpose as a marketing term. However, for those who build models, the suggestive power of the word *explainable* can make us overconfident in our interpretations. That being said, this is just an opinion.

ML interpretability is growing quickly but is lagging behind ML. Some interpretation tools have been integrated into the cloud ecosystem, from SageMaker to DataRobot. They are yet to be fully automated, standardized, consolidated, and regulated, but there's no doubt that this will happen.

Speculating on the future of ML interpretability

I'm used to hearing the metaphor of this period being the "Wild West of AI", or worse, an "AI Gold Rush!" It conjures images of an unexplored and untamed territory being eagerly conquered, or worse, civilized. Yet, in the 19th century, the United States western areas were not too different from other regions on the planet and had already been inhabited by Native Americans for millennia, so the metaphor doesn't quite work. Predicting with the accuracy and confidence that we can achieve with ML would spook our ancestors and is not a "natural" position for us humans. It's more akin to flying than exploring unknown land.

The article *Toward the Jet Age of machine learning* (linked in the *Further reading* section at the end of this chapter) presents a much more fitting metaphor of AI being like the dawn of aviation. It's new and exciting, and people still marvel at what we can do from down below (see *Figure 14.4*)!

However, aviation had yet to fulfill its potential. Decades after the barnstorming era, aviation matured into the safe, reliable, and efficient Jet Age of **commercial aviation**. In the case of aviation, the promise was that it could reliably take goods and people halfway around the world in less than a day. In AI's case, the promise is that it can make fair, accountable, and transparent decisions – maybe not for any decision, but at least those it was designed to make unless it's an example of **Artificial General Intelligence** (**AGI**):

Figure 14.4: Barnstorming during the 1920s (United States Library of Congress's Prints and Photographs Division)

So how do we get there? The following are a few ideas I anticipate will occur in the pursuit of reaching the Jet Age of ML.

A new vision for ML

As we intend to go farther with AI than we have ever gone before, the ML practitioners of tomorrow must be more aware of the dangers of the sky. And by the sky, I mean the new frontiers of predictive and prescriptive analytics. The risks are numerous and involve all kinds of biases and assumptions, problems with data both known and potential, and our models' mathematical properties and limitations. It's easy to be deceived by ML models thinking they are software. Still, in this analogy, the software is completely deterministic in nature – it's solidly anchored to the ground, not hovering in the sky!

For civil aviation to become safe, it required a new mindset – a new culture. The fighter pilots of WWII, as capable as they were, had to be retrained to work in civil aviation. It's not the same mission because when you know that you are carrying passengers on board and the stakes are high, everything changes.

Ethical AI, and by extension, iML, ultimately require this awareness that models directly or indirectly carry passengers "on board," and that models aren't as robust as they seem. A robust model must be able to reliably withstand almost any condition over and over again in the same way the planes of today do. To that end, we need to be using more instruments, and those instruments come in the form of interpretation methods.

A multidisciplinary approach

Tighter integration with many disciplines is needed for models that comply with the principles of FAT. This means more significant involvement of AI ethicists, lawyers, sociologists, psychologists, human-centered designers, and countless other professions. Along with AI technologists and software engineers, they will help code best practices into standards and regulations.

Adequate standardization

New standards will be needed not only for code, metrics, and methodologies but also for language. The language behind data has mostly been derived from statistics, math, computer science, and econometrics, which leads to a lot of confusion.

Enforcing regulation

It will likely be required that all production models fulfill the following specifications:

- Are certifiably robust and fair
- Are capable of explaining their reasoning behind one prediction with a TRACE command and, in some cases, are required to deliver the reasoning with the prediction
- Can abstain from a prediction they aren't confident about
- Yield confidence levels for all predictions (see the **conformal prediction** tutorial and book in the *Further reading* section)
- Have metadata with training data provenance (even if anonymized) and authorship and, when needed, regulatory compliance certificates and metadata tied to a public ledger – possibly a blockchain
- Have security certificates much like websites do to ensure a certain level of trust
- Expire, and stop working upon expiration, until they are retrained with new data
- Be taken offline automatically when they fail model diagnostics and only put online again when they pass
- Have **Continuous Training/Continuous Integration** (**CT/CI**) pipelines that help retrain the model and perform the model diagnostics at regular intervals to avoid any model downtime
- Are diagnosed by a certified AI auditor when they fail catastrophically and cause public damage

New regulations will likely create new professions such as AI auditors and model diagnostics engineers. But they will also prop up MLOps engineers and ML automation tools.

Seamless machine learning automation with built-in interpretation

In the future, we won't program an ML pipeline; it will mostly be a drag-and-drop capability with a dashboard offering all kinds of metrics. It will evolve to be mostly automated. Automation shouldn't come as a surprise because some existing libraries perform automated feature-selection model training. Some interpretability-enhancing procedures may be done automatically, but most of them will require human discretion. However, interpretation ought to be injected throughout the process, much like planes that mostly fly themselves have instruments that alert pilots of issues; the value is in informing the ML practitioner of potential problems and improvements at every step. Did it find a feature to recommend for monotonic constraints? Did it find some imbalances that might need adjusting? Did it find anomalies in the data that might need some correction? Show the practitioner what needs to be seen to make an informed decision and let them make it.

Tighter integration with MLOps engineers

Certifiably robust models trained, validated, and deployed at a click of a button require more than just cloud infrastructure – they also need the orchestration of tools, configurations, and people trained in MLOps to monitor them and perform maintenance at regular intervals.

Summary

Interpretable machine learning is an extensive topic, and this book has only covered some aspects of some of its most important areas on two levels: diagnosis and treatment. Practitioners can leverage the tools offered by the toolkit anywhere in the ML pipeline. However, it's up to the practitioner to choose when and how to apply them.

What matters most is to engage with the tools. Not using the interpretable machine learning toolkit is like flying a plane with very few instruments or none at all. Much like flying a plane operates under different weather conditions, machine learning models operate under different data conditions, and to be a skilled pilot or machine learning engineer, we can't be overconfident and validate or rule out hypotheses with our instruments. And much like aviation took a few decades to become the safest mode of transportation, it will take AI a few decades to become the safest mode of decision-making. It will take a global village to get us there, but it will be an exciting journey! And remember, *the best way to predict the future is to create it*.

Further reading

- Joury, A., 2022, January 6, *Why 90% of machine learning models never hit the market*. TNW | Neural. Retrieved December 4, 2022: `https://thenextweb.com/news/why-most-machine-learning-models-never-hit-market-syndication`
- Wiggers, K., 2019, July 8, *IDC: For 1 in 4 companies, half of all AI projects fail*. VentureBeat. Retrieved December 4, 2022: `https://venturebeat.com/ai/idc-for-1-in-4-companies-half-of-all-ai-projects-fail/`

- O'Neil, C., 2017, *Weapons of Math Destruction*. Penguin Books.
- Talwalkar, A., 2018, April 25, *Toward the Jet Age of machine learning*. O'Reilly: `https://www.oreilly.com/content/toward-the-jet-age-of-machine-learning/`
- Rajiv, S., 2022, September 22, *Getting predictions intervals with conformal inference*. Getting predictions intervals with conformal inference · Rajiv Shah's Projects Blog. Retrieved December 4, 2022: `http://projects.rajivshah.com/blog/2022/09/24/conformal_predictions/`
- Angelopoulos, A.N., and Bates, S., 2021, *A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification*: `https://arxiv.org/abs/2107.07511`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask the author questions, and learn about new releases – follow the QR code below:

`https://packt.link/inml`

‹packt›

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:
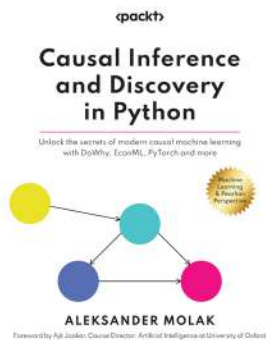
Causal Inference and Discovery in Python

Aleksander Molak

ISBN: 9781804612989

- Master the fundamental concepts of causal inference
- Decipher the mysteries of structural causal models
- Unleash the power of the 4-step causal inference process in Python
- Explore advanced uplift modeling techniques
- Unlock the secrets of modern causal discovery using Python
- Use causal inference for social impact and community benefit
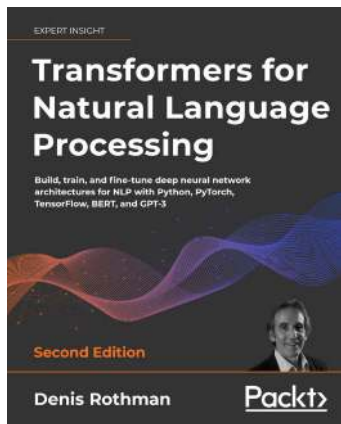
Transformers for Natural Language Processing – Second Edition

Denis Rothman

ISBN: 9781803247335

- Discover new techniques to investigate complex language problems
- Compare and contrast the results of GPT-3 against T5, GPT-2, and BERT-based transformers
- Carry out sentiment analysis, text summarization, casual speech analysis, machine translations, and more using TensorFlow, PyTorch, and GPT-3
- Find out how ViT and CLIP label images (including blurry ones!) and create images from a sentence using DALL-E
- Learn the mechanics of advanced prompt engineering for ChatGPT and GPT-4

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.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 *Interpretable Machine Learning with Python 2e*, 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

W

X

Z

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803235424

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly