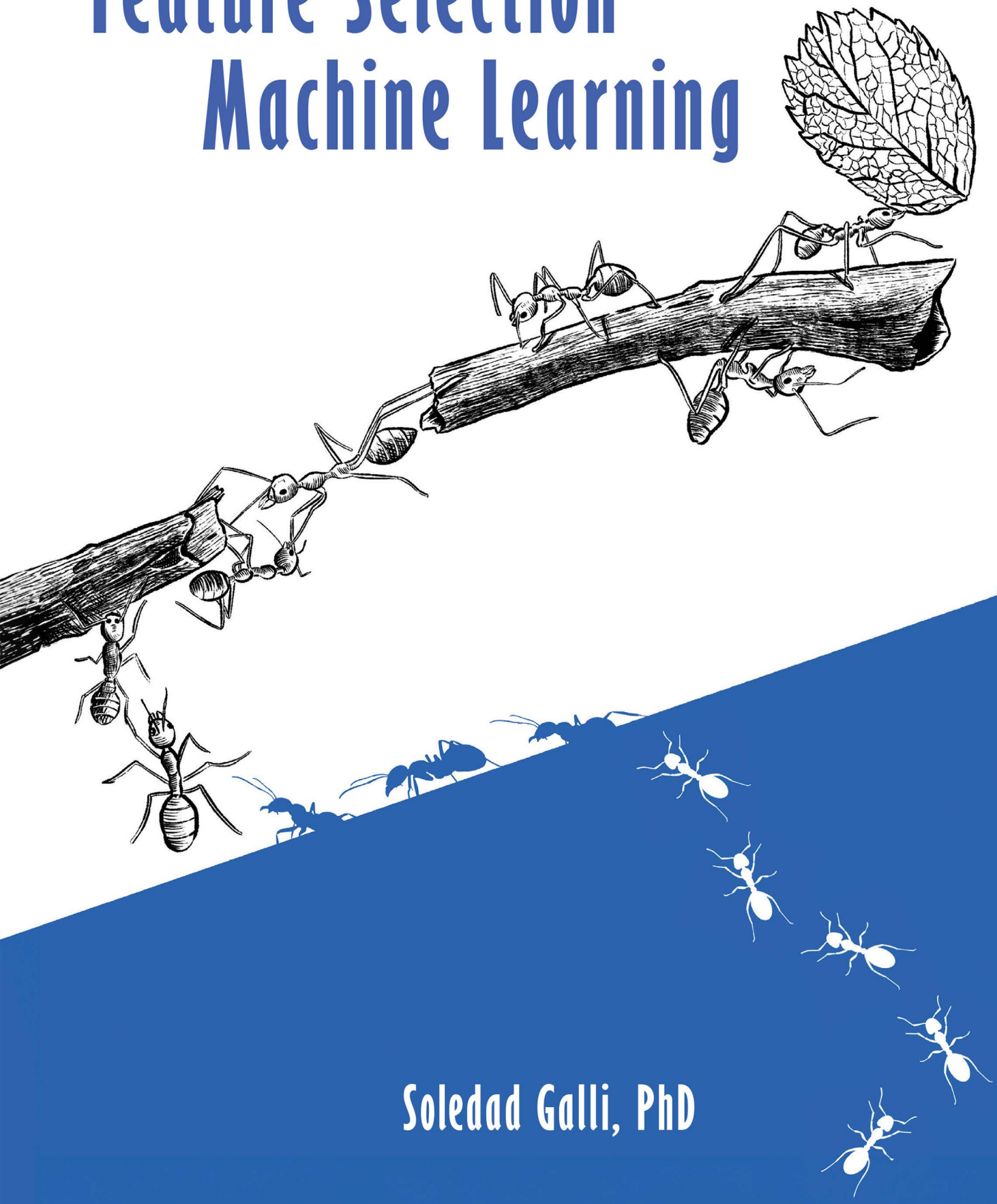
Feature Selection Machine Learning



Soledad Galli, PhD

Feature Selection in Machine Learning

More than 20 methods to select the most predictive features and build simpler, faster and more reliable machine learning models.

Soledad Galli, PhD

Table of contents

Contributors **5**
About the author . . . 5

Preface **6**
Who is this book for . . . 6
What this book covers . . . 6
Technical requirements . . . 7
Download the code files . . . 8
Get in touch . . . 8

1 Feature Selection Overview **10**
1.1 What is feature selection? . . . 10
1.2 Why do we select features? . . . 11
1.3 Feature selection methods . . . 13
1.4 Filter methods . . . 15
1.5 Wrapper methods . . . 18
1.6 Embedded methods . . . 19
1.7 Other methods . . . 21
1.8 References . . . 22

2 Basic Feature Selection Methods **23**
2.1 Removing constant features . . . 23
2.2 Removing quasi-constant features . . . 28
2.3 Removing duplicated features . . . 34
2.4 References . . . 38

3 Correlation of Predictors **39**
3.1 Correlation coefficients . . . 40
3.2 Visualizing correlated features . . . 41
3.3 Removing correlated features: retain first . . . 46
3.4 Removing correlated features: retain best . . . 52
3.5 Correlation of categorical variables . . . 58
3.6 References . . . 59

4 Statistical Methods **60**
4.1 Chi-square . . . 61

4.2 ANOVA . . . 70
4.3 Correlation . . . 84
4.4 Mutual information . . . 91
4.5 References . . . 107

5 Univariate Feature Selection **109**
5.1 Single feature models . . . 109
5.2 Target encoding . . . 124
5.3 References . . . 136

6 Wrapper Methods **137**
6.1 Exhaustive search . . . 137
6.2 Forward feature selection . . . 141
6.3 Backward feature selection . . . 147
6.4 References . . . 152

7 Embedded Methods **153**
7.1 Lasso regularization . . . 154
7.2 Decision trees and tree ensembles . . . 160
7.3 References . . . 166

8 Recursive Feature Elimination **167**
8.1 Recursive feature elimination - embedded importance . . . 167
8.2 Recursive feature elimination - performance degradation . . . 170
8.3 References . . . 178

9 Alternative Feature Selection Methods **179**
9.1 Recursive feature addition . . . 179
9.2 Feature permutation . . . 189
9.3 Probe features selection . . . 196
9.4 Minimum Redundancy Maximum Relevance (MRMR) . . . 202
9.5 Boruta . . . 206
9.6 References . . . 214

More from the author **215**
Congratulations! . . . 215
Keep going . . . 215

Feature Selection in Machine Learning

Copyright © Soledad Galli, PhD

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. The author will not be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

This book is for sale at:

https://www.trainindata.com/p/feature-selection-in-machine-learning-book

Cover Design: Damian Modena, https://www.instagram.com/demilustraciones/

First published: January 2020

Second edition: February 2025



www.trainindata.com

Contributors

About the author

Soledad Galli is a bestselling data science instructor, book author, and open source Python developer. As the leading instructor at Train in Data, Sole teaches intermediate and advanced courses in machine learning that have enrolled 70k+ students worldwide and continue to receive positive reviews. Sole is also the developer and maintainer of the Python open source library Feature-engine (https://feature-engine.trainindata.com), which offers an extensive array of methods for feature engineering and selection.

Sole worked as a data scientist in finance and insurance companies, where she developed and put into production machine learning models to assess insurance claims and credit risk, and prevent fraud. Sole has been selected multiple times as one of LinkedIn's voices in data science. She is passionate about sharing her knowledge and experience, and that is why you'll often hear her talking at meetups, podcasts, or authoring articles online.

Sole is constantly looking for people like you, who can support her in enhancing the functionality of Feature-engine or delivering more and better courses. If you are interested, contact her over social media or through Train in Data's website.

Preface

Feature Selection in Machine Learning covers widely used methods to select predictive subsets of features.

The aim of variable selection is to improve the performance of machine learning models, to create faster and more cost-effective algorithms, and to facilitate the understanding of the predictions returned by the models.

There are many feature selection algorithms, each with advantages and limitations. Some algorithms scale well and are therefore suitable for extremely high feature spaces. However, they may not consider feature interactions or the mechanisms of the predictive model. Other methods are computationally expensive, yet they work well with particular classifiers or regression models.

Feature Selection in Machine Learning discusses many feature selection procedures, highlighting their advantages and shortcomings. In addition, it provides code recipes to select features utilizing Python open-source libraries.

Throughout the book, you will learn how to select the best subsets of features based on the size of your dataset, the nature of your variables, and the characteristics of the machine learning models you use. By the end of the book, you will be able to decide and implement the best selection algorithm for your data science project.

Who is this book for

This book is intended for data scientists and machine learning engineers who want to know more about the different methods and tools that can be used for feature selection. Knowledge of Python programming (pandas, numpy and scikit-learn) and machine learning is required to make the most out of this book.

What this book covers

Chapter 1: Feature Selection Overview introduces the aim of feature selection, why selecting variables is important, and the main groups of feature selection algorithms. This chapter provides an overview of the main characteristics of the different feature selection algorithms.

Chapter 2: Basic Feature Selection Methods discusses quick data screening techniques for eliminating redundant or irrelevant features, such as constant and duplicate variables. By looking at feature variability, we can often significantly reduce the feature space with little computational cost.

Chapter 3: Correlation of Predictors introduces the use of correlation to discard redundant features. We discuss a brute force approach and an alternative method that takes advantage of the feature's characteristics to select the most useful variable from a correlated group.

Chapter 4: Statistical Methods describes the use of statistical tests to select features in high-dimensional data, such as ANOVA, chi-square, correlation, and mutual information.

Chapter 5: Univariate Feature Selection covers algorithms that evaluate features individually based on single feature models or target encoding.

Chapter 6: Wrapper Methods discusses methods that wrap the search around the classifier or regression model, such as forward and backward sequential search and exhaustive search.

Chapter 7: Embedded Methods describes feature selection methods that are integrated into the learning process of the machine learning model, such as regularization with Lasso and the induction of decision trees.

Chapter 8: Recursive Feature Elimination describes procedures in which features are examined by removing them recursively. Feature relevance is determined by embedded methods or through the model's performance. These methods help account for the inaccuracies in feature importance determined by embedded methods when there is multicolinearity.

Chapter 9: Alternative Feature Selection Methods covers additional feature selection algorithms, like permutation feature importance, Minimum Redundancy and Maximum Relevance (MRMR), introducing probe features, and recursive feature addition.

Technical requirements

We will use the Python libraries Matplotlib, NumPy, Pandas, Scikit-learn, MLXtend and Feature-engine.

If you need to install Python, the free Anaconda Python distribution (https://www.anaconda.com/download) comes with most numerical computing libraries out-of-the-box.

Feature-engine can be installed with pip:

```
pip install feature-engine
```

If you use Anaconda, you can install Feature-engine with conda:

```
conda install -c conda-forge feature_engine
```

To learn more about Feature-engine, visit the following sites:

- Docs: https://feature-engine.trainindata.com
- GitHub: https://github.com/feature-engine/feature_engine

MLXtend can be installed with pip:

```
pip install mlxtend
```

To learn more about MLXtend, visit the following sites:

- Docs: http://rasbt.github.io/mlxtend/
- GitHub: https://github.com/rasbt/mlxtend

Download the code files

The code bundle for the book is available on Github at:

- https://github.com/solegalli/feature-selection-in-machine-learning-book

Get in touch

I welcome feedback from readers.

If you have questions or suggestions for improvement, please let me know. Also, let me know if you find the book useful. It boosts my mood and keeps me motivated :)

I've taken every care to ensure the accuracy of the content. Yet, mistakes do happen. If you find mistakes or typos in the book, please let me know.

> ⚠ Warning
>
> If you come across any illegal copies of my book, please let me know by providing the web address where you found the copy.

For all of the above, you can reach me at solegalli@trainindata.com.

Last but not least, if you'd like to make me happy, please leave a review of the book on Trustpilot at: https://www.trustpilot.com/review/trainindata.com

Enjoy!

1 Feature Selection Overview

I'd like to start by introducing the concept of feature selection and discussing why it is important to select variables when developing machine learning models that we'll use in real-world situations, often to make decisions that'll affect people, like you and me. After that, I'll briefly describe the main groups of feature selection techniques, their characteristics, advantages, and limitations, so that you understand why I divided the book in the present chapters. So, let's get started.

1.1 What is feature selection?

Feature selection is the process of selecting a subset of features from the total variables in a dataset to train machine learning models.

> **!** Important
>
> Feature selection techniques do not alter the original representation of the variables, but merely select a subset of them. Hence, models like Principal Component Analysis (PCA) are not per se feature selection methods.

The aims of variable selection are to improve the performance of machine learning models, to create faster and more cost-effective algorithms, and to facilitate the understanding of the predictions returned by the models.

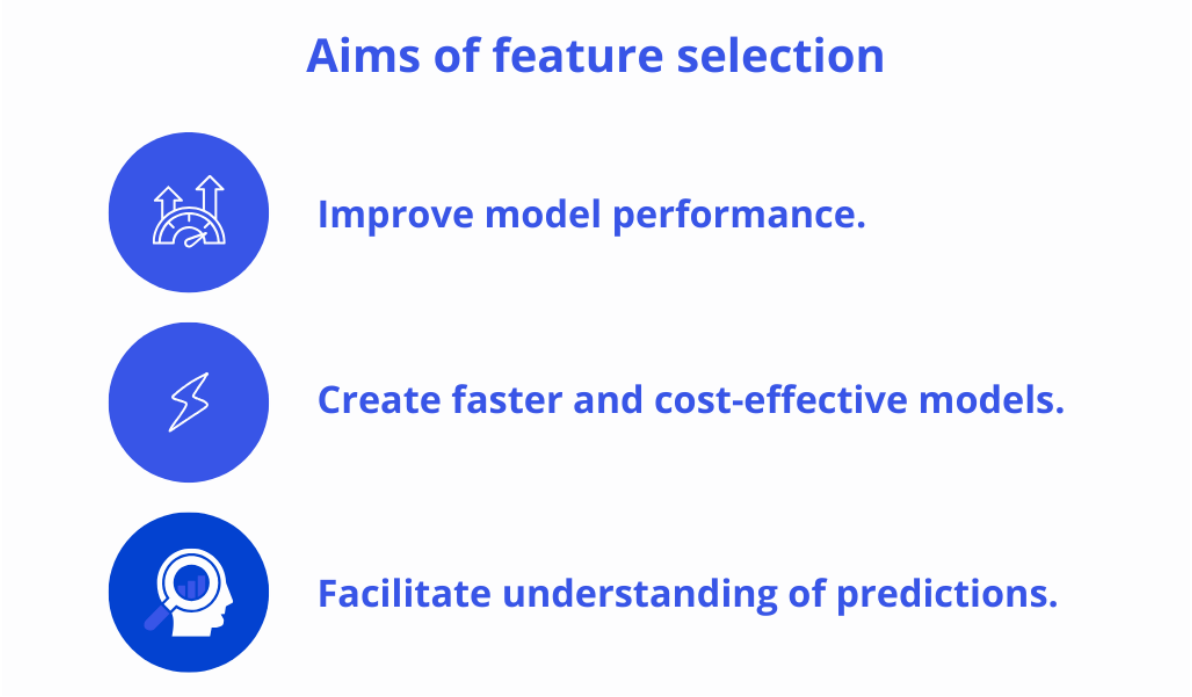


Figure 1.1: Aims of feature selection.

The process of finding the best feature subset typically consists of four basic steps:

1. Generating feature subsets.
2. Assessing the feature subsets.
3. A criterion for stopping the search.
4. Validating the results.

The different feature selection algorithms differ in how they create and evaluate the feature subsets, and how they stop the search. There are many feature selection algorithms, each with advantages and limitations. But before diving into the nature of the search algorithms, I'd like to discuss why we select features in the first place.

1.2 Why do we select features?

When developing a machine learning model for a business or other organization, we rarely incorporate all the variables from the dataset into the model. Sure, adding more variables rarely makes a machine learning model less accurate, but using fewer features has some advantages:

- It facilitates data visualization and data understanding.
- It reduces model storage requirements.
- It reduces model training and utilization times.
- It makes the model more interpretable.

At first glance, it may seem that the more information we feed to the machine learning model, the more accurate it will be. However, simpler models, that is, models trained using fewer features, are more effective in a variety of ways.

First, machine learning models with fewer features are easier to interpret. If a model has 10 predictive variables instead of a hundred, the users of the model can understand the effect of the variables on the outcome much more clearly.

Second, many of the variables in our datasets can be redundant. Redundant variables do not add information to the machine learning model. In fact, in some cases, large feature spaces may reduce the performance of some machine learning models, such as decision tree-based algorithms. Thus, simpler models can be, at least, as accurate as those trained using the entire dataset.

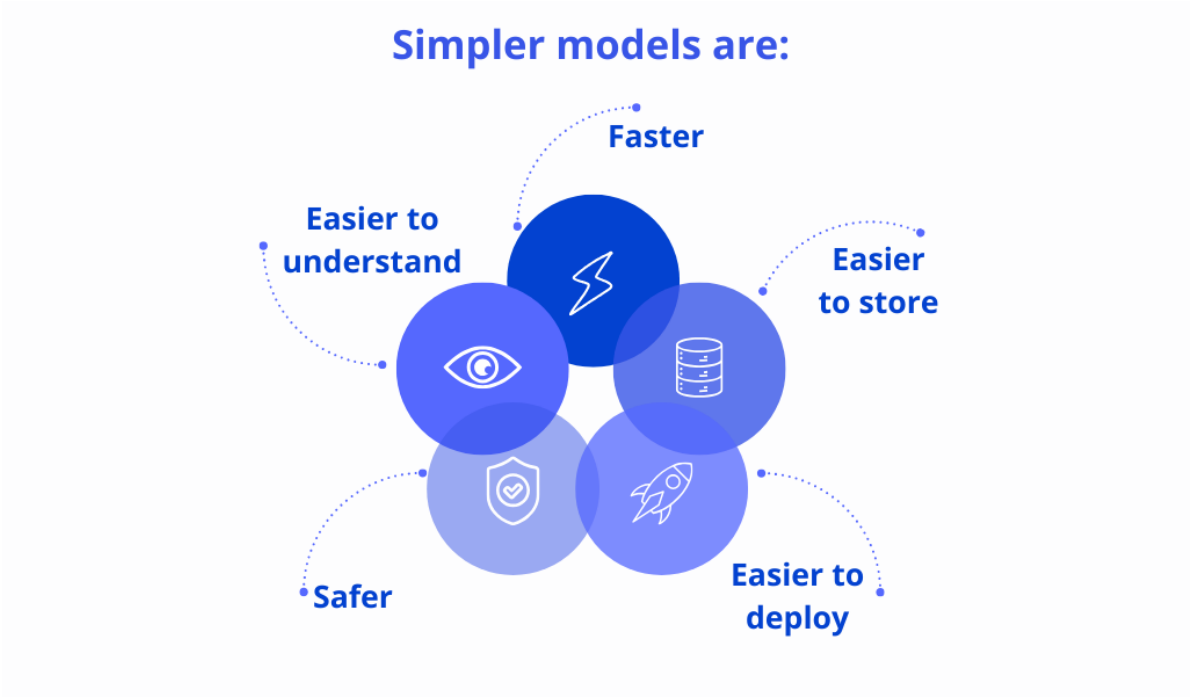


Figure 1.2: Benefits of training models with fewer features.

Third, training a machine learning model using fewer variables reduces the computational cost. This means that simpler models require shorter training times and also make predictions from

incoming data much faster. If the model is to make real-time decisions, speed is particularly important.

Fourth, simpler models require simpler production code. There are a number of checks and tests performed on incoming variables after deployment, like checks for distributions, missing data, and expected values, among others. Models that consume fewer features need fewer tests and less code. This makes model deployment easier and faster to implement, and also easier to maintain.

Finally, models that use fewer features also have a reduced risk of data-dependent errors; the more variables the model uses, the bigger the exposure to errors in the data collection process or changes made to the variable definition.

As you can see, there are a number of advantages to selecting features in machine learning. So how can we select the best features in our data?

1.3 Feature selection methods

Throughout the years, many feature selection methods have been developed, based on specific data and model needs. Depending on how they generate and evaluate the feature subsets, feature selection algorithms can be divided into three groups: filter methods, wrapper methods, and embedded methods.

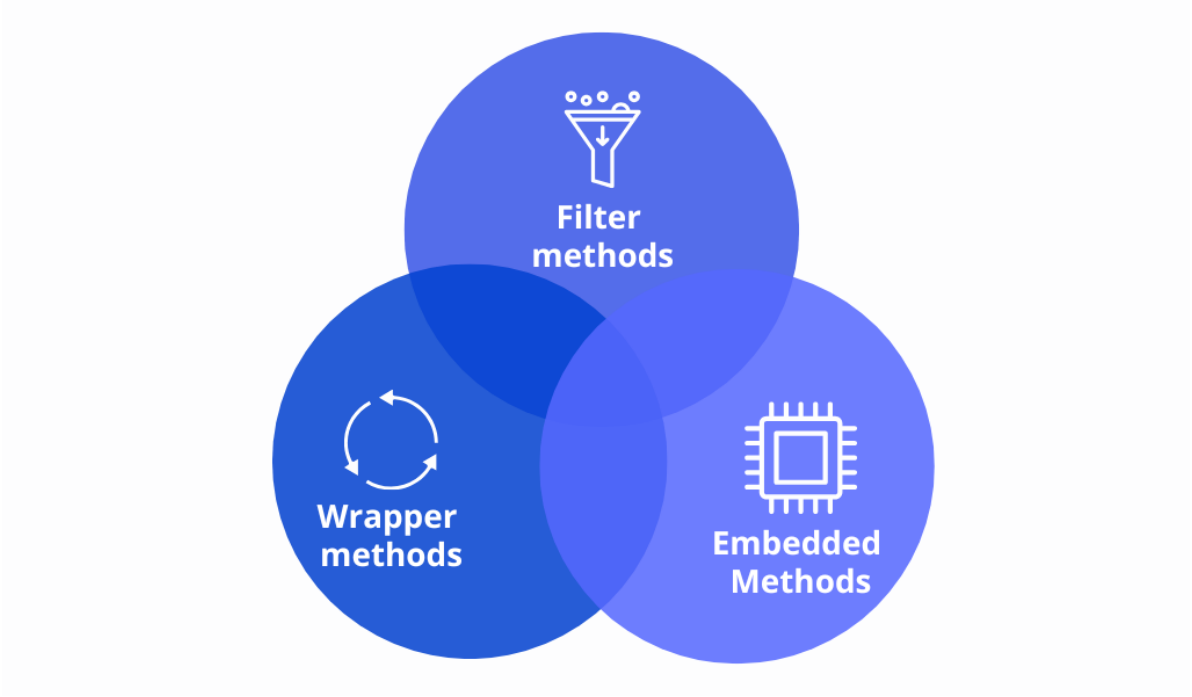


Figure 1.3: Main feature selection groups.

Filter methods select features based on the intrinsic characteristics of the variables, ignoring their interaction with the machine learning model. They are *model agnostic*.

Wrapper methods are algorithms that *wrap* the search around a predictive model. They generate multiple feature subsets and then evaluate their performance based on the classification or regression model.

Embedded methods *embed* the selection procedure in the training or induction of the predictive model.

In addition to the three main groups of selection algorithms, there are additional feature selection procedures that share aspects of filter, wrapper, or embedded methods without falling completely into any category.

In the coming paragraphs, I will discuss the characteristics of each group, as well as common scenarios where these methods are used.

1.4 Filter methods

Filter methods evaluate the significance of the features by focusing solely on the intrinsic properties of the data. In general, filter methods first determine a feature importance score and then remove features with low scores. The classifier or regression model is trained utilizing the top ranking features.

The typical workflow of filter methods involves:

- Ranking features based on some criteria.
- Selecting high-ranking features.

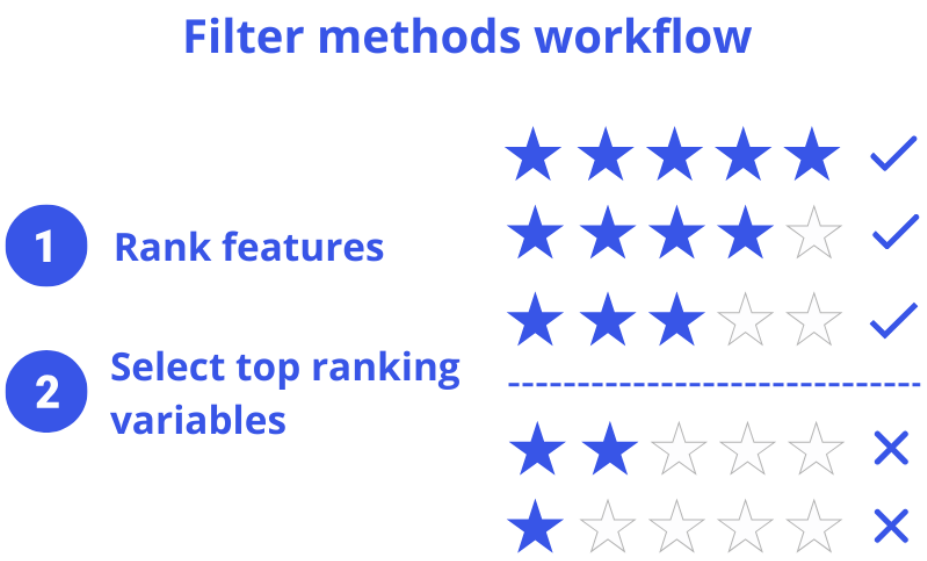


Figure 1.4: Typical filter method workflow.

There are various methods utilized to rank features. Features can be ranked based on how well they separate the classes or how much they correlate with the target. This can be achieved by using statistical tests like ANOVA, chi-square, or correlation, among others.

Features can be ranked based on the performance of a predictive model trained using a single feature, this is what we call single feature classifiers or regression models. Features can also be ranked based on their intrinsic variability. These methods constitute *univariate* filter methods because they assess each feature individually.

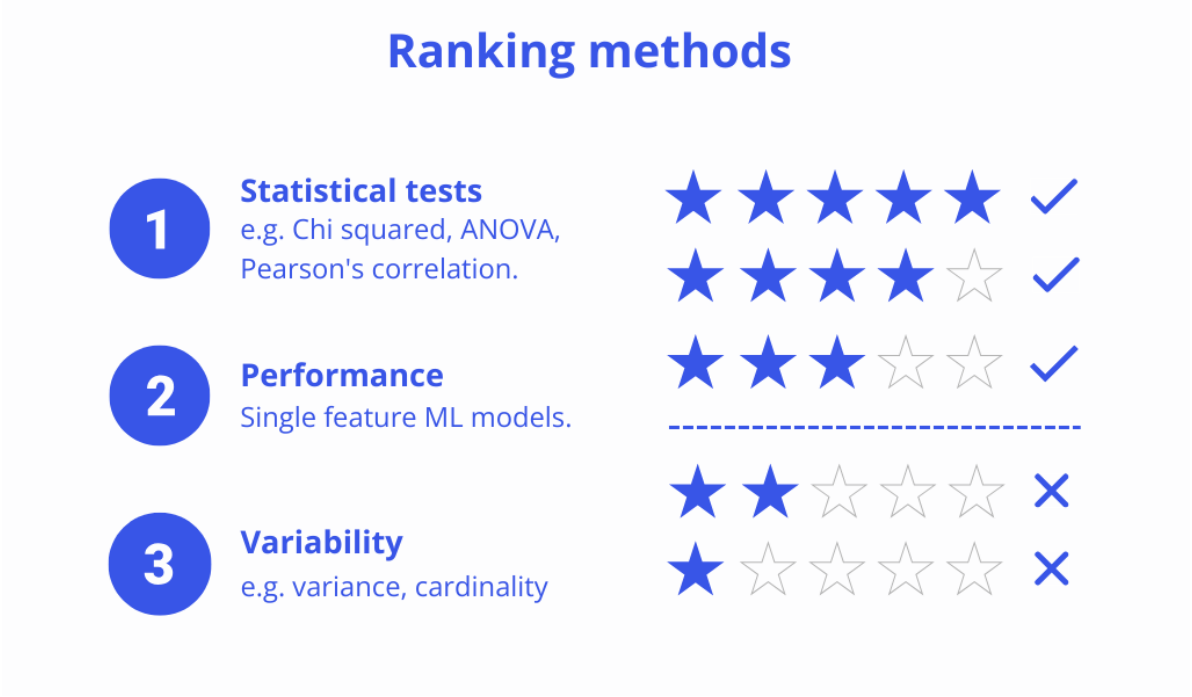


Figure 1.5: Feature ranking methods.

There are additional methods that assess the features collectively and are therefore called *multivariate* filter methods. Finding and removing duplicated or correlated features falls within this group.

Filter methods rank the features and then select the top-ranking ones. But how do they determine the cut-off that delimits the features with the highest ranking? In most cases, the user sets the cut-off. One possibility is to select an arbitrary number of top ranking features. If the test returns a probability value as a ranking metric, then we can set a probability threshold to control the type I error.

Advantages and limitations

Filter methods have several advantages:

- They are fast to compute and hence scalable to high-dimensional datasets.
- They are independent of the machine learning algorithm.

As a result, with filter methods, the feature selection is carried out once, and then different machine learning models can be trained using the selected feature subset.

Filter methods have some limitations as well:

- They ignore the interaction of the features with the machine learning model.
- Most filter methods are univariate, thus, they ignore interactions among features.

Most filter methods evaluate each feature separately, thereby ignoring feature dependencies or interactions with other features. If there is a large number of correlated variables, the ranking criteria will likely assign them a similar score.

In most cases, feature subsets selected by filter methods return models with lower prediction performance compared to those trained on subsets selected by wrapper methods. Yet, because they scale well, filter methods are well suited for quick screening and removal of the most irrelevant features, particularly in large datasets.

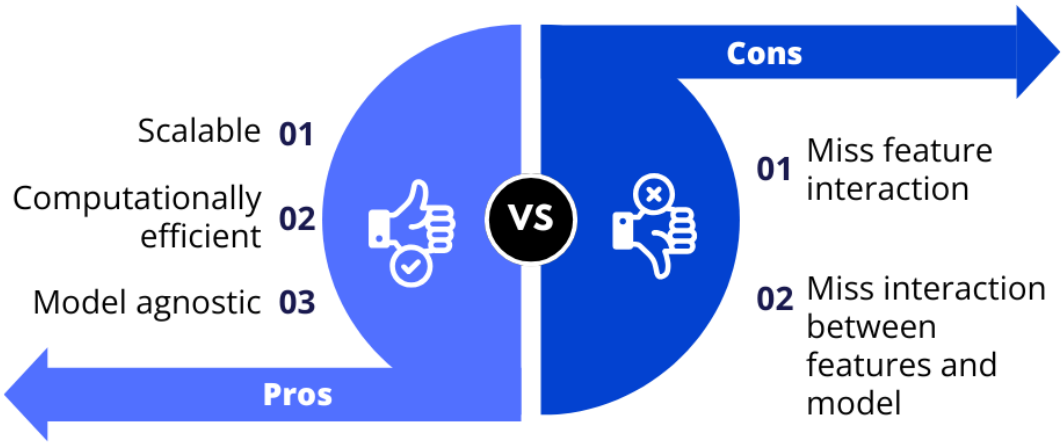


Figure 1.6: Pros and cons of filter methods.

Uses

Filter methods based on statistical tests are commonly used in bioinformatics to discriminate among genes or drugs, or in experiments that involve thousands of features, because of their ability to scale well to high-dimensional data.

Removing constant, low variance, and duplicated features is a common first step in most data science projects with high-dimensional datasets.

1.5 Wrapper methods

Wrapper methods "wrap" the feature selection procedure around a predictive model. The regression or classification algorithm is used as a "black box" to evaluate the performance of the feature subsets.

In the ideal scenario, the algorithm will evaluate all possible feature subsets and select the subset that returns the best performing model. In reality, the size of the search space for n features is 2^n, so it is impractical, if not impossible, to search the whole space exhaustively, unless n is small.

To reduce time and cost, researchers developed less greedy methods for generating feature subsets that guide the search toward the most promising feature groups. The two most prominent examples are forward and backward selection.

In forward selection, the algorithm starts evaluating all individual features and selecting the best one. Then it selects the second best feature by adding the remaining variables, one at a time, and so on. In backward search, the algorithm starts with the complete dataset and removes all features, one at a time, to select which one it should drop. Then it repeats this process with the remaining features.

Advantages and limitations

Wrapper methods have a few advantages:

- They consider the interaction between features.
- They consider the interaction between features and the predictive model.

At least in theory, wrapper methods should find the best feature subset given a machine learning model. In reality, the computational cost is so high that we use stopping criteria to shorten the search time, which can compromise the quality of the feature subset identified by the search.

Wrapper methods have these main limitations:

- The computational cost.
- The stopping criteria can be arbitrary.
- The selected feature subset is optimal for a specific machine learning model.

The main disadvantage of wrapper methods is their computational cost. In addition, the stopping criteria can be somewhat arbitrary. In general, the search stops when the performance of the machine learning model does not improve beyond a certain threshold, and this threshold is usually set arbitrarily by the user. Simpler implementations stop after a certain number of features are selected.

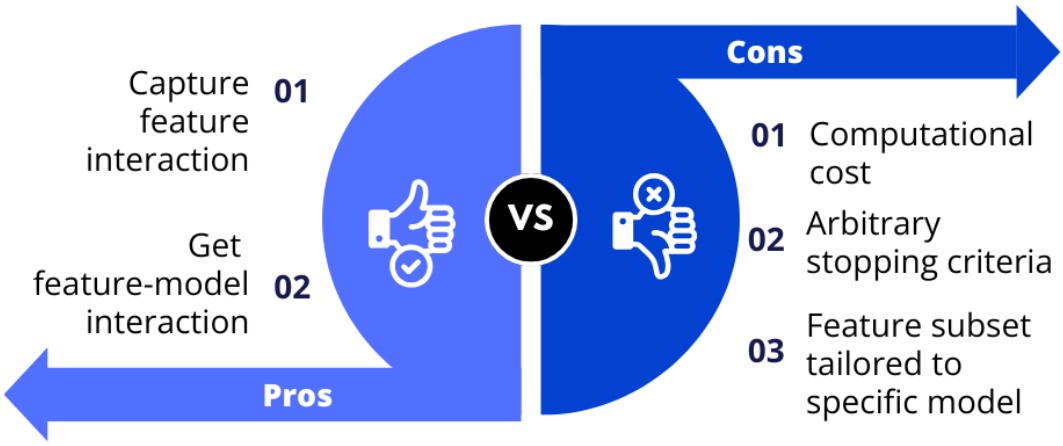


Figure 1.7: Pros and cons of wrapper methods.

Finally, the feature subset that is optimal for one machine learning model may not be well suited for a different machine learning model. So if we want to compare the performance of different models, we need to conduct the feature selection procedure for each model independently.

1.6 Embedded methods

In embedded methods, the search for an optimal subset of features is built into the construction of the classifier or the regression model. Just like wrapper techniques, embedded approaches are specific to a given learning algorithm. But differently from wrapper methods, the embedded strategy trains only one machine learning model. Thus, it is much faster.

A typical embedded feature selection workflow involves:

- Training a machine learning model.
- Deriving feature importance.
- Selecting the top ranking features.

Some machine learning models, like linear regression and decision tree-based models, can determine feature importance as part of their optimization procedure.

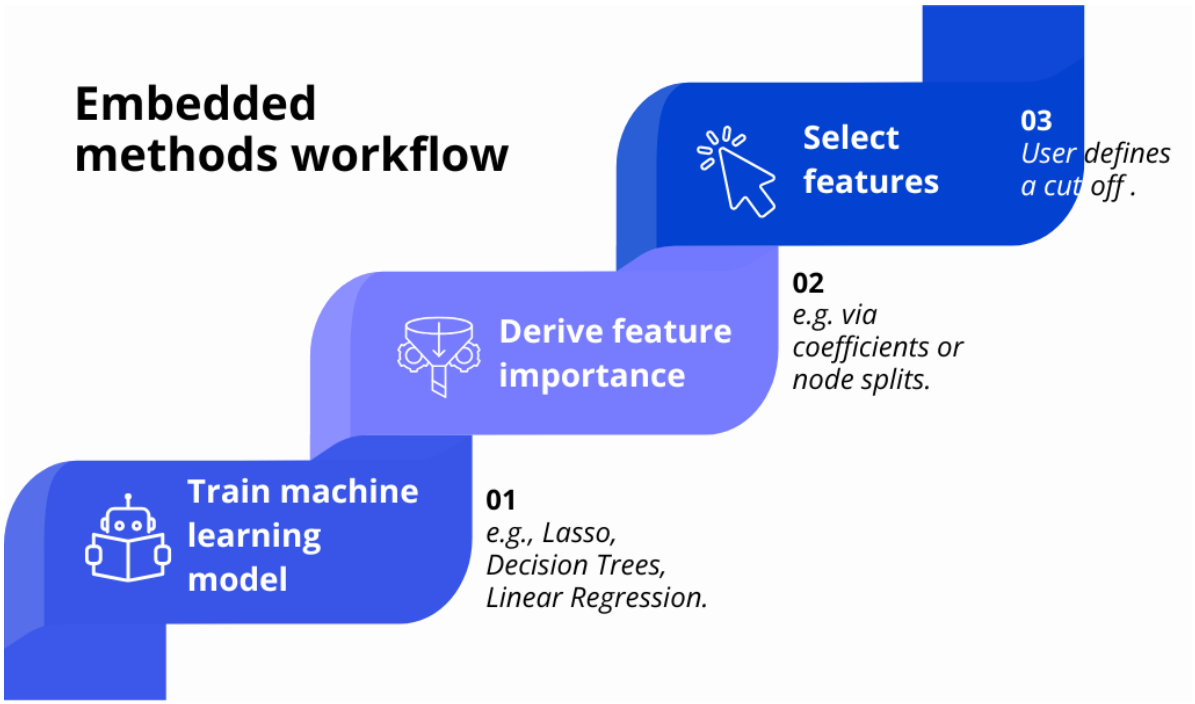


Figure 1.8: Embedded methods workflow.

The question then is how to choose the best features. For Lasso regularization, that is clear: we remove the features whose coefficients have been reduced to zero. For feature importance derived from decision trees, it is not so straightforward. After deriving the feature importance, the algorithm selects the top ranking features. Thus, as with filter methods, the user needs to determine an arbitrary cut-off for the selection.

Advantages and limitations

Embedded methods have a few advantages:

- They consider the interaction between features.
- They consider the interaction between features and the predictive model.
- They are less computationally intensive than wrapper methods.

Therefore, embedded methods tend to provide an effective solution for feature selection: they identify optimal subsets of features at a fraction of the computational cost.

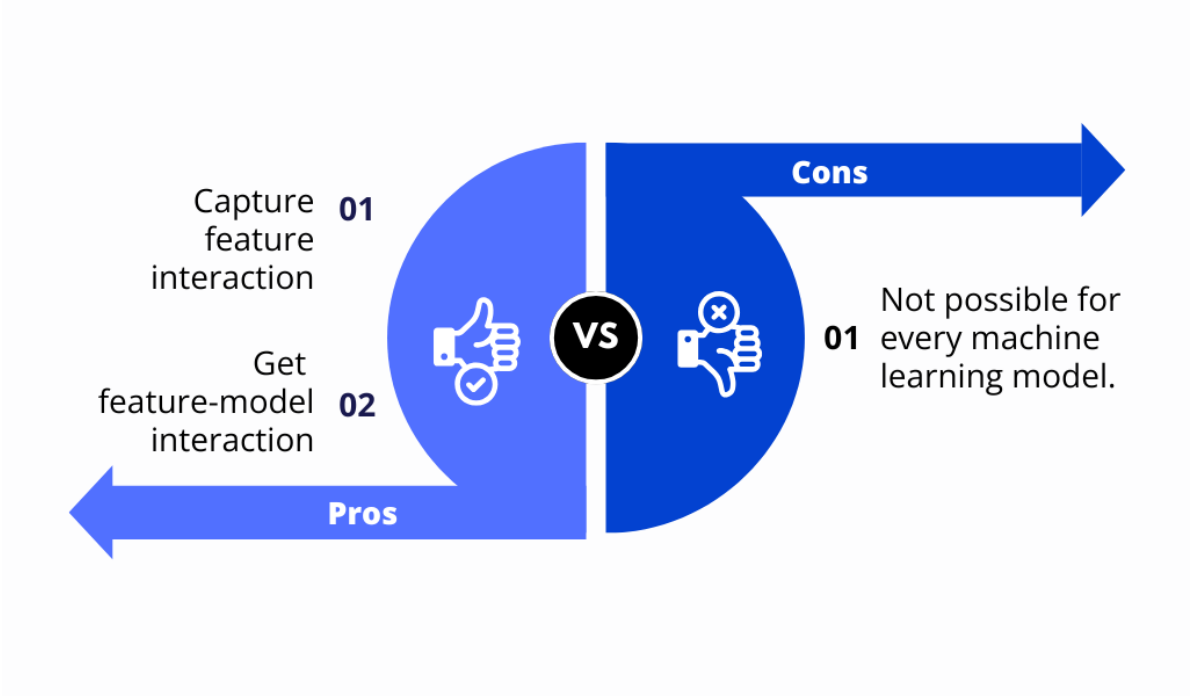


Figure 1.9: Pros and cons of embedded methods.

There are, of course, some limitations, the main one being that it is not possible to derive feature importance from every machine learning model. Thus, embedded methods are only possible for a small number of algorithms. Multicolinearity also affects the importance determined by these models, hence, important features may get lower importance scores if there are correlated variables in the dataset.

1.7 Other methods

There are additional feature selection procedures that are harder to classify into one of the filter, wrapper, or embedded categories. Some of these methods use the feature importance from embedded methods to create the feature subsets, but then evaluate these subsets by training a machine learning model and assessing its performance. Thus, they train more than one model in the selection process. Differently from wrapper methods, these feature selection procedures explore a less exhaustive feature subset space. Therefore, they tend to be faster. Examples of these procedures are recursive feature elimination and recursive feature addition.

Other feature selection procedures derive feature importance by calculating the drop in model performance after a random permutation of the feature values. Alternative methods involve

adding random variables to the dataset (probes) to identify features whose importance is higher than the probes.

These methods work quite differently, so it is hard to derive common advantages or limitations. I will refer the discussion of these feature selection methods to *Chapter 9*.

1.8 References

- Guyon et. al., An Introduction to Variable and Feature Selection, JMLR, 3:1157-1182, 2003.
- Jović, et. al., A review of feature selection methods with applications, 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1200-1205, 2015, doi: 10.1109/MIPRO.2015.7160458.
- Kohavi et. al., Wrappers for feature subset selection, Artificial Intelligence, 97: 273-324, 1997, https://doi.org/10.1016/S0004-3702(97)00043-X.
- Saeys et. al., A review of feature selection techniques in bioinformatics, Bioinformatics, 23: 2507-2517, 2007.
- Tang et. al., Feature Selection for Classification: A Review. Data Classification: Algorithms and Applications, Chapter, 2014.

2 Basic Feature Selection Methods

In the 2009 KDD data science competition (https://kdd.org/kdd-cup/view/kdd-cup-2009), the top 3 ranking solutions started the challenge by drastically reducing the feature space. They did so by removing constant, quasi-constant, and duplicated variables.

Constant features are those that show the same value in all observations. Quasi-constant features show the same value in most of the observations. Duplicated features are 2 or more variables that have identical values on each row.

Removing constant, low variance (i.e., quasi-constant), and duplicated features is an easy first step towards reducing the feature space of any given dataset. Hence, in this chapter, we'll discuss:

- Removing constant features.
- Removing quasi-constant features.
- Removing duplicated features.

2.1 Removing constant features

A feature is constant if it has the same value in all of the observations. Features with no variability are not useful for machine learning, so we can safely remove them from the data. In this section, we will remove constant features utilizing Scikit-learn, Feature-engine and pandas. We will use the following classes and methods:

- `VarianceThreshold` from Scikit-learn.
- `DropConstantFeatures` from Feature-engine.
- `nunique()` and `std()` from pandas.

Let's start by importing the libraries, functions, and classes:

```
import pandas as pd

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold
```

```
from feature_engine.selection import DropConstantFeatures
```

Let's create a toy dataframe with 1000 observations and 10 variables, and then replace the values in 3 of the variables by 1, to create constant features:

```
# Create numpy arrays with 1000 observations and 10 features.
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_classes=2,
    random_state=10,
)

# Convert arrays to a dataframe and a series.
X = pd.DataFrame(X)
y = pd.Series(y)

# Create constant features.
X[[0, 5, 9]] = 1
```

Let's separate the dataset into a training and a testing set:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

> **!** Important
>
> Constant variables should be identified in the training set to avoid leaking information onto the test set.

Scikit-learn

`VarianceThreshold` from Scikit-learn removes features with low variance. By default, it removes features with 0 variance, that is, constant features.

> **Note**
>
> `VarianceThreshold` only works with numerical variables. Categorical variables need to be encoded first. The fastest and most convenient way is to use `OrdinalEncoder` to replace the categories with numbers when removing constant features.

To remove constant features with `VarianceThreshold`, we set the `threshold` to 0:

```
sel = VarianceThreshold(threshold=0)
```

With the method `fit()`, the transformer identifies the constant features in the training set:

```
sel.fit(X_train)
```

After fitting `VarianceThreshold`, we can use the attribute `get_support()` to identify the features that will be retained. If we execute `sum(sel.get_support())`, we will see that `7` features will be retained, which are the non-constant variables in our dataframe.

We can find the names of the constant features utilizing `get_support()` in combination with the variable names. If we execute `X_train.columns[~sel.get_support()]`, we will obtain the following array with the constant variables: `[0, 5, 9], dtype='int64'`.

To remove the constant features from the dataset, we use the `transform()` method:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

By executing `X_train_t.shape, X_test_t.shape`, we see that the constant features were removed: `((700, 7), (300, 7))`. We now have 7 features in the data, and originally there were 10.

> **Tip**
>
> Scikit-learn transformers return numpy arrays by default. We can enforce returning pandas dataframes by setting the output to pandas when setting up the transformer: `VarianceThreshold(threshold=0).set_output(transform="pandas")`.

Let's convert the arrays back to dataframes:

```
X_train_t = pd.DataFrame(X_train_t, columns=sel.get_feature_names_out())
X_test_t = pd.DataFrame(X_test_t, columns=sel.get_feature_names_out())

X_train_t.head()
```

The previous code returns the following dataframe:

	x1	x2	x3	x4	x6	x7	x8
0	0.039801	1.501392	-0.189240	1.546828	-1.831193	1.919634	0.209412
1	-0.078494	-1.536507	-0.496806	0.965100	-0.873804	-1.246872	0.629114
2	-0.731712	0.972453	-0.309300	-1.432922	-0.419046	-0.975984	0.377169
3	-0.121187	0.516685	-0.800862	-0.736170	-1.219396	-2.312341	-1.027631
4	-2.089187	0.899235	-0.241111	1.287536	0.643273	-2.310912	0.085618

Figure 2.1: First five rows of the reduced dataframe.

Note that the column names were changed. This is because in the original data they were numeric, and Scikit-learn likes working with strings as column names.

Feature-engine

We can remove constant features by utilizing `DropConstantFeatures` from Feature-engine. `DropConstantFeatures`, instead of looking at the variance of the variable, evaluates the number of unique values. Hence, it can identify constant numerical and categorical variables simultaneously. By default, it removes variables with only 1 value.

To remove constant variables, we set the tolerance (`tol`) to 1:

```
sel = DropConstantFeatures(tol=1)
```

With `fit()` the transformer identifies the constant variables in the training set:

```
sel.fit(X_train)
```

The constant features are stored in the transformer's attribute `features_to_drop_`, so if we execute `sel.features_to_drop_`, we will find a list with the constant features' names: `[0, 5, 9]`.

With `transform()` we can remove the constant variables from the datasets:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

By executing `X_train_t.shape, X_test_t.shape` we can corroborate that the variables were removed:

```
((700, 7), (300, 7))
```

We can display the resulting training set by executing `X_train_t.head()`, which will display the following:

	1	2	3	4	6	7	8
105	0.039801	1.501392	-0.189240	1.546828	-1.831193	1.919634	0.209412
68	-0.078494	-1.536507	-0.496806	0.965100	-0.873804	-1.246872	0.629114
479	-0.731712	0.972453	-0.309300	-1.432922	-0.419046	-0.975984	0.377169
399	-0.121187	0.516685	-0.800862	-0.736170	-1.219396	-2.312341	-1.027631
434	-2.089187	0.899235	-0.241111	1.287536	0.643273	-2.310912	0.085618

Figure 2.2: First five rows of the reduced dataframe.

> **Note**
>
> Feature-engine returns a dataframe by default preserving the names of the original variables.

Pandas

We can identify and remove constant features utilizing pandas. We can find features with 0 variance using a list comprehension and the `std()` method:

```
constant_features = [
    feat for feat in X_train.columns if X_train[feat].std() == 0
]
```

If we execute `len(constant_features)` we will see `3` which is the number of variables with zero variance in the dataset.

> **Note**
>
> The method `.std()` only works with numerical variables.

To remove constant numerical and categorical variables simultaneously, we can use the method `nunique()`, which returns the number of unique values in a variable:

```
constant_features = [
    feat for feat in X_train.columns if X_train[feat].nunique() == 1
]
```

Finally, let's remove the variables from the datasets:

```
X_train_t = X_train.drop(labels=constant_features, axis=1)
X_test_t = X_test.drop(labels=constant_features, axis=1)
```

That's it! We have now removed constant variables with 3 different Python open-source libraries.

2.2 Removing quasi-constant features

Variables with a large percentage of identical observations are considered quasi-constant or low variability features. In most cases, we can remove quasi-constant variables without sacrificing the performance of the predictive algorithm.

> **Tip**
>
> In some cases, particularly in those where the target is highly imbalanced, those few observations in which the values differ can be very important. In my experience, quasi-constant features are rarely ranked high by machine learning models, but they could be useful to set up rules to flag important observations before passing the data through the model.

We can remove quasi-constant features utilizing Scikit-learn, Feature-engine and pandas, like we did for constant features. We just need to use different thresholds.

Let's begin by importing the libraries, functions, and classes:

```
import random
import pandas as pd

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import VarianceThreshold

from feature_engine.selection import DropConstantFeatures
```

Let's create a toy dataset with 1000 observations and 10 variables. We will re-define 3 of the variables so that they show the same value in most, but not all of the observations.

```
# Create numpy arrays with 1000 observations and 10 features.
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_classes=2,
    random_state=10,
)

# Convert arrays to a dataframe and a series.
X = pd.DataFrame(X)
y = pd.Series(y)

# Create quasi-constant features.
random.seed(10)
X.iloc[random.sample(range(0, 1000), 990), [5, 7, 9]] = 1
```

To better understand quasi-constant features, let's count the number of occurrences of each value of feature 5:

```
X[5].value_counts()
```

In the following output, we see that 990 observations show the value 1, and only 10 observations show different values:

```
 1.000000    990
-1.262250      1
 0.110908      1
 1.226226      1
 0.570835      1
```

```
 1.645921      1
-0.182120      1
-1.417213      1
-0.272211      1
-0.112173      1
-1.296468      1
Name: 5, dtype: int64
```

> **Important**
>
> Quasi-constant or low variability features should be identified in the training set to avoid leaking data to the test set.

Let's separate the dataset into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

Scikit-learn

`VarianceThreshold` from Scikit-learn removes features with low variance.

> **Note**
>
> It is hard to define what constitutes low variance. One way to approach it is through data exploration. Another way is to optimize the threshold for low variance with hyperparameter optimization.

For our toy dataset, a threshold of variability of 0.2 should identify the quasi-constant features, so let's set `VarianceThreshold` to identify variables whose variability is lower than 0.2, and also enforce the transformer to return pandas dataframes:

```
sel = VarianceThreshold(threshold=0.2).set_output(transform="pandas")
```

The method `fit()` triggers the identification of the quasi-constant variables:

```
sel.fit(X_train)
```

After fitting the transformer, we can find the quasi-constant features by combining the variable names with the `get_support()` attribute. If we execute `X_train.columns[~sel.get_support()]` we will obtain an array with the indexes of the quasi-constant variables: `[5, 7, 9], dtype='int64'`. If we execute `sum(sel.get_support())` we will obtain `7`, which is the number of features that will be retained.

With `transform()` we can remove the quasi-constant variables from the data:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

> **Note**
>
> Note that with `set_output(transform="pandas")`, we set the output of `transform()` to a pandas dataframe instead of a numpy array.

Now, we can examine the resulting dataframe by executing `X_test_t.head()`, which displays the following output:

	x0	x1	x2	x3	x4	x6	x8
105	-1.155673	0.039801	1.501392	-0.189240	1.546828	-1.831193	0.209412
68	0.404169	-0.078494	-1.536507	-0.496806	0.965100	-0.873804	0.629114
479	0.360850	-0.731712	0.972453	-0.309300	-1.432922	-0.419046	0.377169
399	0.816893	-0.121187	0.516685	-0.800862	-0.736170	-1.219396	-1.027631
434	1.129063	-2.089187	0.899235	-0.241111	1.287536	0.643273	0.085618

Figure 2.3: First 5 rows of the reduced dataframe.

> **Note**
>
> `VarianceThreshold` works only with numerical variables. If you have categorical variables you need to encode them first, however, the encoding method will affect the variability of the variable. So if you have categorical variables, I'd suggest you use Feature-engine or pandas instead.

Feature-engine

To remove quasi-constant numerical and categorical variables simultaneously, we can use `DropConstantFeatures` from Feature-engine.

`DropConstantFeatures` identifies the percentage of observations that show just 1 value. For example, if we set the `tol` parameter to `0.95`, then, if more than 95% of the observations have the same value, the feature will be classified as quasi-constant.

> **Tip**
>
> Setting up a threshold or tolerance for `DropConstantFeatures` is more intuitive than estimating the minimum variability required to drop features with `VarianceThreshold`.

```
sel = DropConstantFeatures(tol=0.95)
```

The method `fit` finds the quasi-constant variables in the training set:

```
sel.fit(X_train)
```

The names of the quasi-constant variables can be identified by executing `sel.features_to_drop_`, which will return `[5, 7, 9]`.

To drop the variables we use `transform()`:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

> **Note**
>
> `DropConstantFeatures` returns a dataframe without the quasi-constant variables. To check out the result, execute `X_train_t.head()`.

Pandas

To capture the names of variables with low variance, we can use a list comprehension utilizing pandas `std()`, setting the threshold for variability at `0.2`:

```
constant_features = [
    feat for feat in X_train.columns if X_train[feat].std() <= 0.2
]
```

The method `std()` works only for numerical variables. To identify quasi-constant variables in both numerical and categorical features, we can use `nunique()`. In the following code, we'll count the number of occurrences of every value in every variable, and then check if the most frequent value occurs in more than 95% of the observations; if it does, the feature is added to a list for removal:

```
# Create an empty list.
quasi_constant_feat = []

# Iterate over every feature.
for feature in X_train.columns:

    # Find the predominant value.
    predominant = X_train[feature].value_counts(
        normalize=True).sort_values(
            ascending=False).values[0]

    # Evaluate the predominant feature: do more
    # than 95% of the observations show 1 value?
    if predominant > 0.95:

        # If yes, add the variable to the list.
        quasi_constant_feat.append(feature)
```

> **Tip**
>
> The previous pandas implementation is what `DropConstantFeatures` does under the hood.

If we now execute `len(quasi_constant_feat)` we will obtain `3`, which is the number of quasi-constant variables in our data. If we execute `quasi_constant_feat`, we will obtain a list with the variable names: `[5, 7, 9]`.

Let's now remove the variables from the data:

```
X_train_t = X_train.drop(labels=quasi_constant_feat, axis=1)
X_test_t = X_test.drop(labels=quasi_constant_feat, axis=1)
```

And that's it. We have removed quasi-constant variables from the data with 3 different Python open-source libraries.

2.3 Removing duplicated features

Duplicated features are variables that are identical. We can identify and remove duplicate features with Feature-engine and pandas.

Let's begin by importing the libraries, functions and classes:

```
import pandas as pd

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

from feature_engine.selection import DropDuplicateFeatures
```

Let's create a toy dataset with 1000 observations and 10 features, 6 of which are duplicates:

```
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_repeated=6,
    n_classes=2,
    random_state=10,
)

X = pd.DataFrame(X)
y = pd.Series(y)
```

> **!** Important
>
> Duplicated variables should be identified in the train set to avoid leaking data to the test set.

Let's separate the dataset into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

Feature-engine

`DropDuplicateFeatures` from Feature-engine finds and removes features that are copies of each other. Let's set up the transformer:

```
sel = DropDuplicateFeatures()
```

The method `fit()` makes the transformer find the duplicates in the training set:

```
sel.fit(X_train)
```

After fitting the transformer, we can see the names of the features that will be removed by executing `sel.features_to_drop_`, which returns `{4, 5, 6, 7, 8, 9}`.

We can also identify which variables were duplicates by executing `sel.duplicated_feature_sets_`, which returns the following list of sets:

`[{0, 7}, {1, 6, 8}, {2, 5, 9}, {3, 4}]`.

Each set corresponds to a group of duplicated features. That is, features 0 and 7 are duplicates, features 1, 6 and 8 are duplicates, and so on.

`DropDupicateFeatures` retains the first feature from each set, and drops the rest.

We can drop the duplicated variables with `transform()`:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

We can now examine the shape of the resulting dataframes:

```
X_train_t.shape, X_test_t.shape
```

The previous command returns the shape of the datasets without the duplicates:

```
((700, 4), (300, 4))
```

We can visualize the resulting dataframe by executing `X_train_t.head()`, which returns the following output:

	0	1	2	3
105	-3.189671	-1.672739	1.128395	-1.101922
68	0.655279	-1.253133	-0.499965	-0.253241
479	1.719269	-2.113632	-1.114576	-0.311732
399	-1.157952	-1.531827	0.254379	-0.677742
434	2.744846	2.705456	-0.758431	1.328512

Figure 2.4: First 5 rows of the reduced dataframe.

Pandas

We can also find duplicated features manually with pandas. In the following code, we iterate over every feature, and compare it with the remaining features in the dataset. Duplicates are added to a list for removal:

```
# Create an empty dictionary where we store
# the groups of duplicates.
duplicated_feat_groups = {}

# Create an empty list to collect features
# that are found to be duplicated.
_duplicated_feat = []


# Iterate over every feature in our dataset.
for i in range(0, len(X_train.columns)):

    # Choose 1 feature.
    feat_1 = X_train.columns[i]

    # Check if the feature has already been identified
    # as a duplicate of another one.

    # If this feature was already identified as a duplicate, skip it. If
    # it has not yet been identified as a duplicate, then proceed:
    if feat_1 not in _duplicated_feat:
```

```
        # Create an empty list as an entry for this feature in the dictionary.
        duplicated_feat_groups[feat_1] = []

        # Iterate over the remaining features of the dataset.
        for feat_2 in X_train.columns[i + 1:]:

            # Check if the second feature is identical to the first one.
            if X_train[feat_1].equals(X_train[feat_2]):

                # If it is identical, append it to the list in the dictionary.
                duplicated_feat_groups[feat_1].append(feat_2)

                # Append duplicated feature to our list for
                # duplicated variables.
                _duplicated_feat.append(feat_2)
```

If we execute `len(_duplicated_feat)`, we obtain `6`, which is the number of duplicated variables. If we execute `_duplicated_feat` we obtain the names of the features to remove: `[7, 6, 8, 5, 9, 4]`.

The previous code block captured the groups of duplicated features in the following variable: `duplicated_feat_groups`. If we execute `duplicated_feat_groups`, we'll obtain the following output where the keys are the features that will be retained, and the values are duplicates of those features.

```
{0: [7], 1: [6, 8], 2: [5, 9], 3: [4]}
```

> **Tip**
>
> If the datasets are big, finding duplicated features may take some time to process. Feature-engine's implementation runs faster than the pandas code shown in the book.

Let's now remove the duplicates:

```
X_train_t = X_train[duplicated_feat_groups.keys()]
X_test_t = X_test[duplicated_feat_groups.keys()]
```

And that's it! We have now removed duplicated features with Feature-engine and Pandas.

2.4 References

- Gideon et. al., Winning the KDD Cup Orange Challenge with Ensemble Selection, JMLR Workshop and Conference Proceedings, 7:23–34, 2009.
- Xie et. al., A Combination of Boosting and Bagging for KDD Cup 2009 Fast Scoring on a Large Database, JMLR Workshop and Conference Proceedings 7:35–43, 2009.
- Miller et. al., Predicting customer behaviour: The University of Melbourne's KDD Cup report, JMLR Workshop and Conference Proceedings 7:45–55, 2009.

3 Correlation of Predictors

Correlation is a statistical measure of the degree of association between two random variables. When two variables are correlated, by knowing the values of one of the variables, we can infer the values of the second one. When these variables are predictors in machine learning models, one of the variables already provides all, or most of the predictive information to estimate the target. Thus, the second variable is redundant, and we can remove it.

There are several methods to estimate correlation. The most common is **Pearson's correlation coefficient**, which measures the degree of *linear association* between two variables. Other correlation coefficients, like **Spearman's rank correlation coefficient** or **Kendall's Tau correlation coefficient**, measure nonlinear associations between variables.

Correlation coefficients vary between -1 and 1. A value of 1 indicates a perfect positive correlation and a value of -1 indicates a perfect negative correlation. Positive correlation means that the greater the values of one variable, the greater the values of the second variable. A negative correlation implies that the greater the values of one variable, the smaller the values of the second variable. A correlation coefficient of 0 indicates that the variables do not have a relationship or association.

In this chapter, we will first discuss how to calculate the different correlation coefficients in more detail. Then, I'll show how to visualize correlated variables. And finally, I'll introduce 2 approaches to remove correlated features. In the first approach, we examine features as they appear in the dataset. In the second approach, we first find groups of correlated variables, and then select the most predictive feature from each group. I'll close the chapter with a discussion about correlation among categorical variables.

In summary, we'll discuss:

- Correlation coefficients.
- Visualizing correlated features.
- Removing correlated features: retain first.
- Removing correlated features: retain best.
- Correlation of categorical variables.

3.1 Correlation coefficients

There are 3 main statistics to estimate correlation between variables:

- Pearson's correlation coefficient.
- Spearman's rank correlation coefficient.
- Kendall's tau correlation coefficient.

Pearson's correlation coefficient

Pearson's correlation coefficient is a measure of the linear association between two random variables and is given by:

$$R = \sum \frac{(X - \overline{X})(Y - \overline{Y})}{\sigma_x \sigma_y}$$

where X and Y are the random variables, $\overline{X}$ and $\overline{Y}$ are the mean of each variable, and σ_x and σ_y are the standard deviations of X and Y, respectively.

Pearson's correlation coefficient tells us that the more one variable deviates from its mean, the more the other variable deviates from its mean, when its absolute value is closer to 1.

Spearman's correlation coefficient

Spearman's rank correlation coefficient is similar to Pearson's correlation coefficient, except that X and Y are the ranks of the variable values instead of the values themselves.

The ranks are assigned based on the magnitude of the values. The highest value takes rank 1, the subsequent value takes rank 2, and so on. Values of equal magnitude share the rank. So, for example, for a feature with the following values: `f1 = [10, 8, 8, 5, 3]`, the vector X takes the values `X = [1, 2, 2, 3, 4]`, and for a different feature with values `f2 = [500, 300, 250, 250, 100]`, the vector Y is `Y = [1, 2, 3, 3, 4]`.

The calculation of the coefficient is then done, utilizing ranks instead of the real values of the variables. With this formulation, Spearman's rank coefficient is able to capture non-linear, but monotonic relationships.

Kendall's tau coefficient

Kendall's tau coefficient also utilizes ranks to find the association, and it is defined by:

$$R = \frac{(C - D)}{(C + D)}$$

where C is the number of concordant pairs and D is the number of discordant pairs. Concordant pairs are those that share the same rank. Discordant pairs are those whose ranks are different. So for the vectors with the ranks `X = [1, 2, 2, 3, 4]` and `Y = [1, 2, 3, 3, 4]`, we have 4 concordant pairs and 1 discordant pair. Kendall's tau also captures non-linear, monotonic associations between variables.

> Tip
>
> **When should we use each correlation metric?**
> If we suspect a linear association between variables or if we want to create linear regression models, it is a good idea to evaluate variable association by utilizing Pearson's correlation coefficient. Alternatively, if we want to create non-linear models like those based on decision trees, or if we know beforehand that our variables are not linearly associated, we can assess their association with Spearman's or Kendall's coefficients.

3.2 Visualizing correlated features

We can determine variable associations through correlation coefficients. In addition, making heatmaps with these coefficients and scatter plots with variable pairs help us better understand the relationships between the variables and the degree of multicolinarity in our dataset.

In a scatter plot, we plot the values of one variable against the values of the other to visualize their relationship. A heatmap represents the relationship among all the variables in a dataset. It consists of a color matrix that shows the correlation of all possible feature pairs in the dataset. The X and Y axis contain the features, the diagonal highlights the correlation of a feature with itself and is therefore always 1, and the remaining intersections represent the correlation between feature pairs, which is displayed in colours: more intense colors highlight greater correlations.

Python implementation

We'll start by determining the correlation between variables using Pearson's correlation coefficient, and then create correlation heatmaps and scatter plots.

> **Note**
>
> In this demo, I use Pearson's correlation coefficient. However, remember that you can find non-linear associations using alternative correlation coefficients.

We will use pandas to calculate the correlation, seaborn for the heatmap, and matplotlib for the scatter plots. Let's import the libraries and functions:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

Let's create a toy dataset with 1000 observations and 10 features, 7 of which are correlated:

```
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_redundant=7,
    n_classes=2,
    random_state=10,
)

X = pd.DataFrame(X)
y = pd.Series(y)
```

Let's split the dataset into a training and a testing set, which is what we would do if we were selecting features for machine learning:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

> **Important**
>
> Correlated variables should be identified in the training set to avoid leaking information to the test set.

We can calculate the correlation of all variable pairs automatically by using pandas `corr()`. The method `corr()` can determine Pearson's, Spearman's, and Kendall's correlation coefficients.

Let's create a matrix with the Pearson's correlation coefficients for all variable pairs:

```
corrmat = X_train.corr(method="pearson")
```

After calculating the correlation coefficients, we can plot a heatmap with those values to visually identify correlated variables:

```
fig, ax = plt.subplots()
fig.set_size_inches(10, 8)
cmap = sns.diverging_palette(220, 20, as_cmap=True)
sns.heatmap(corrmat, cmap=cmap, annot=True)
```

The previous code returns the following plot:



Figure 3.1: Variable correlation heatmap.

The axes of the plot contain the feature names, which for our data are integers from 0 to 9. The dark orange squares correspond to highly and positively correlated features (R>0.8). Darker blue squares correspond to highly and negatively correlated features (R<-0.8). The diagonal represents the correlation of a feature with itself, and therefore the value is 1.

We can see in the plot that there are several features that are highly correlated: take a look at the number of darker orange and darker blue squares. To train machine learning models, we could probably remove some of the redundant features, without sacrificing model performance.

Another way of visualizing variable correlation is through scatter plots, where we plot the values of one variable against the values of a second variable.

Let's plot 2 variables with a high positive correlation, like variables `6` and `8`:

```
plt.scatter(X_train[8], X_train[6], alpha=0.3, color="purple")
plt.xlabel("var_8")
plt.ylabel("var_6")
plt.show()
```

In the following image, we can see that the greater the values of variable `var_8`, the greater the values of variable `var_6` and the values of both variables lie approximately along a straight line:

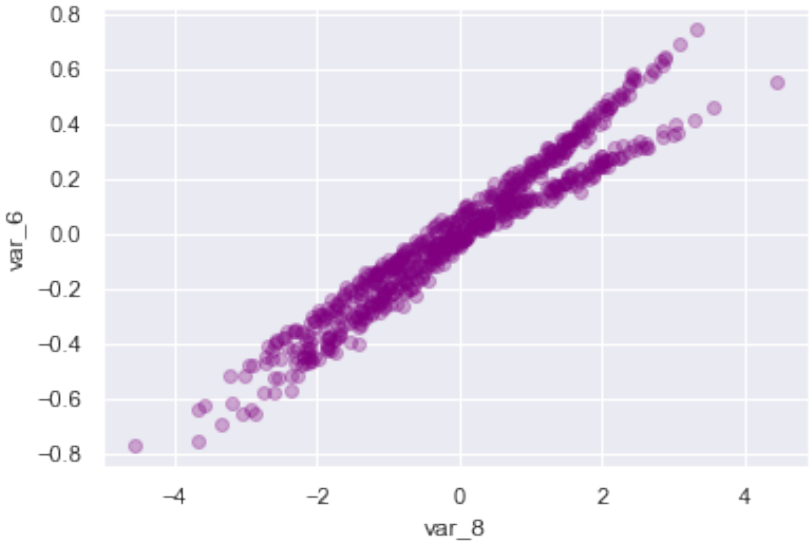


Figure 3.2: Scatterplot: positively correlated variables.

Let's now plot negatively correlated variables, like variables `5` and `7`:

```
plt.scatter(X_train[7], X_train[5], alpha=0.3, color="purple")
plt.xlabel("var_7")
plt.ylabel("var_5")
plt.show()
```

In the following plot, we see that the greater the values of variable `var_7`, the smaller the values of variable `var_5`, and again, the values lie approximately on a straight line:

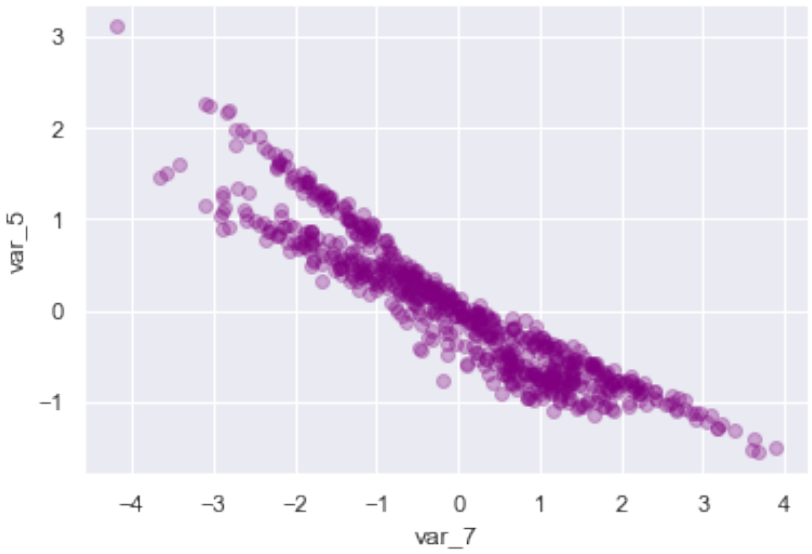


Figure 3.3: Scatterplot: negatively correlated variables.

That's it! Now you know how to identify and visualize correlated variables.

3.3 Removing correlated features: retain first

When variables are correlated, one of the variables is usually enough to predict the outcome. In fact, linear regression models assume that there is no, or little colinearity among the predictors. Hence, when we have high-dimensional data and we want to train linear models, we can reduce the feature space by removing correlated variables.

The most straightforward (though probably not the best) approach to remove correlated variables is what I call the *"brute force"* approach. Here, we examine feature per feature as they appear in the dataset, and remove all variables that are correlated with them.

Let's illustrate this with an example. Say we have a dataset with 10 variables, x1 to x10. We start by examining x1, and find that features x3, x5, and x8 are correlated with it, so we remove them. Then, we move on to variable x2, and find that features x4 and x9 are correlated with it, so we remove them. Now we move to x6, and find that x10 is correlated with it. After this process, the selected variables are x1, x2, x6 and x7.

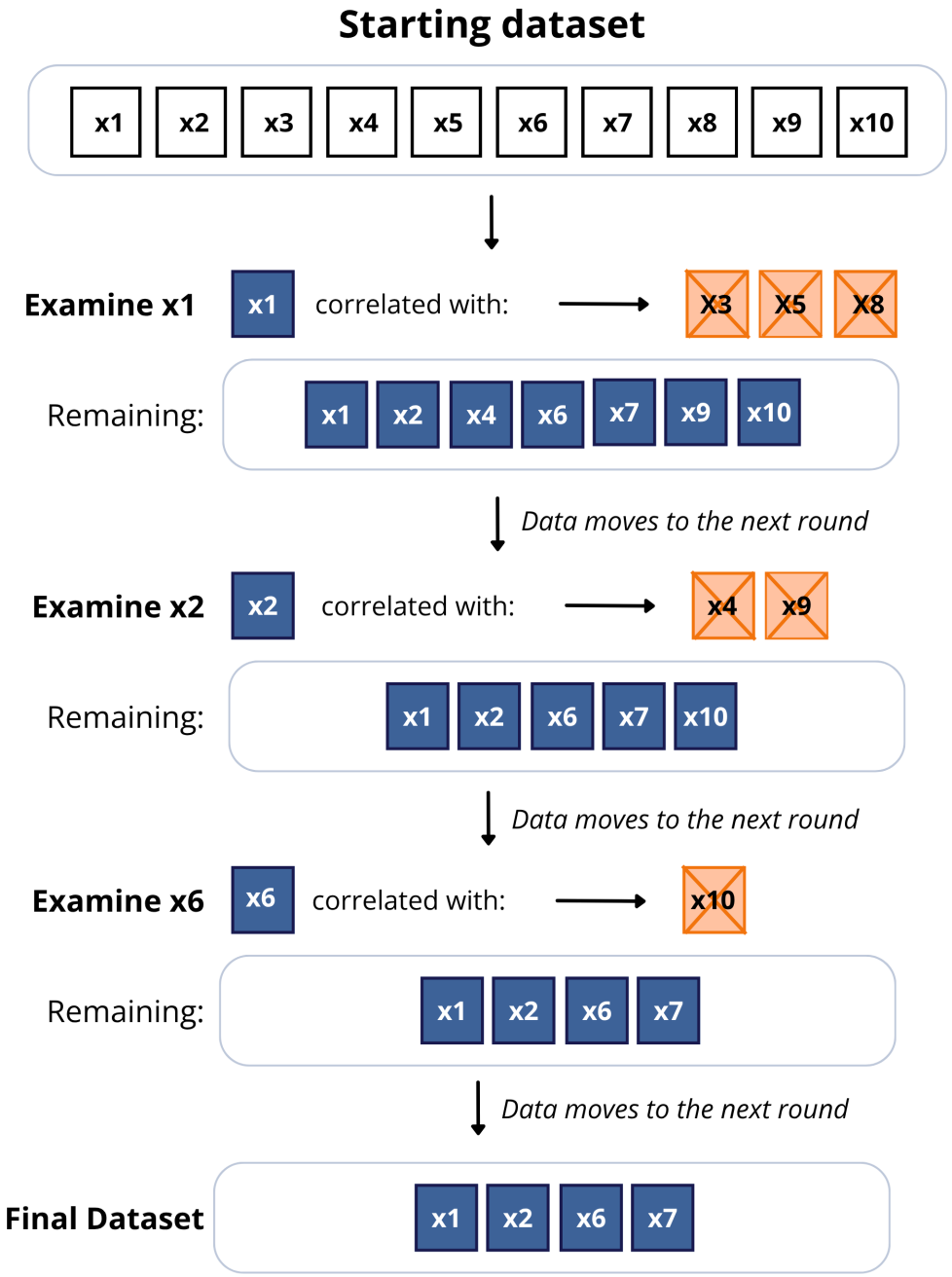


Figure 3.4: Remove correlated features: retain first or brute force approach.

This method is fast and straightforward and ensures that the remaining features have little multicolinearity. On the downside, from a group of correlated features, it may not retain the most predictive one.

> ⚠ Warning
>
> This method is non-deterministic: if the features are entered in a different order, we'll obtain a different subset of selected variables.

Python implementation

In the following Python implementation, we will remove correlated features using the brute force approach. We'll first use the open-source Python library Feature-engine, and then, we'll implement the entire process with pandas.

Let's start by importing the libraries, functions and classes:

```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from feature_engine.selection import DropCorrelatedFeatures
```

Let's create a toy dataset with 1000 observations and 10 features, 7 of which are correlated:

```
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_redundant=7,
    n_classes=2,
    random_state=10,
)

X = pd.DataFrame(X)
y = pd.Series(y)
```

Let's separate the data into a training and a testing set:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

> **Important**
>
> Correlated variables should be identified in the training set to avoid leaking information to the test set.

Feature-engine

`DropCorrelatedFeatures` from Feature-engine identifies correlated features based on any of the correlation coefficients that we described at the beginning of this chapter. It removes features whose correlation is greater than a threshold determined by the user.

In the next code block, we identify features that are linearly related, i.e., we use Pearson's correlation coefficient, with an absolute correlation coefficient greater than 0.8:

```
sel = DropCorrelatedFeatures(method="pearson", threshold=0.8)
```

> **Tip**
>
> By changing the method to "spearman" or "kendall" we can find non-linear associations between the variables.

The method `fit()` finds the correlated features that should be removed:

```
sel.fit(X_train)
```

> **Important**
>
> I mentioned previously that this brute-force approach of finding correlated features is non-deterministic, in that, if the order of the features changes, we'd select a different feature subset. To make the selection process deterministic, `DropCorrelatedFeatures` sorts the features alphabetically before starting with the search.

By executing `sel.features_to_drop_`, we find the features that will be removed: `{3, 4, 5, 6, 7, 8, 9}`.

We can also find the groups of correlated features as follows:

```
sel.correlated_feature_sets_
```

The previous command returns the following lists:

```
[{0, 3, 4, 5, 7, 9}, {1, 6, 8}]
```

Each set in the list contains a group of features that are correlated. For example, features 3, 4, 5, 7 and 9 are correlated with feature 0. Features 6 and 8 are correlated with 1. Hence, there are 2 groups of correlated features in the dataset.

To identify which feature will be retained from each group, we can execute:

```
sel.correlated_feature_dict_
```

The output of the previous command is:

```
{0: {3, 4, 5, 7, 9}, 1: {6, 8}}
```

The dictionary contains in its keys the features that will be retained, and the values contain the list of features that are correlated with those in the keys, and will therefore be removed.

> **Tip**
>
> Go ahead and make scatter plots displaying the features in the keys of the dictionary against the features that correspond to their values.

To remove the features we use the `transform()` method:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

We can corroborate the reduced shape of the resulting dataframes by executing `X_train_t.shape, X_test_t.shape`, which will return `((700, 3), (300, 3))`.

We can also visualize the first 5 rows of the reduced data by executing `X_train_t.head()`:

	0	1	2
105	1.693553	-3.715633	0.250835
68	-1.470185	3.029661	-1.979157
479	-0.483779	1.192578	1.439996
399	1.409885	-0.505949	-0.049844
434	-1.085478	1.894522	-1.161771

Figure 3.5: First 5 rows of the data without correlated features.

Pandas

To remove correlated features with pandas, we can create a function that first determines the correlation between all features, and then implement the brute force approach to remove correlated variables.

In the following function, we first create a correlation matrix using pandas `corr()`. Then, we loop over every feature, and find those that are correlated with it:

```
def correlation(dataset, threshold):

    # Create a set where we store the correlated variables.
    col_corr = set()

    # Create the correlation matrix.
    corr_matrix = dataset.corr()

    # Iterate over each feature in the dataset
    # (columns of the correlation matrix).
    for i in range(len(corr_matrix.columns)):

        # Compare a feature to the other features.
        for j in range(i):

            # Check if the correlation is greater than a threshold
            # (we are interested in the absolute correlation value).
            if abs(corr_matrix.iloc[i, j]) > threshold:
```

```
                # Get the name of the correlated feature.
                colname = corr_matrix.columns[j]

                # Add it to the correlated set.
                col_corr.add(colname)

    return col_corr
```

Let's find the features in the training set with an absolute correlation coefficient greater than 0.8 using the previous function:

```
corr_features = correlation(X_train, 0.8)
```

If we execute `len(set(corr_features))`, we obtain `7`, which is the number of correlated variables. If we execute `corr_features`, we will see the names of the correlated features: `{0, 1, 3, 4, 5, 6, 7}`.

We can now drop those features from the data as follows:

```
X_train_t = X_train.drop(labels=corr_features, axis=1)
X_test_t = X_test.drop(labels=corr_features, axis=1)
```

The methods described in this section removed correlated features without further insight. From a group of correlated variables, the first one was retained, and the rest were removed. In the next section, we will see how we can select the best feature from a group of correlated variables instead.

3.4 Removing correlated features: retain best

We often find groups of features that are correlated. In the brute-force approach, we retain the first feature and remove the rest. A best approach consists of retaining the feature that brings most value to the predictive model and removing the rest.

To illustrate this with an example, say we have a dataset with 10 variables, x1 to x10. We start by examining x1, and find all features that are related to it; in this case features x3, x5, and x8. Then, we move on to variable x2, and find the features that are correlated with it; in our example, x4 and x9. Now we move to x6, and find that x10 is correlated with it. After this process, we have 3 groups of correlated features. From each group, we find that features x3, x4 and x6 are the most valuable for our model, so we keep those, together with feature x7, and remove the rest.

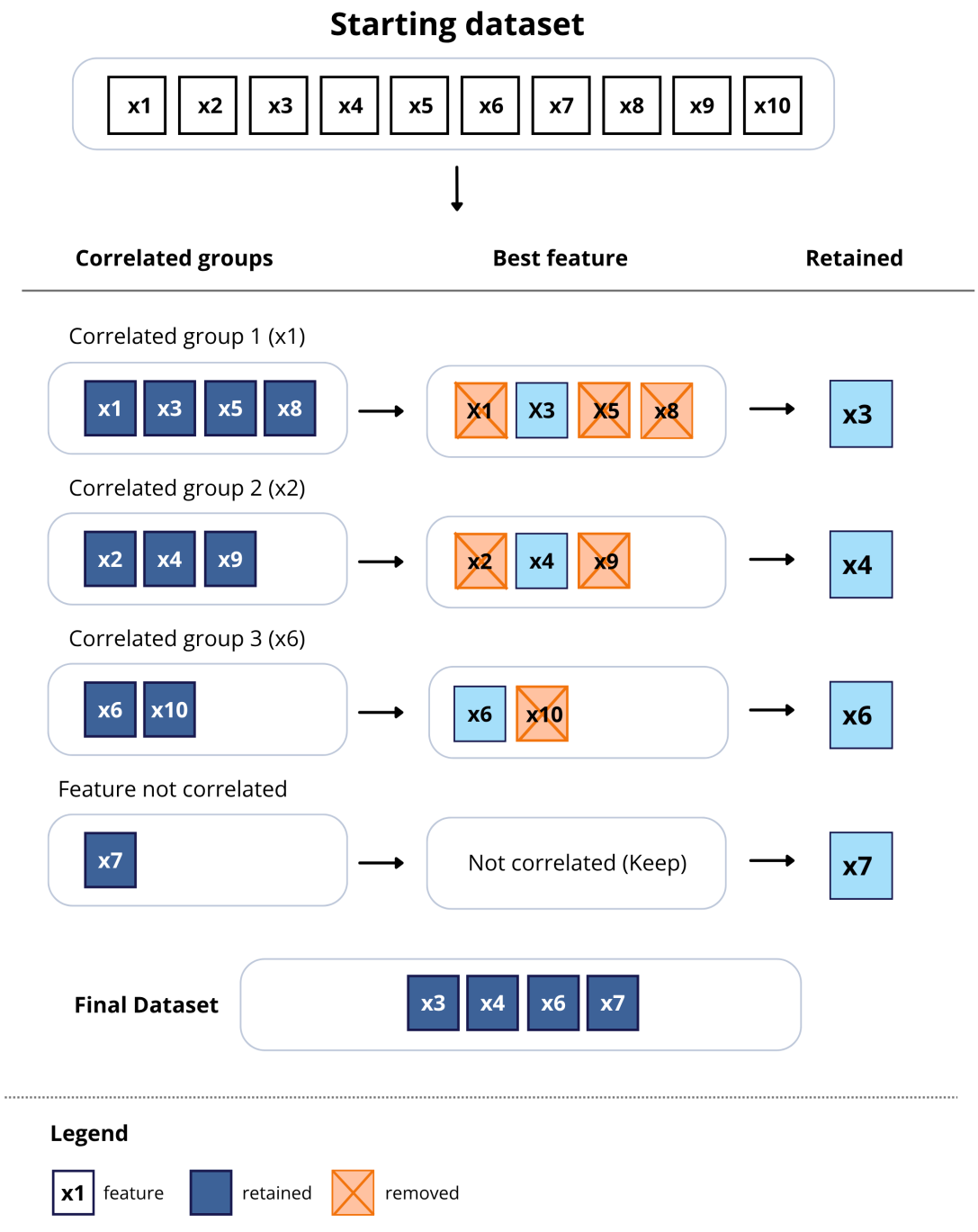


Figure 3.6: Remove correlated features: retain best.

The question is: how do we determine which feature brings the most value to the model?

Simple ways to infer the predictive value of the feature are through its variability, its cardinality, or by the amount of missing data. The assumption is that a feature with more values (greater variability or cardinality, or fewer missing observations), contains more information about the target.

There are, however, better ways to determine the value of a feature. We could select the feature whose correlation with the target is the greatest. Similarly, we can select the feature with the greatest mutual information with the target.

Alternatively, we can train a decision tree based model using the correlated features and retain the feature whose importance derived from the decision tree is the greatest. Similarly, we can train single feature models, and retain the feature that returns the best performing model.

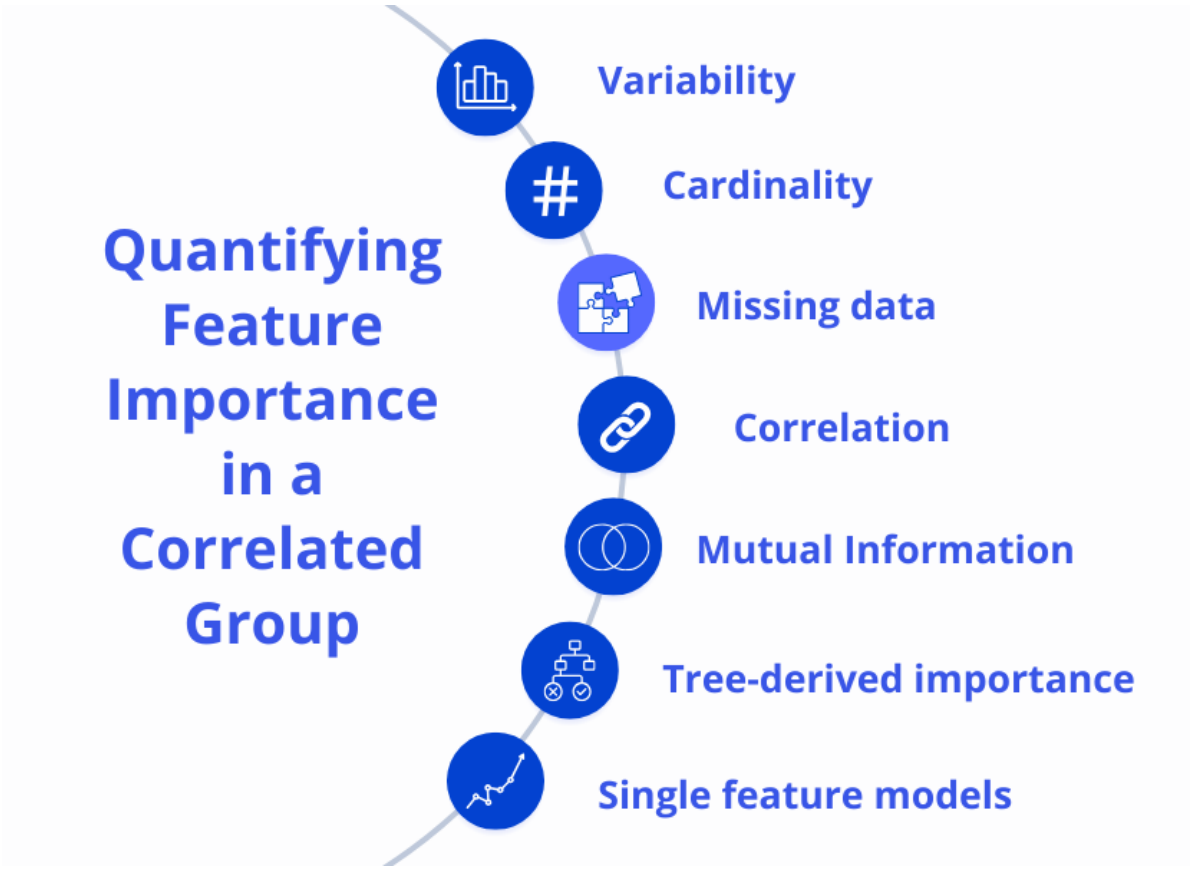


Figure 3.7: Ways to quantify feature importance in a correlated group.

Python implementation

In the following Python demo, I'll use Feature-engine to select features based of single feature models.

Let's start by importing the libraries, functions, and classes:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from feature_engine.selection import SmartCorrelatedSelection
```

Let's create a toy dataset with 1000 observations and 10 features, 7 of which are correlated:

```
X, y = make_classification(
    n_samples=1000,
    n_features=10,
    n_redundant=7,
    n_classes=2,
    random_state=10,
)

X = pd.DataFrame(X)
y = pd.Series(y)
```

Let's split the data into a training and a testing set:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

> **!** Important
>
> Correlated variables should be identified in the training set to avoid leaking information to the test set.

To select the best features from a group of correlated variables, we use `SmartCorrelatedSelection` from Feature-engine.

> **Note**
>
> Using Feature-engine, we can select the most valuable feature from each group based on variability, cardinality, missing data, and single feature models. At the time of writing, there is no functionality to select features based on correlation or mutual information with the target, or by deriving importance from embedded methods (like tree-derived importance), but keep checking the documentation, because we may add the functionality in coming months.

We'll identify groups of features showing an absolute correlation coefficient greater than 0.8. From each group, we'll rank the features using single feature random forests, and select the top ranking feature. To select features based on single feature models, we need to set the parameter `selection_method` to `"model_performance"` and then pass the model that we want to use for the selection to the parameter `estimator`.

```
sel = SmartCorrelatedSelection(
    method="pearson",
    threshold=0.8,
    selection_method="model_performance",
    estimator=RandomForestClassifier(n_estimators=5, random_state=10),
    scoring="roc_auc",
    cv=3,
)
```

> **Tip**
>
> Here, I ranked features using single feature random forests. However, training a random forest with one feature is a bit of an overkill. Training a single decision tree would have been enough to get an idea of the value of each feature, and much faster, so I suggest you do that instead.

The method `fit()` finds the correlated feature groups and the most important feature from each group:

```
sel.fit(X_train, y_train)
```

By executing `sel.features_to_drop_`, we can see the features that will be removed: `[0, 3, 4, 5, 6, 8, 9]`.

We can find the correlated feature groups by executing `sel.correlated_feature_sets_`, which returns `[{0, 3, 4, 5, 7, 9}, {1, 6, 8}]`.

To find out which feature will be retained from each group, we execute:

```
sel.correlated_feature_dict_
```

The output of the previous command is:

```
{7: {0, 3, 4, 5, 9}, 1: {6, 8}}
```

The dictionary contains in its keys the features that will be retained in the dataframe, and the values contain the list of features that are correlated with those in the keys. Hence, from the first group, feature 7 will be retained, and from the second group, feature 1 will be retained. Each `{key:pair}` contains the entire correlated group from the sets displayed with `sel.correlated_feature_sets_`.

> **Note**
>
> Correlation is not transitive; thus, all features within the group may not necessarily be correlated to each other with an absolute correlation coefficient higher than the threshold indicated when setting up `SmartCorrelatedSelection`.

We can remove the features using `transform()`:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

We can visualize the resulting data with `X_train_t.head()`:

	1	2	7
105	-3.715633	0.250835	2.115155
68	3.029661	-1.979157	-1.822263
479	1.192578	1.439996	-0.613533
399	-0.505949	-0.049844	1.577046
434	1.894522	-1.161771	-1.321103

Figure 3.8: First 5 rows of the data without correlated features.

And that's it! You now know how to first find correlated feature groups and then keep the most predictive feature from each group!

3.5 Correlation of categorical variables

The correlation coefficients discussed in this chapter are suited to numerical variables. In particular, Pearson's correlation coefficient is suitable for continuous variables, while Spearman's and Kendall's rank coefficients work for continuous and discrete variables, respectively. There are other metrics, like Cramer's V coefficient, that measure the degree of association between categorical variables, but I have not seen these coefficients applied to feature selection for machine learning.

If you have categorical data, you can encode them into numbers first, and then proceed with the selection as we discussed here, using Spearman's, or better, Kendall's correlation coefficients. But other feature selection methods, like those based on mutual information, can be better suited to select among categorical variables.

> ⚠ Warning
>
> Note that the encoding method used to replace the categorical variables with numbers will affect the relationship among the categories and also to other variables, so different encodings may lead to the selection of different feature subsets. That applies to selection based on correlation and other selection methods.

3.6 References

- Hall, Correlation-based Feature Selection for Machine Learning, 1999, Master thesis. Available at:
  - https://ml.cms.waikato.ac.nz/publications/1999/99MH-Thesis.pdf

4 Statistical Methods

Filter methods for feature selection select variables by evaluating the 'merits' of the features while disregarding the machine learning algorithm (see *Chapter 1* for more details). They typically involve a variable ranking step followed by a feature selection step based on that ranking.

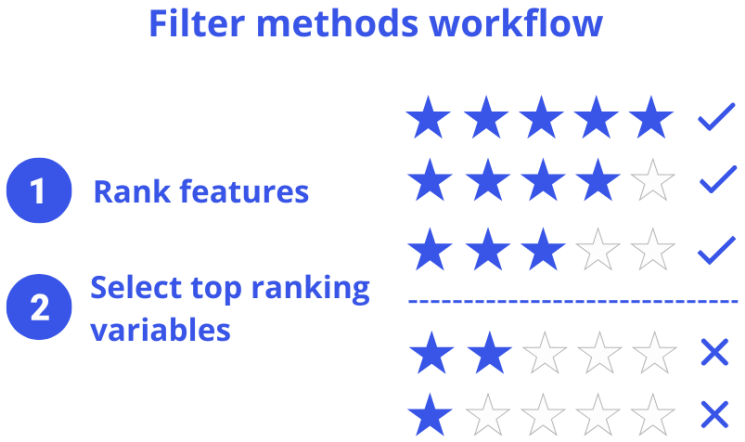


Figure 4.1: Typical filter method workflow.

Selection through variable ranking has been widely used because of its simplicity, scalability, and success in many cases.

One way to rank variables is by using statistical tests. The choice of test depends on several factors, including whether the target variable is continuous or categorical, whether the predictor variables are continuous or categorical, and the relationship between the variables and the target. Many statistical tests rely on certain assumptions about the data, so if these assumptions are not met, the tests may lose their statistical power.

In this chapter, we will discuss the use of the following statistical tests as variable ranking functions:

- chi^2 test.

- ANOVA.
- Correlation.
- Mutual information.

4.1 Chi-square

Pearson's chi-square test is used to determine the association between categorical variables. Thus, we can use this test to evaluate the association between categorical data and a binary or multiclass target.



Figure 4.2: Pearson's chi-square test is suitable to select among categorical features when the target is also categorical.

Pearson's chi-square test analyses the distribution of the observations across the categories of the variables, that is, the frequencies of the categories. It compares the *observed frequencies* against the *expected frequencies*, which are the frequencies we would obtain by chance.

To illustrate category frequencies with an example, let's say we'd like to analyse the association between eating bananas, candy, or fish and being healthy. In a fictitious experiment, we ask 97 individuals to eat 1 of the food items a day for a week, and then we assess if they are healthy or not. We obtain the following contingency table with the category frequencies:

Table 4.1: Contingency table - food vs. health (observed distribution).

	Bananas	Candy	Fish
Healthy	30	2	10
Unhealthy	10	30	15

The first cell in the table represents the number of individuals who ate bananas and are healthy, the second cell represents the number of individuals who ate candy and are healthy, and so on.

If there is an association between food and health, then we expect that the distribution of food items over health status is not given by chance. Looking at the table, we see that 30 out of 40 people who ate bananas are healthy, whereas 30 out of the 32 people who ate candy are unhealthy. Thus, we could conclude that eating bananas is healthy and eating candies is unhealthy. With fish, it is a bit harder to say because the number of individuals who eat fish and are healthy or unhealthy is similar.

Pearson's chi-square test allows us to determine which of the associations are significant. So how do we calculate significance with the chi-square test?

First, we need to calculate the degree of association between the variables, which is captured by the chi-square statistic. The chi-square statistic is calculated as:

$$X^2 = \sum \frac{(observed—expected)^2}{expected}$$

where the observed distribution is the real distribution of the observations across the categories, and the expected distribution is the distribution we would observe by chance.

Thus, to use the chi-square test to rank variables we need 2 distributions:

- The observed distribution of the categories.
- The expected distribution of the categories.

We've got the observed distribution: those in the contingency table (table 4.1). We need to define the expected distribution. If food and health are not associated, we could expect half of the individuals to be healthy and the remaining half unhealthy, irrespective of the food they ate. But this assumes that the same number of people ate each of the food times, and had the same initial health status. A more general way to calculate the expected distribution uses the columns and row totals from the contingency table as follows:

$$E = \frac{row_{total} \times column_{total}}{observations_{total}}$$

The row and column totals for **table 4.1** are:

Table 4.2: Contingency table with row and column totals.

	Bananas	Candy	Fish	**Total**
Healthy	30	2	10	42
Unhealthy	10	30	15	55
Total	40	32	25	97

With the row and column totals we can calculate the expected distributions as follows:

$E_{Healthy;Bananas} = \frac{42 \times 40}{97} = 17.32$

$E_{Healthy;Candy} = \frac{42 \times 32}{97} = 13.85$

$E_{Healthy;Fish} = \frac{42 \times 25}{97} = 10.82$

$E_{Unhealthy;Bananas} = \frac{55 \times 40}{97} = 22.68$

$E_{Unhealthy;Candy} = \frac{55 \times 32}{97} = 18.14$

$E_{Unhealthy;Fish} = \frac{55 \times 25}{97} = 14.17$

Now, we can create a contingency table with the expected distributions:

Table 4.3: Expected distribution - food vs. health.

	Bananas	Candy	Fish
Healthy	17.32	13.8	10.82
Unhealthy	22.68	18.14	14.17

With the observed (table 4.1) and expected (table 4.3) distributions, we can now calculate the value of the chi-square statistic:

$$X^2 = \frac{(30—17.32)^2}{17.32} + \frac{(2—13.85)^2}{13.85} + \frac{(10—10.82)^2}{10.82} + \frac{(10—22.68)^2}{22.68} + \frac{(30—18.14)^2}{18.14} + \frac{(55—14.17)^2}{14.17}$$

which results in:

$$X^2 = 9.28 + 10.14 + 0.06 + 7.09 + 7.74 + 0.048 = 34.37$$

X^2 follows the chi-square distribution, whose only parameter is the degrees of freedom. Below we see examples of chi-square distributions with varying degrees of freedom:

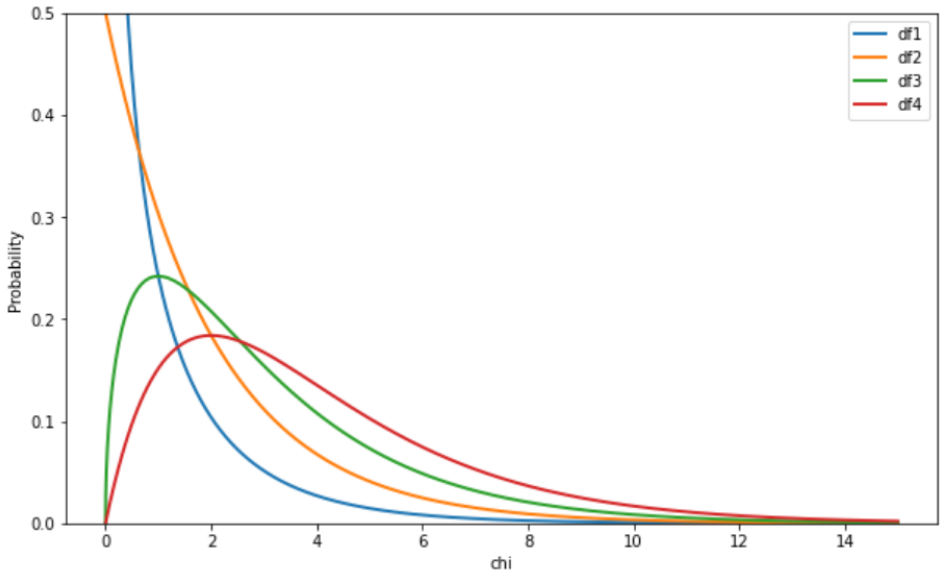


Figure 4.3: Chi-square distributions with different degrees of freedom (df).

The degrees of freedom are calculated as follows:

$$dof = (rows - 1) \times (columns - 1)$$

For our example, the degrees of freedom is 2:

$$dof = (2 - 1) \times (3 - 1) = 2$$

With the value of chi-square and the degrees of freedom, we can obtain the probability of obtaining said chi-square value if there is no association between the variables.

> **Note**
>
> The chi-square distribution is very well studied. With the chi-square statistic and the degrees of freedom we can obtain probability values using pre-recorded tables.

The p-value for this example is 0.001. The bigger the value of the chi-square statistic, the smaller the probability that the 2 variables are *not associated*. Thus, features with large chi-square statistic or small p-values have strong associations with the target. We can use either of them to rank the features.

Assumptions

Pearson's chi-square test makes some assumptions about the data:

1. The observations are independent.
2. The frequencies in the expected distribution must be greater than 5.

The first assumption means that each observation can only be assigned to one of the cells of the contingency table. The second assumption means that every cell in the contingency table with the **expected distribution** (table 4.3) must have a value greater than 5. If the assumptions are not met, the test loses statistical power.

Considerations

If we have categorical variables with high cardinality, we may obtain expected distributions with a frequency lower than 5. In such cases, we can group rare categories before carrying out the test.

If we have continuous variables and we want to assess them together with the categorical variables, we can discretize the variables first. Note, however, that the way in which we construct the intervals will affect the value of the chi-square statistic.

Python implementation

We will use Pearson's chi-square to assess the association of categorical and discrete variables in the Titanic dataset with survival.

Let's begin by importing the libraries, functions, and classes:

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import chi2_contingency
from sklearn.model_selection import train_test_split
```

Let's load a subset of the Titanic dataset with categorical and discrete variables:

```
variables = ['pclass', 'survived', 'sex', 'sibsp', 'parch', 'embarked']

data = pd.read_csv('https://www.openml.org/data/get_csv/16826755/phpMYEkMl',
                   usecols=variables,
                   na_values='?',
                   )

data.dropna(subset=['embarked'], inplace=True)
data.head()
```

Below we see the first 5 rows of the dataset:

	pclass	survived	sex	sibsp	parch	embarked
0	1	1	female	0	0	S
1	1	1	male	1	2	S
2	1	0	female	1	2	S
3	1	0	male	1	2	S
4	1	0	female	1	2	S

Figure 4.4: View of the Titanic dataset.

Two of the variables are categorical: gender and embarked. The rest of the variables are discrete and contain only a few distinct values, so we will treat them as categorical and evaluate their association with survival using Pearson's chi-square test.

Let's split the data into a train and a test set:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop("survived", axis=1),
    data['survived'],
```

```
    test_size=0.3,
    random_state=0,
)
```

The first step to calculating the chi-square statistic is to obtain the contingency tables between the input variables and the target. Let's create a contingency table between the variable gender and the target:

```
c = pd.crosstab(y_train, X_train['sex'])
```

If we execute `c`, we will see see the observed frequencies for the variables gender and survival:

sex	female	male
survived		
0	89	483
1	230	112

Figure 4.5: Contingency table of survival vs gender.

With the contingency table, we can now calculate the expected frequencies, degrees of freedom, chi-square, and the probability of the variables not being associated:

```
chi2_contingency(c)
```

The chi-square statistic, the probability of no association, the degrees of freedom, and the expected frequencies, are all included in the output of `chi2_contingency`:

```
(249.44419858265127,
 3.432495124524887e-56,
 1,
 array([[199.63676149, 372.36323851],
        [119.36323851, 222.63676149]]))
```

The probability of no association between gender and survival is `3.432495124524887e-56`, which is almost 0. That means that there is a statistically significant association between gender and survival, and by looking at the frequencies in *Figure 4.5*, women were more likely to survive.

We've analyzed the association between gender and survival. Let's now obtain the probability of association between every variable and the target:

```
# We'll store the chi2 values here.
chi_ls = []

for feature in X_train.columns:

    # Create contingency table.
    c = pd.crosstab(y_train, X_train[feature])

    # Chi-square test.
    p_value = chi2_contingency(c)[1]
    chi_ls.append(p_value)
```

Let's transform the list with the probabilities into a pandas series, add the variable names in the index, sort the probability values in ascending order, and make a bar plot:

```
pd.Series(chi_ls, index=X_train.columns).sort_values(
    ascending=True).plot.bar(rot=45)
plt.ylabel("p value")
plt.title("Feature importance based on chi-square test")
```

In the following bar plot, we see the probability of *no association* between each feature and the target. The smaller the probability, the more statistically significant the association between the variable and the target is. In other words, the lower the probability, the more important the variable is:

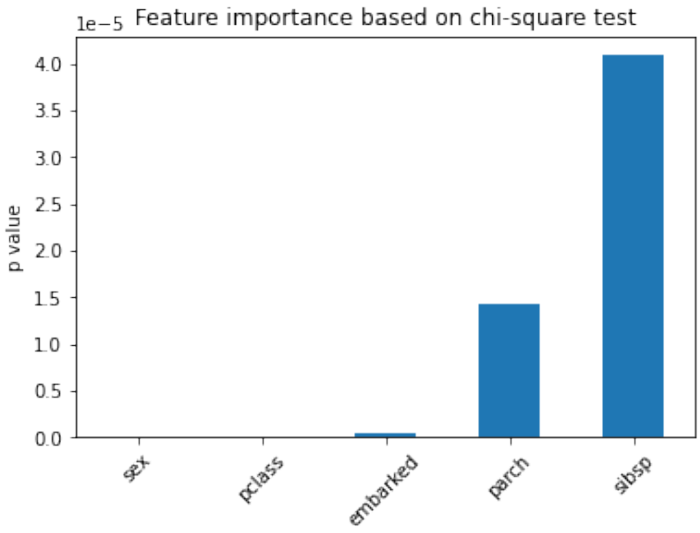


Figure 4.6: Feature ranking based on the chi-square test.

We ranked the features. The next question is: how do we select the top ranking features? How do we find a threshold or cut-off?

We can use different criteria to select features based on probabilities. We can, for example, select all features with a p-value greater than 0.05. If this is your preferred approach, note, however, that with bigger datasets, small differences in the category frequencies will be picked up as significant. In other words, in large datasets, selecting features based on the p-value may increase the error type I. An alternative selection criteria is to select the top k ranking features or the features in the top k percentile.

Let's capture the names of the top 3 ranking features:

```
selected = pd.Series(chi_ls, index=X_train.columns).sort_values(
    ascending=True)[0:3].index
```

If we execute `selected`, we see the names of the most important variables according to Pearson's chi-square:

```
Index(['sex', 'pclass', 'embarked'], dtype='object')
```

Let's now reduce the data to the selected features:

```
X_train = X_train[selected]
X_test = X_test[selected]
```

Below we see the selected data:

	sex	pclass	embarked
868	male	3	S
429	male	2	S
480	male	2	C
1308	male	3	S
455	male	2	S

Figure 4.7: Dataframe with the selected variables.

That's it. We have now selected features from a pool of categorical variables utilizing Pearson's chi-square test.

> ⚠ Warning
>
> Note that scikit-learn's `chi2` function is not designed to assess categorical variables with respect to a categorical target as described in this chapter. Don't use scikit-learn to assess categorical variables with chi^2.

4.2 ANOVA

When we have a binary or multi-class target and continuous variables, we can use one-way ANOVA to assess the contribution of the variables towards the outcome. Here, we divide the continuous variables into two or more groups based on the target classes, and then with ANOVA, we evaluate if the means of the variable in each of these groups are the same. Intuitively, we would expect the group means to vary with the class if the feature is predictive.

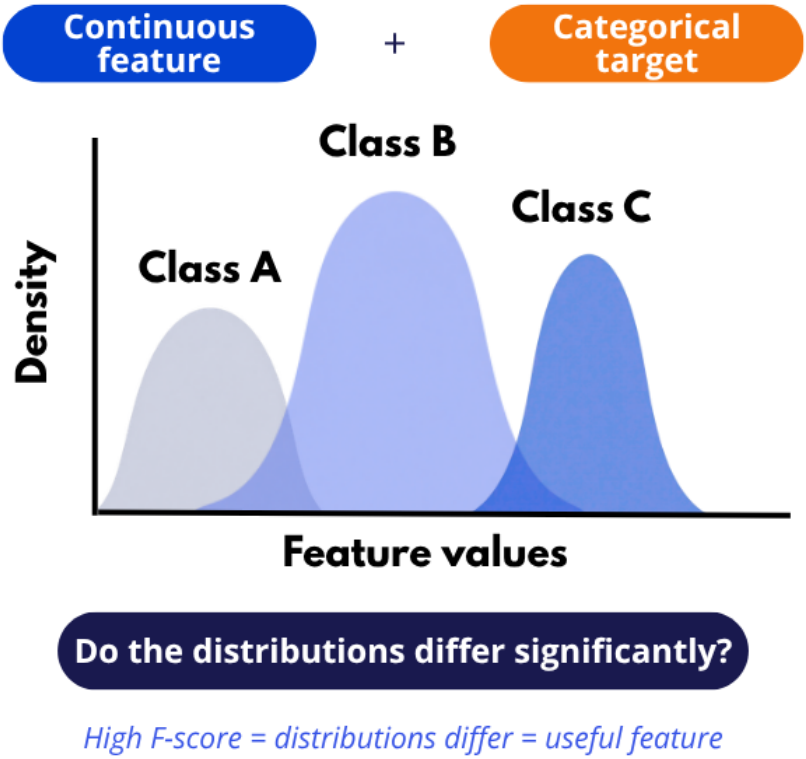


Figure 4.8: ANOVA is suitable to select among continuous variables when the target is categorical.

Let's say we want to predict the plant species based on the length of their leaves. We take 5 plants from each species, measure their leaves length (in cm), and obtain the following data:

Table 4.4: Plant leaf length.

Species A	Species B	Species C
28	21.8	22.2
25.7	24	21.4
28.2	26.7	22.1
32.3	25.6	28.2
27.5	23.8	19.5

If leaf length was a good indicator of species, intuitively, we would expect the mean of the leaf length to differ across the species. If we calculate the mean length per group, we obtain the

following:

Table 4.5: Mean leaf length per species.

Species A	Species B	Species C
28.34	**24.38**	**22.68**

Can we conclude that the length is different across groups? It is hard to say. It depends on the variability within each group. If the variability is large, then most likely these values are not different. But if the variability is small, then these values could be significantly different.

ANOVA stands for Analysis of Variance. One-way ANOVA is a statistical test used to analyze the differences among group means considering the variability. The null hypothesis for the ANOVA is that the means of the different groups are equal.

Sum of squares

ANOVA decomposes the *total variability* observed in the data into "explained" and "unexplained" variability. The *total variance* (we'll refer to variance and variability interchangeably) in the data is calculated by summing the square difference between each observation and the mean leaf length in the dataset. This is called the **Total sum of squares (SST)**:

$$SS_T = \sum (x_{ij} - \overline{x}_{grand})^2$$

In our data, $\overline{x}_{grand}$ is **25.13**. The total sum of squares is then:

$$\begin{aligned} SS_T = &(28 - 25.13)^2 + (25.7 - 25.13)^2 + (28.2 - 25.13)^2 + (32.3 - 25.13)^2 + (27.5 - 25.13)^2 + \\ &(21.8 - 25.13)^2 + (24 - 25.13)^2 + (26.7 - 25.13)^2 + (25.6 - 25.13)^2 + (23.8 - 25.13)^2 + \\ &(22.2 - 25.13)^2 + (21.4 - 25.13)^2 + (22.1 - 25.13)^2 + (28.2 - 25.13)^2 + (19.5 - 25.13)^2 \end{aligned}$$

That is, $SS_T = 164.63$.

Some of the total variability could be explained by the fact that different observations come from different groups. The variability among the group means is called the **Model sum of squares (SSM)** and is calculated as:

$$SS_M = \sum n_k(\overline{x_j} - \overline{x}_{grand})^2$$

where $\overline{x_j}$ is the mean within the group, n_k is the number of observations in each group, and $\overline{x}_{grand}$ is the mean of all observations combined.

For our data:

$$SS_M = 5(28.34 - 25.13)^2 + 5(24.38 - 25.13)^2 + 5(22.68 - 25.13)^2$$

That is:

$$SS_M = 84.34$$

The total variability in the data is **164.63**, our model, that is the group means, can explain only **84.34** of the variability. The **Residual sum of squares (SST)** captures the "unexplained" variability, that is, the differences between individuals in each group, like for example plant age or amount of sun received, among others.

The Residual sum of squares is calculated as:

$$SS_R = \sum (x_{ij} - \overline{x}_j)^2$$

For our data:

$$\begin{aligned} SS_R = &(28 - 28.34)^2 + (25.7 - 28.34)^2 + (28.2 - 28.34)^2 + (32.3 - 28.34)^2 + (27.5 - 28.34)^2 + \\ &(21.8 - 24.38)^2 + (24 - 24.38)^2 + (26.7 - 24.38)^2 + (25.6 - 24.38)^2 + (23.8 - 24.38)^2 + \\ &(22.2 - 22.68)^2 + (21.4 - 22.68)^2 + (22.1 - 22.68)^2 + (28.2 - 22.68)^2 + (19.5 - 22.68)^2 \end{aligned}$$

That is:

$$SS_R = 80.29$$

Degrees of freedom

The degrees of freedom for the different sums of squares vary. For the total sum of squares, it is the number of observations minus 1.

$$dof_{SS_T} = n_{observations} - 1 = 15 - 1 = 14$$

For the model sum of squares, the degrees of freedom are the number of groups minus 1:

$$dof_{SS_M} = n_{groups} - 1 = 3 - 1 = 2$$

For the residual sum of squares, the degrees of freedom are the number of observations minus the number of groups:

$$dof_{SS_R} = n_{observations} - n_{groups} = 15 - 3 = 12$$

Mean squares

The total sums of squares are influenced by the number of terms being summed. We can obtain the average of the sums of squares by dividing the sum of squares by their degrees of freedom. Thus, we obtain the mean squares for the model (MS_M) and mean squares for the residuals (MS_R) as follows:

$$MS_M = \frac{SS_M}{dof_{SS_M}} = \frac{84.34}{2} = 42.17$$

$$MS_R = \frac{SS_R}{dof_{SS_R}} = \frac{80.29}{12} = 6.69$$

MS_M is the mean variation explained by the group means, and MS_R is the mean variation that is left unexplained, or in other words, the mean random variation.

F-ratio

The F-ratio captures the variability explained by the model, that is, the variability of the group means versus the variability that cannot be explained. F is calculated as follows:

$$F = \frac{MS_M}{MS_R}$$

For our example:

$$F = \frac{42.17}{6.69} = 6.30$$

If the model can't explain any variability, we expect the MS_M to be smaller than the MS_R. The larger the MS_R, the more of the total variance that can be explained by the group means.

The way to interpret the value of 6.30 is that there is indeed a difference between the groups, because they explain a bigger part of the variability than the random variability per se. Otherwise, the F-ratio would be close to 1.

The F-ratio follows the F-distribution under the null hypothesis, a well-characterized distribution that depends on the degrees of freedom of the numerator and denominator of the F-ratio. The F-distribution represents the probability of obtaining the group means from table 4.5 by chance.

Once we know the F-ratio, we can determine the probability of obtaining the observed F under the null hypothesis, that is, by chance. For our data, that is $p - value = 0.0135$.

The p-value tells us that there is an approximate 1.35 in 100 chance that if we obtain the data from table 1, there is no difference between the group means. In other words, the leaf length is a good indicator of plant species.

Assumptions

One-way ANOVA makes some assumptions about the data:

- The variance in each group is similar (homogeneity of variance).
- The observations are independent.
- The values within each group are normally distributed.

When the assumptions are not met, the results of the test become less reliable. There's been a lot of research examining the impact of the violations to these assumptions. In short, ANOVA tends to be robust when the sample sizes are equal, meaning there are the same number of observations in each group. However, different sample sizes can affect the reliability of the test. For a full discussion, see Chapter 10 of *Field, et. al., Discovering statistics with R*, and references therein.

Considerations

The effect of the sample size means that ANOVA will be less reliable when the target is imbalanced. Additionally, ANOVA may not perform equally well with normally distributed versus skewed variables, making the probability values obtained for these variables not directly comparable.

When the assumptions of ANOVA are not met, we can transform the data with "variance stabilizing transformations" to improve value spread, remove skewedness and stabilize the variance. For more details about variance stabilizing transformations check out the article at https://t.ly/U_44m.

Python implementation

We will select features based on one-way ANOVA using the breast cancer dataset from Scikit-learn. The data contains characteristics of tumors obtained from images, and the aim is to predict if the tumor is benign or malignant. The variables are continuous and the target is binary.

To select features using one-way ANOVA, we first calculate the p-value for each feature, which tests whether the mean values of the feature differ across the target classes. We then rank the features based on their p-values and select the top-ranked ones.

Let's import the libraries, functions, and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import (
    f_classif,
    SelectFpr,
    SelectKBest,
)
from sklearn.model_selection import train_test_split
```

Let's load the breast cancer dataset and separate it into train and test sets:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's examine the distribution of the variables through histograms:

```
X_train.hist(bins=30, figsize=(20,20))
plt.show()
```

From the following histograms we see that most variables are not normally distributed. Thus, the data violates one of the assumptions of ANOVA. This means that the p-values obtained for skewed variables may be less reliable.

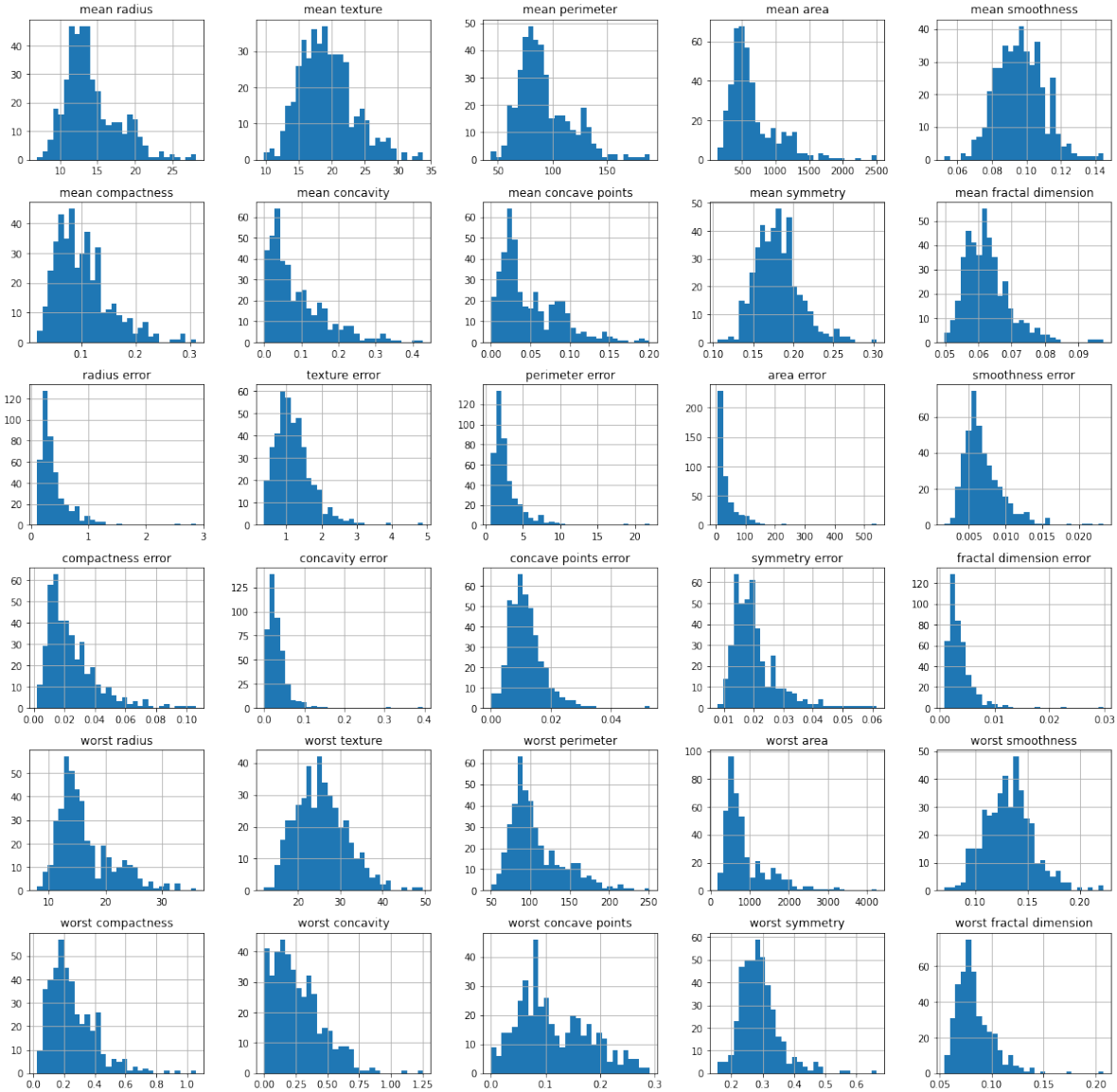


Figure 4.9: Histograms with the distributions of the predictive variables.

For the sake of the demonstration, we will ignore the fact that the assumptions are not met and continue with the implementation.

> **Tip**
>
> To transform skewed variables into normally distributed features we can apply variance stabilizing transformations like the logarithm, square root, power functions or the Box-Cox or Yeo-Johnson transformations. Check out Feature-engine's `transformation` module for more details.

Let's examine all features with one-way ANOVA:

```
univariate = f_classif(X_train, y_train)
```

If we execute `univariate` we will obtain 2 arrays, the first one with the F-ratio values and the second one with the p-values per feature:

```
(array([4.56888468e+02, 8.07899168e+01, 4.90890258e+02, 3.94647061e+02,
        6.91090732e+01, 2.51091933e+02, 3.77169033e+02, 6.40392388e+02,
        5.64534892e+01, 2.32061889e-01, 1.89198831e+02, 4.60132652e-01,
        1.77794821e+02, 1.61603286e+02, 3.44368683e+00, 3.19209297e+01,
        1.52804441e+01, 6.90719571e+01, 3.73129536e-01, 1.22007037e+00,
        6.11484676e+02, 1.04059005e+02, 6.34234486e+02, 4.60094540e+02,
        9.71806726e+01, 2.42990980e+02, 3.24298240e+02, 7.30475928e+02,
        9.36811856e+01, 4.94480861e+01]),
 array([2.55823842e-69, 8.37108067e-18, 8.19690935e-73, 1.48052645e-62,
        1.27734112e-15, 9.49913611e-45, 1.45377554e-60, 8.98762428e-87,
        3.44436651e-13, 6.30247708e-01, 7.43308506e-36, 4.97932018e-01,
        4.06310902e-34, 1.36320509e-31, 6.41875480e-02, 2.94961572e-08,
        1.07859166e-04, 1.29817853e-15, 5.41630631e-01, 2.69973265e-01,
        3.11012794e-84, 5.36444219e-22, 3.08095116e-86, 1.18230031e-69,
        8.87987480e-21, 1.24060859e-43, 2.92390622e-54, 2.90246801e-94,
        3.75876060e-20, 8.20428355e-12]))
```

Let's capture the p-values in a pandas series, add the variable names in the index, sort the features based on their p-values and make a bar plot:

```
univariate = pd.Series(univariate[1])
univariate.index = X_train.columns
univariate.sort_values(ascending=True).plot.bar(figsize=(20, 6), rot=45)
plt.ylabel("p-values")
plt.title("Anova")
```

In the following plot, we see that most features have p-values smaller than 0.05. For those features, we conclude that the mean value of the features between benign and malignant tumors is significantly different.

There are a few features whose p-value is bigger than 0.05, which means that their mean value is similar for benign and malignant tumors.

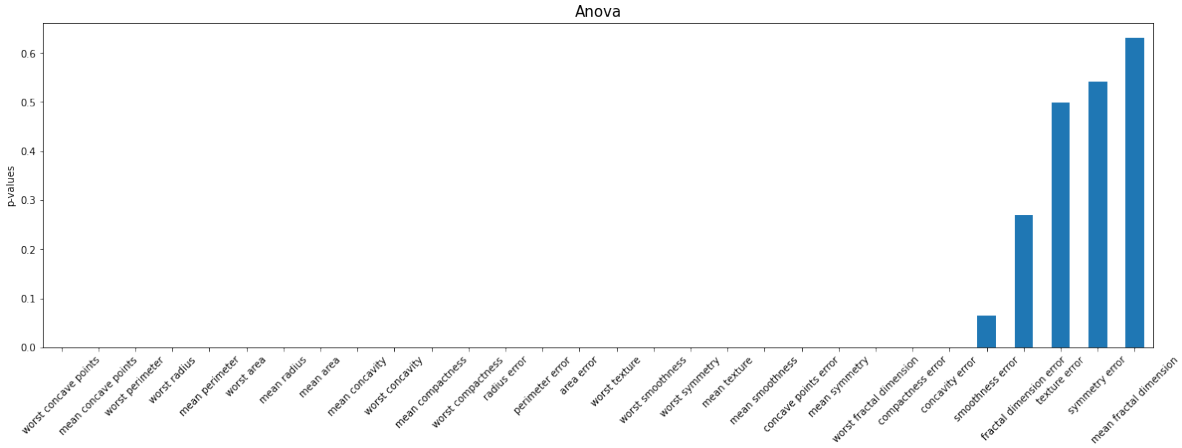


Figure 4.10: p-values obtained with ANOVA for each feature.

Let's compare the distribution of a feature whose p-value is smaller than 0.05 (e.g., `worst smoothness`) between target classes:

```
sample1 = X_train.loc[y_train==0, "worst smoothness"]
sample2 =  X_train.loc[y_train==1, "worst smoothness"]

plt.hist(sample1, bins=30, alpha=0.3, label="benign")
plt.hist(sample2, bins=30, alpha=0.3, label="malign")
plt.legend(loc="upper right")
plt.xlabel("Smoothness")
plt.ylabel("Number of Observations")
plt.show()
```

In the following plot, we see that the feature "worst smoothness" has different distributions associated with the target classes, showing smaller values for malignant tumors. In other words, malignant tumors are less smooth:

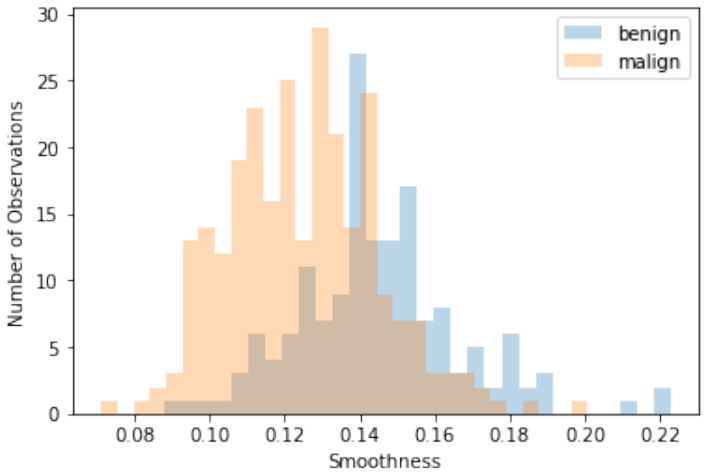


Figure 4.11: Distribution of “worst smoothness” in benign and malignant tumors.

Let’s compare the distribution of a feature whose p-value is greater than 0.05 (e.g., `symmetry error`) between target classes:

```
sample1 = X_train.loc[y_train==0, "symmetry error"]
sample2 =  X_train.loc[y_train==1, "symmetry error"]

plt.hist(sample1, bins=30, alpha=0.3, label="benign")
plt.hist(sample2, bins=30, alpha=0.3, label="malign")
plt.legend(loc="upper right")
plt.xlabel("Symmetry error")
plt.ylabel("Number of Observations")
plt.show()
```

In the following plot, we see that the values of the feature “symmetry error” are very similar among benign and malignant tumors. Hence, with this feature, we can’t discriminate if a tumor is benign or malignant:

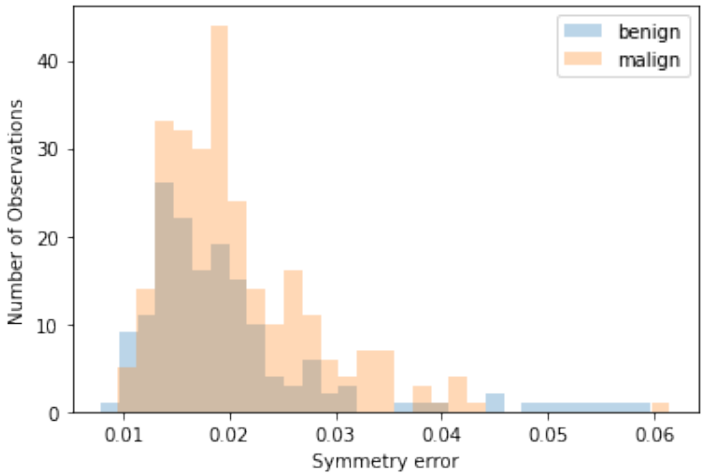


Figure 4.12: Distribution of “symmetry error” in benign and malignant tumors.

We will use one-way ANOVA to rank features based on p-values and then select features whose p-value is greater than 0.05. To select features based on a probability threshold, we wrap scikit-learn’s `f_classif` with the `SelectFpr` transformer, highlighting the probability value that should be used as cut-off:

```
sel = SelectFpr(f_classif, alpha=0.05).set_output(transform="pandas")
```

> **Note**
>
> Note how we use `set_output()` to enforce the transformer `SelectFpr` to return pandas dataframes.

With `fit()` we trigger the calculation of the p-values and the selection of the features that are above the threshold:

```
sel.fit(X_train, y_train)
```

If we execute `X_train.columns[sel.get_support()]` we’ll see the selected features:

```
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'radius error',
       'perimeter error', 'area error', 'compactness error',
```

```
       'concavity error', 'concave points error', 'worst radius',
       'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry',
       'worst fractal dimension'],
      dtype='object')
```

Let's reduce the data to the selected variables:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

The final dataset contains 25 of the original 30 variables. You can corroborate that by executing `X_train_t.shape`.

Instead of selecting features based on a p-value cut-off, we can select the top *x* ranking features. To do this, we wrap scikit-learn's `f_classif` with the `SelectKBest` transformer, highlighting the number of features to retain. Here, we'll select the top 10 features:

```
sel = SelectKBest(f_classif, k=10).set_output(transform="pandas")
```

With `fit()` we trigger the calculation of the p-values and the identification of the 10 best ranked features:

```
sel.fit(X_train, y_train)
```

If we execute `X_train.columns[sel.get_support()]` we obtain the names of the selected features:

```
Index(['mean radius', 'mean perimeter', 'mean area', 'mean concavity',
       'mean concave points', 'worst radius', 'worst perimeter', 'worst area',
       'worst concavity', 'worst concave points'],
      dtype='object')
```

Now we can reduce the data to the selected variables:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

That's it! We have now selected variables based on one-way ANOVA.

4.3 Correlation

We can assess the importance of continuous variables to predict a continuous target by determining the degree of association between the feature and the target variable through correlation.

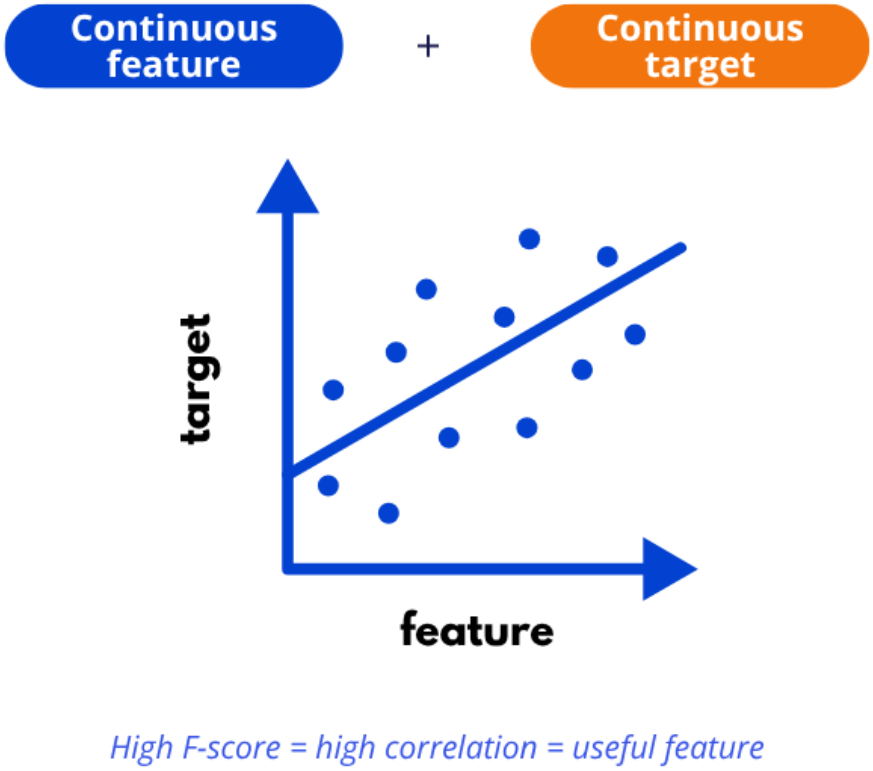


Figure 4.13: Correlation is suitable to select among continuous variables when the target is also continuous.

Two variables are related if changes in one variable are met with similar changes in the other variable. Thus, if one of the variables deviates from its mean, we expect the other variable to also deviate from its mean value. This is captured in the following equation:

$$R = \frac{\sum(x - \overline{x})(y - \overline{y})}{(N-1)s_x s_y}$$

where we measure if deviations in x from its mean are followed by deviations in y from its mean.

If you went over *Chapter 3*, you'll probably recognize the former equation as Pearson's correlation coefficient. The numerator of the equation is nothing other than the covariance of the variables. The denominator standardizes the distances from the mean by the product of the standard deviations. Pearson's correlation coefficient, then, measures the degree of association between 2 variables and can vary between -1 and 1.

When R is positive, there is a positive association between the variables; that is, the greater the values of x, the greater the values of y. When R is negative, there is a negative association between the variables; that is, the greater the values of x, the smaller the values of y. When R is 0, the variables are not associated.

We can determine the strength of the association between the continuous variable and the target by the magnitude of the (absolute) correlation coefficient. We could then rank the features based on the absolute correlation coefficient and select the top ranking features.

We can also obtain the probability of 2 variables *not* being associated given the value of R using the *t*-statistic with N-2 degrees of freedom. This probability can be directly obtained from R through the following equation:

$$t = r \times \sqrt{\frac{N-2}{1-r^2}}$$

The *t*-statistic follows the well characterized *t*-student's distribution and tells us the probability of getting R if the variables are not associated. As we see from the formula, with larger values of R, we obtain larger values of *t*, and thus, smaller probabilities. In other words, greater correlation coefficients are less likely to result from a random association between variables.

If we choose to rank the features based on probabilities, then, smaller probability values indicate a stronger relationship between the variable and the target.

Assumptions

Pearson's correlation coefficient requires the variables to be continuous for an accurate representation of the linear association between the variables. If we want to derive a measure of probability from the correlation coefficients using the *t*-distribution, the *t*-test has an additional assumption, which is that the variables are normally distributed.

Considerations

Pearson's correlation coefficient measures the linear association between variables. We saw in *Chapter 3* that there are other correlation coefficients that measure non-linear associations between variables. We could use these coefficients to assess the non-linear association between continuous predictor variables and a continuous target as well, although without obtaining a measure of probability, as these tests are non-parametric. If we use Spearman's or Kendall's correlation coefficients, we would rank the features based on the magnitude of the coefficient, and then select the top ranking features.

Python implementation

We will select features based on Pearson's correlation utilizing Scikit-learn's `f_regression` function. This function calculates the Pearson correlation coefficient for each feature and then derives the *t*-statistic and the probability of obtaining said coefficient if the variables are not associated. Thus, when using Scikit-learn, we need to ensure that the data meets both the assumptions of Pearson's correlation coefficient and Student's *t*-test.

Let's begin by importing the libraries, classes, and functions:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.feature_selection import (
    f_regression,
    SelectPercentile,
)
from sklearn.model_selection import train_test_split
```

Let's load the California house price dataset and separate it into a train and a test set:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's make histograms of the predictor variables:

```
X_train.hist(bins=30, figsize=(12,12))
plt.show()
```

We see in the histograms that the variables are not normally distributed, so the data violates the assumptions of the *t*-test:

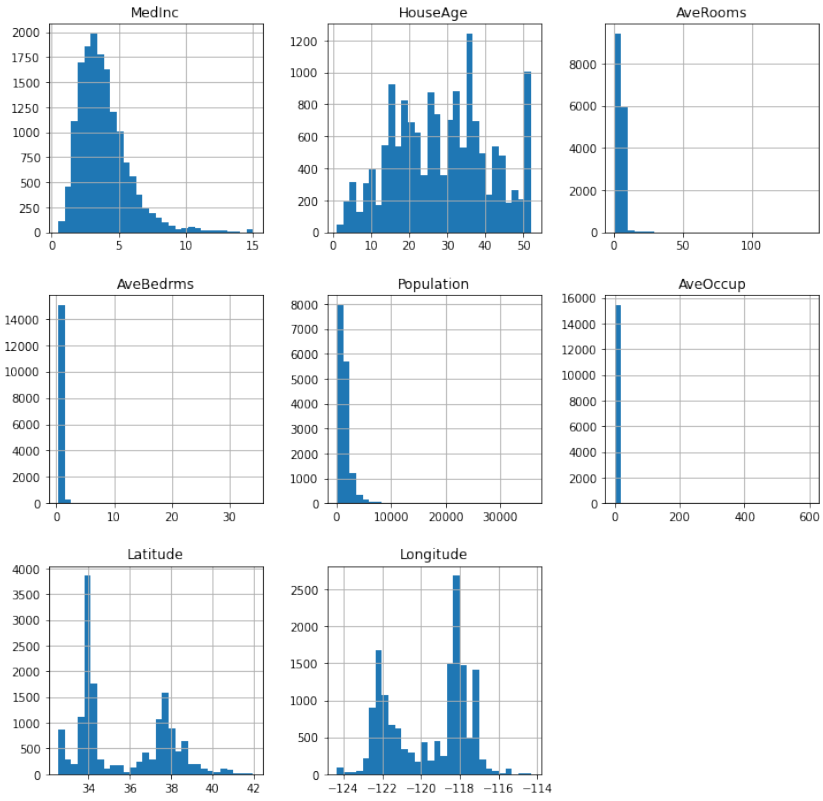


Figure 4.14: Histogram with the distribution of the predictive variables.

We can transform the variables using variable stabilizing transformations like the logarithm or the Box-Cox transformation among others. I will leave that with you and carry on with the demo instead.

For the sake of the demonstration, let's continue with the calculation of the probabilities based on Pearson's correlation coefficient. Let's obtain the *t*-statistic and p-value that measures the association between each feature and the target variable:

```
univariate = f_regression(X_train, y_train)
```

If we execute `univariate` we will see the following 2 arrays; the first one with the *t*-statistics and the second one with the p-values:

```
(array([1.43228768e+04, 1.74602749e+02, 3.73936211e+02, 2.74469506e+01,
        1.01599261e+01, 1.59941176e+01, 3.30604925e+02, 3.17583459e+01]),
 array([0.00000000e+00, 1.19851133e-39, 2.43044189e-82, 1.63580388e-07,
        1.43811689e-03, 6.38347779e-05, 4.08861089e-73, 1.77640721e-08]))
```

Let's capture the p-values in a pandas series, add the feature names in the index, sort them in increasing order, and make a bar plot:

```
plt.rc("axes", titlesize=15)
univariate = pd.Series(univariate[1])
univariate.index = X_train.columns
univariate.sort_values(ascending=True).plot.bar(figsize=(10, 5), rot=45)
plt.ylabel("p-values")
plt.title("Correlation")
plt.show()
```

In the following plot, we see that all variables are significantly associated with the target since all probability values are smaller than 0.05:

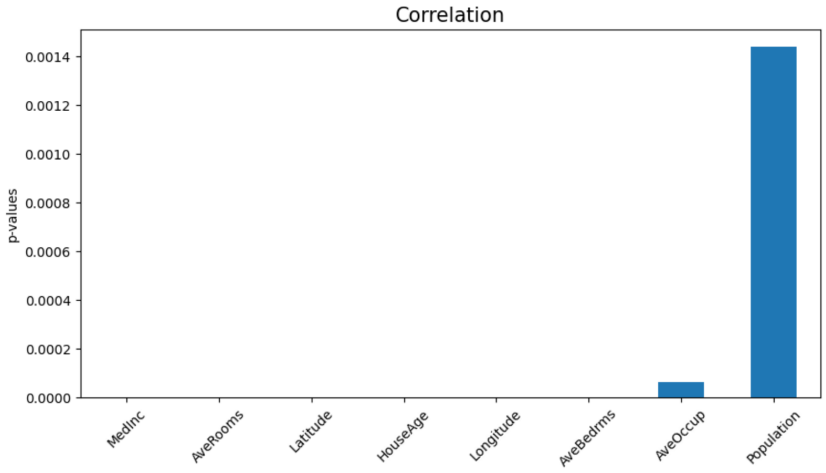


Figure 4.15: Probability of no-association between features and target.

The smaller the p-value the more predictive the feature is.

> **Important**
>
> Remember that large datasets can push the probability values to arbitrarily low numbers, hence increasing the type I error. When using p-values to rank features, that is, in general not a problem, but if we are interested in the probability of the association per se, we need to consider the sample size.

Let's plot the association of the most important feature, i.e., `MedInc`, with the target:

```
plt.scatter(X_train["MedInc"], y_train, alpha=0.3, color="purple")
plt.xlabel("MedInc")
plt.ylabel("House price")
plt.show()
```

In the following scatter plot, we see a decent linear relationship between median income (`MedInc`) and house price:

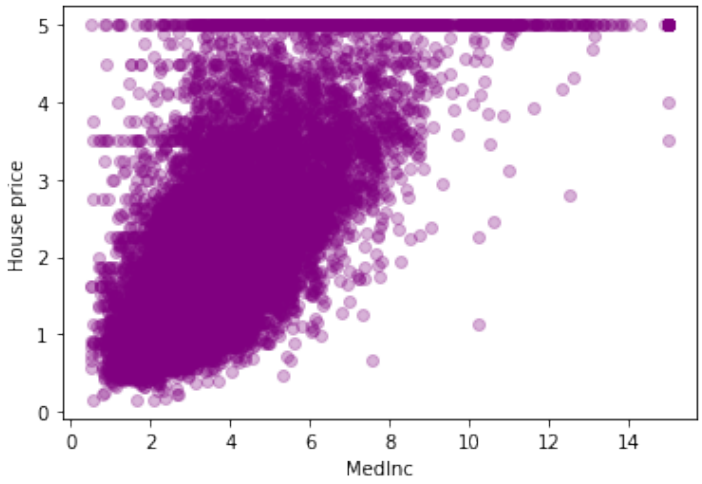


Figure 4.16: Scatter plot of linearly associated variables.

Now, let's plot the association between the least important feature, i.e., `Population`, and the target:

```
plt.scatter(X_train["Population"], y_train, alpha=0.3, color="purple")
plt.xlim(0, 10000)
plt.xlabel("Population")
plt.ylabel("House price")
plt.show()
```

The relationship between `Population` and house price is not linear as can be witnessed from the following plot:

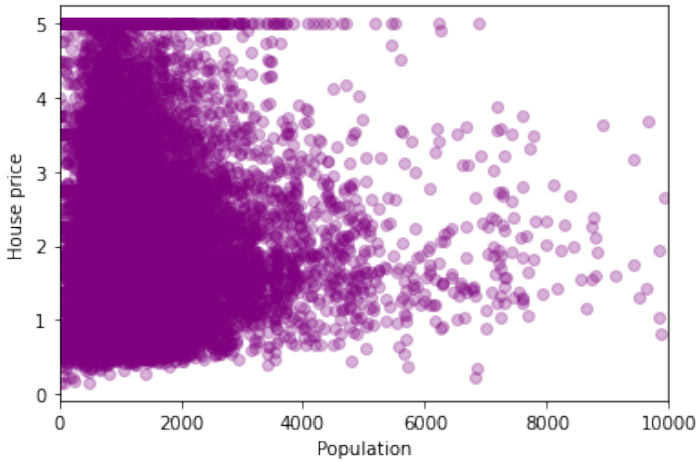


Figure 4.17: Scatter plot showing the poor linear association between a predictor and the target.

We will now use Scikit-learn's `f_regression` together with `SelectPercentile` to select the features ranked in the top 30th percentile:

```
sel = SelectPercentile(f_regression, percentile=30).set_output(
    transform="pandas")
```

> **Note**
>
> We enforce the output from `SelectPercentile` to be a pandas dataframe by setting the `transform` output to pandas.

With `fit()`, we trigger the calculation of the p-values and the selection of the features in the top 30th percentile:

```
sel.fit(X_train, y_train)
```

Let's print out the selected features:

```
sel.get_feature_names_out()
```

Below we see the features that rank in the top 30th percentile:

```
array(['MedInc', 'AveRooms', 'Latitude'], dtype=object)
```

Now, we reduce the dataset to the selected features:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

If we now execute `X_train_t.head()` we'll see the first five rows of the resulting data:

	MedInc	AveRooms	Latitude
0	7.3003	7.961600	38.46
1	5.9547	6.547325	32.95
2	3.2125	5.207547	34.68
3	6.9930	6.428571	33.51
4	2.5162	4.429348	38.62

Figure 4.18: Dataframe with the selected features.

And that's it! We have now selected variables based on their correlation with a continuous target.

4.4 Mutual information

Mutual information (MI) measures the mutual dependence between two random variables. It "quantifies" the amount of information we can know about one variable by observing the values of the other variable.

Since MI measures information, it has units, which can be bits, nats, or hartleys, as we will discuss in a coming paragraph.

MI offers a good statistic for measuring the association between variables and target for many reasons:

- MI detects any type of relationship, not just linear ones.

- MI has a straightforward interpretation: the amount of shared information between variables is measured in, for example, bits.
- MI is insensitive to the size of the datasets (contrary to the p-values derived from statistical tests like chi-square, ANOVA, or correlation, which can be pushed to arbitrarily low values in large datasets).
- MI can be used with discrete and continuous predictors and discrete and continuous targets, offering a suitable solution for most datasets.

MI is closely related to the concept of entropy. Hence, in the following sections, we will first introduce entropy, then explain how to calculate the entropy of a discrete variable. Next, we will illustrate how to determine MI between a pair of discrete variables. We will extend the definition of MI to continuous variables, and finally, explain how to estimate MI for continuous features. We will conclude this chapter with a Python implementation of feature selection based on MI.

In summary, in the following sections we will discuss:

- Entropy.
- Relative entropy.
- Mutual information between discrete variables.
- Mutual information between continuous variables.
- Feature selection based on MI with Python.

Entropy

The entropy of a variable is a measure of the information, or in other words, the "uncertainty," of the variable's possible values. It is defined as:

$$H(X) = -\sum p(x)log(p(x))$$

where H(X) is the variable's entropy and p(x) is the probability of X' values. If the base of the logarithm is 2, then the unit of the entropy is a bit. If the logarithm's base is *e*, then the unit is the nat, and if it is 10, the unit is the hartley.

To illustrate the calculation with an example, the entropy of a fair coin toss is 1 bit:

$$H(X) = -(p(x=0)log(p(x=0)) + p(x=1)log(p(x=1)))$$

$$H(X) = -(0.5 \times log(0.5) + 0.5 \times log(0.5))$$

$$H(X) = 0.5 + 0.5 = 1$$

Note that the logarithm in base 2 of 0.5 is -1.

Relative entropy

The relative entropy measures the distance between two distributions and it is also called **Kullback-Leibler distance**. It is given by:

$$D(p|q) = \sum p(x) log(\frac{p(x)}{q(y)})$$

where p(x) and q(y) are two probability distributions.

MI definition

Utilizing the relative entropy, we can now define MI. MI is the relative entropy between the joint distribution of two variables and the product of their marginal distributions. Thus, MI is defined as:

$$I(X,Y) = \sum_X \sum_Y p(x,y) log(\frac{p(x,y)}{p(x)p(y)})$$

where I(X,Y) is the MI between variables x and y, p(x,y) is the joint probability of the two variables, and their marginal probabilities are p(x) and p(y).

Note that MI can be equal or greater than 0. When $p(x,y) = p(x)p(y)$, MI is 0: The joint probability is equal to the product of the marginals when there is no association between the variables.

MI between discrete variables

To illustrate the calculation of MI with an example, let's say we have the following contingency table of survival in the Titanic based on gender:

Table 4.6: survival vs. gender.

	Female	Male	Total
Not survived	89	483	571
Survived	230	112	342

	Female	Male	Total
Total	319	595	914

With the table frequencies, we can create probability estimates by dividing the counts in each cell by the total number of passengers, which is 914:

Table 4.7: Probability distributions.

	Female	Male	Total
Not survived	0.0974	0.5285	0.6258
Survived	0.2516	0.1225	0.3742
Total	0.3490	0.6510	1

The MI between the the variables survival and gender is:

$$I(X,Y) = 0.0974 \times log\frac{0.0974}{0.6258 \times 0.3490} + 0.5285 \times log\frac{0.5285}{0.6258 \times 0.6510} +$$

$$0.2516 \times log\frac{0.2516}{0.3742 \times 0.3490} + 0.1225 \times log\frac{0.1225}{0.3742 \times 0.6510}$$

Thus, $I(X,Y) = 0.2015$, which is bigger than 0, indicating that by knowing the gender of the passenger, we know more about their probability of survival.

MI estimation for continuous variables

We can extend the definition of MI to continuous variables by changing the sum over the values of x and y by the integrals:

$$I(X,Y) = \int_X \int_Y p(x,y) log(\frac{p(x,y)}{p(x)p(y)})$$

With continuous variables, the challenge is how to estimate the probability densities for each variable value. When the variable was discrete, we created a contingency table, estimated the marginal and joint probabilities, and then used those to calculate the MI. With continuous variables, this is not possible for 2 reasons: first, the variables can take infinite values, and second, in any dataset, we will only have a few of those probable values. So, how can we calculate the MI?

Perhaps the most obvious approach is to discretize the continuous variables, often into intervals of equal frequency, and then proceed as if they were discrete variables. The issue here is how to define the number of intervals. It's been shown that an incorrect number of intervals results in poor estimates of the MI.

Alternatively, nearest-neighbor methods have been introduced to estimate the MI between continuous variables and a continuous target, or between continuous variables and a discrete target. These methods have been shown to provide much better estimates of MI for continuous data.

Nearest-neighbours method to estimate MI

We have a series of data points in our dataset that contain values for the continuous variables x and y, with a joint probability p(x,y) that we do not know but must estimate from the observed data. The nearest-neighbor method estimates the joint probability of these two continuous variables, as well as the joint probability of a continuous and a discrete variable.

The following figure (panel A) illustrates the joint distribution of the discrete variable x, which takes 3 values: red, green, or blue; and the continuous variable y. In this example, we see that the different values of x are associated with different values of y; for example, y is generally smaller when x is green or red than when x is blue. Therefore, there is a relation between x and y, implying that MI is some positive number.

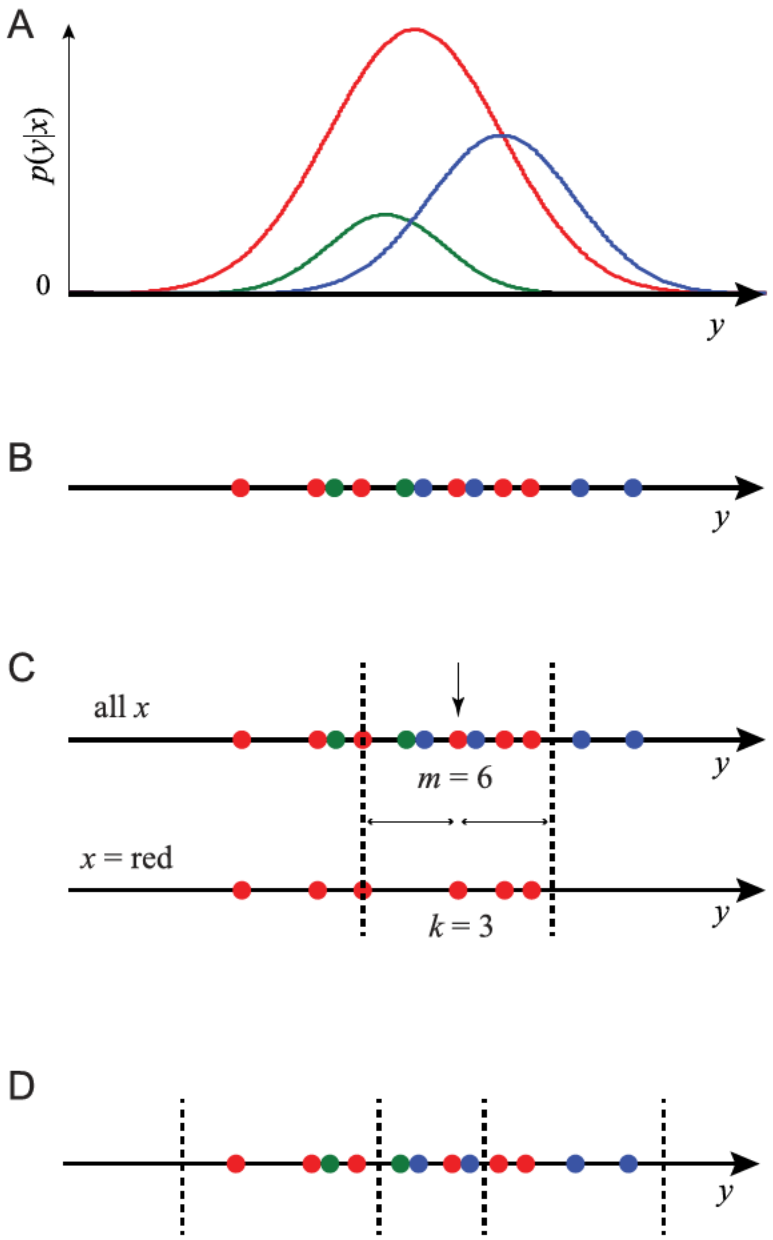


Figure 4.19: Nearest-neighbor approach to estimate the MI. Taken from Ross, 2014, PLoS ONE 9(2): e87357.

From the joint distribution (panel A), we sample some observations, which represent the available data (panel B). The challenge is to estimate the MI between x and y given those few observations.

The nearest-neighbour approach works as follows:

1. We take 1 observation and find the k closest neighbours that show the same value for x (N_{xi}). So if we take an observation that is red, like the example in panel C, we find its 3 closest red neighbours.
2. We calculate the distance *d* between the observation and its furthest neighbour (broken lines in panel C).
3. We count the total number of observations (m_i), red and otherwise, within d of the observation in question.

Based on N_{xi}, m_i, k (the number of neighbours) and N (the total number of observations), we calculate the MI for that particular observation as:

$$I_i = \psi(N) - \psi(N_{xi}) + \psi(k) - \psi(m_i)$$

where ψ is the digamma function. To estimate the MI from the dataset, we average I_i over all data points:

$$I(X,Y) = \psi(N) - \psi(N_x) + \psi(k) - \psi(m_i)$$

To evaluate the association between 2 continuous variables, i.e., when the target is continuous and the variables are continuous, MI is calculated as:

$$I(X,Y) = \psi(N) + \psi(k) - \psi(N_x) - \psi(N_y)$$

where N_x and N_y are the number of neighbours of the same value and different values found within the sphere generated by the distance determined in step 2.

The demonstration of how these equations were derived and how this method compares with the binning approach is beyond the scope of this book. For further details, I invite you to revise the references at the end of this section.

Feature selection using MI

Selecting features with MI is straightforward. First, we determine the MI between each feature and the target. Next, we rank the features based on the MI. Finally, we select the top ranking features.

The next question is, where do we put the cut-off to select the top ranking features? There are a few ways in which this can be done:

- We can select the top k features, where k is an arbitrary number of features.
- We can select the features in the top *n* percentile, where *n* is an arbitrary number.
- We can select the features above a certain MI value, arbitrarily decided.

Python implementation

We will select features using mutual information in Python. First, we will rank continuous features against a discrete target and select the top ranked features. Next, we will rank continuous features against a continuous target and select the features in the top k percentile. Finally, we will rank continuous and discrete features to become familiar with the correct implementation of mutual information in Scikit-learn.

Continuous features vs categorical target

We will use the breast cancer dataset from Scikit-learn to predict if a tumor is benign or malignant based on certain continuous tumor attributes. We will rank features using MI and select the top k ranking features.

Let's import the required libraries, functions, data and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

# To obtain the mutual information values.
from sklearn.feature_selection import mutual_info_classif

# To select the features.
from sklearn.feature_selection import SelectKBest
```

Let's load the breast cancer dataset and separate it into a train and a test set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

> **Tip**
>
> You can execute `X_train.hist(bins=50, figsize=(20, 20))` to corroborate that the variables are continuous.

Let's calculate the MI between each feature and the target. Because the target is binary, we use `mutual_info_classif`:

```
mi = mutual_info_classif(X_train, y_train, random_state=42)
```

If we execute `mi` we obtain the array with the MI values for each feature:

```
array([0.33245563, 0.07886352, 0.3955884 , 0.35041167, 0.09681211,
       0.2543844 , 0.36952764, 0.4481318 , 0.08601205, 0.02245148,
       0.24540984, 0.        , 0.2531849 , 0.33463724, 0.01364823,
       0.06080088, 0.12302022, 0.144789  , 0.        , 0.04887738,
       0.44247429, 0.12516385, 0.45588724, 0.45111359, 0.08631011,
       0.23481521, 0.30971506, 0.44215969, 0.0716687 , 0.02077104])
```

Next, let's capture the above array in a pandas series, add the variable names in the index, sort the features based on their mutual information, and make a bar plot:

```
mi = pd.Series(mi)
mi.index = X_train.columns
mi.sort_values(ascending=False).plot.bar(figsize=(20, 6))
plt.ylabel('Mutual Information')
plt.title("Mutual information between predictors and target")
```

In the following output, we see features ranked based on their MI with the target:

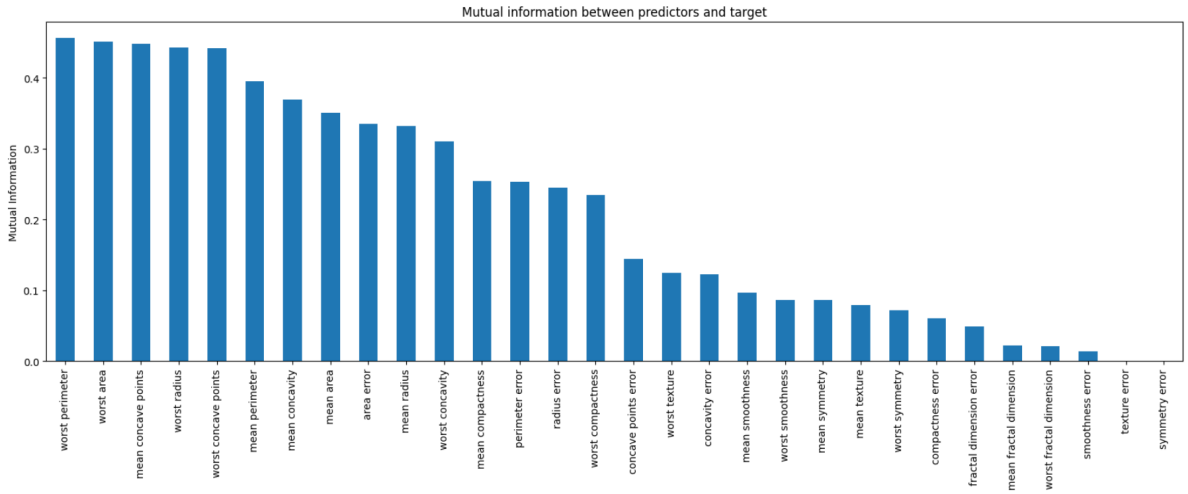


Figure 4.20: MI between the features and the target.

We've got a couple of features whose MI is 0. We could discard those from the dataset. Alternatively, we can select the top 5 ranking features:

```
sel = SelectKBest(mutual_info_classif, k=5).fit(X_train, y_train)
```

If we execute `X_train.columns[sel.get_support()]` we will see an array with the top 5 ranking variables:

```
Index(['mean concave points', 'worst radius', 'worst perimeter', 'worst area',
       'worst concave points'],
      dtype='object')
```

Let's remove the low ranking features:

```
X_train = sel.transform(X_train)
X_test = sel.transform(X_test)
```

Scikit-learn returns numpy arrays. Let's convert them to dataframes:

```
X_train = pd.DataFrame(X_train, columns=sel.get_feature_names_out())
X_test = pd.DataFrame(X_test, columns=sel.get_feature_names_out())
```

> **Note**
>
> Remember that you can enforce scikit-learn transformers to return dataframes by setting their output to pandas as follows: `SelectKBest(mutual_info_classif, k=5).set_output(transform="pandas")`.

If we execute `X_train.head()` we will see the top 5 rows of the resulting dataframe:

	mean concave points	worst radius	worst perimeter	worst area	worst concave points
0	0.022800	13.06	84.35	517.8	0.09140
1	0.007583	11.98	76.91	436.1	0.02022
2	0.097910	23.69	155.00	1731.0	0.16280
3	0.012380	15.50	98.91	739.1	0.05185
4	0.023070	19.18	127.30	1084.0	0.08737

Figure 4.21: Dataframe with the selected features.

Continuous features vs continuous target

We will use the California house price dataset from Scikit-learn to predict house prices based on house and area attributes. We will rank features using MI and select the features in the top k percentile.

Let's import the required libraries, functions, data and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

# To obtain the mutual information values.
from sklearn.feature_selection import mutual_info_regression

# To select the features.
from sklearn.feature_selection import SelectPercentile
```

Let's load the dataset and separate it into train and test sets:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

> **Tip**
>
> You can execute `X_train.hist(bins=50, figsize=(20, 20))` to corroborate that the variables are continuous.

Let's calculate the MI between each feature and the target. Because the target is continuous, we use `mutual_info_regression`:

```
mi = mutual_info_regression(X_train, y_train, random_state=42)
```

Next, let's capture the above array in a pandas series, add the variable names in the index, sort the features based on their mutual information, and make a bar plot:

```
mi = pd.Series(mi)
mi.index = X_train.columns
mi.sort_values(ascending=False).plot.bar(figsize=(10, 5))
plt.ylabel('Mutual Information')
plt.title("Mutual information between predictors and target")
```

In the following output, we see features ranked based on their MI with the target:

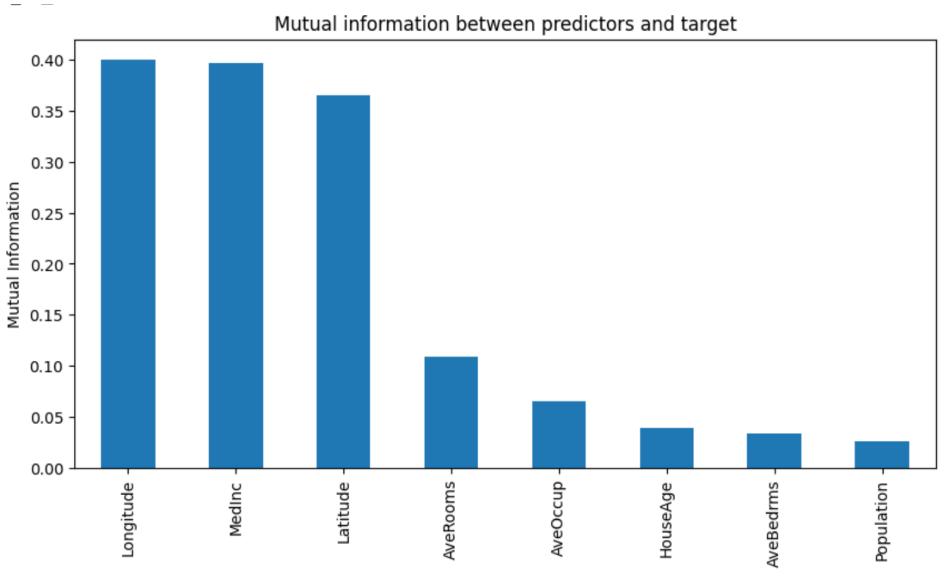


Figure 4.22: MI between features and target.

All features show positive MI, so we could keep them all. For the demo, we will select the features in the top 30th percentile:

```
sel = SelectPercentile(
    mutual_info_regression,
    percentile=30).set_output(transform="pandas")

sel.fit(X_train, y_train)
```

If we execute `X_train.columns[sel.get_support()]` we obtain an array with the features in the top 30th percentile:

```
Index(['MedInc', 'Latitude', 'Longitude'], dtype='object')
```

Let's now remove the low ranking features:

```
X_train = sel.transform(X_train)
X_test = sel.transform(X_test)
```

If we execute `X_train.head()` we obtain the first 5 rows of the dataset with the selected features:

	MedInc	Latitude	Longitude
0	7.3003	38.46	-122.68
1	5.9547	32.95	-117.24
2	3.2125	34.68	-118.27
3	6.9930	33.51	-117.18
4	2.5162	38.62	-121.41

Figure 4.23: Dataframe with the selected features.

Dataset with continuous and discrete variables

We mentioned previously that MI is calculated differently depending on whether the variables are continuous or discrete. Thus, it is important to inform the functions `mutual_info_classif` and `mutual_info_regression` if the predictor variables are continuous or discrete. We do so by passing a mask with `True` for discrete variables or `False` otherwise.

We will work with the Titanic dataset, which has continuous and discrete variables. Let's begin by making the necessary imports:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.feature_selection import mutual_info_classif

from feature_engine.encoding import RareLabelEncoder, OrdinalEncoder
```

Let's load and prepare the Titanic dataset:

```
variables = ['pclass', 'survived', 'sex', 'age',
             'sibsp', 'parch', 'fare', 'cabin', 'embarked']

data = pd.read_csv(
    'https://www.openml.org/data/get_csv/16826755/phpMYEkMl',
    usecols=variables,
```

```
    na_values='?',
    dtype={'fare': float, 'age': float},
    )

data.dropna(subset=['embarked', 'fare'], inplace=True)
data['age'] = data['age'].fillna(data['age'].mean())

def get_first_cabin(row):
    try:
        return row.split()[0]
    except:
        return 'N'


data['cabin'] = data['cabin'].apply(get_first_cabin).str[0]
encoder = RareLabelEncoder(variables='cabin', n_categories=2)
data = encoder.fit_transform(data)

# Convert categorical variables to numbers.
encoder = OrdinalEncoder(
    encoding_method='arbitrary',
    variables=["sex", "cabin", "embarked"],
    )
data = encoder.fit_transform(data)
```

If we execute `data.head()` we obtain the first 5 rows of the resulting dataframe:

	pclass	survived	sex	age	sibsp	parch	fare	cabin	embarked
0	1	1	0	29.0000	0	0	211.3375	0	0
1	1	1	1	0.9167	1	2	151.5500	1	0
2	1	0	0	2.0000	1	2	151.5500	1	0
3	1	0	1	30.0000	1	2	151.5500	1	0
4	1	0	0	25.0000	1	2	151.5500	1	0

Figure 4.24: View of the Titanic dataset.

Let's separate the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('survived', axis=1),
    data['survived'],
    test_size=0.3,
    random_state=0,
)
```

Let's print out the variable names:

```
X_train.columns
```

We obtain the following array:

```
Index(['pclass', 'sex', 'age', 'sibsp', 'parch', 'fare', 'cabin', 'embarked'],
dtype='object')
```

Let's create a mask flagging discrete variables:

```
discrete_vars = [True, True, False, True, True, False, True, True]
```

Now, let's calculate the mutual information of these discrete or continuous variables against the target, which is discrete:

```
mi = mutual_info_classif(
    X_train,
    y_train,
    discrete_features=discrete_vars,
    random_state=42,
    )
```

If we execute `mi` we obtain the MI between the features and the target:

```
array([0.05762333, 0.13060906, 0.00353772, 0.01505698, 0.01739128,
       0.13471196, 0.04897479, 0.01454174])
```

Now, let's capture the array in a pandas series, add the variable names in the index, sort the features based on the MI and make a bar plot:

```
mi = pd.Series(mi)
mi.index = X_train.columns
mi.sort_values(ascending=False).plot.bar(figsize=(10, 5))
plt.ylabel('Mutual Information')
plt.title("Mutual information between predictors and target")
```

We obtain the following plot with the MI between each feature and the target:

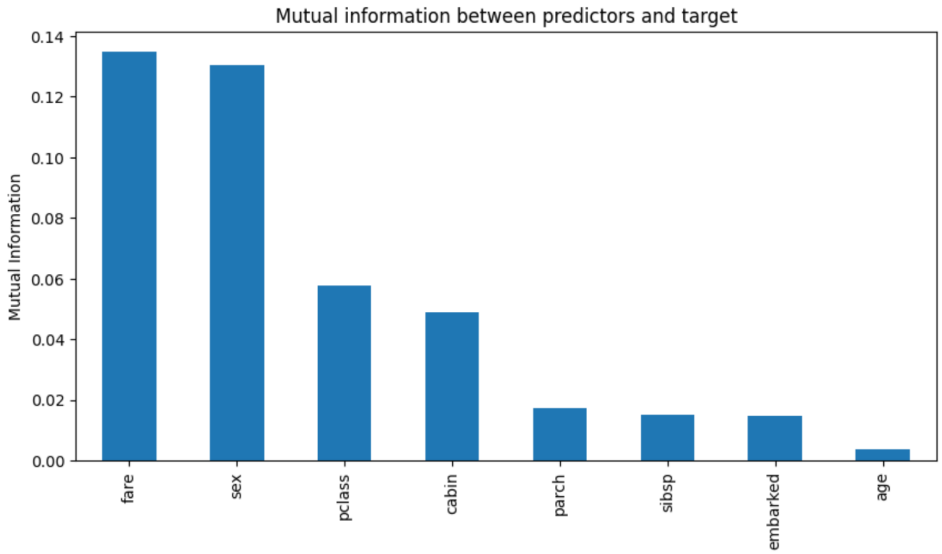


Figure 4.25: MI between features and target.

In this case, all features show MI greater than 0, so we could select them all. For larger datasets, you can select the top k features with `SelectKBest`, or the features in the top percentile using `SelectPercentile` from scikit-learn.

4.5 References

Chi-square

- Pearson, On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling, Philosophical Magazine, 50: 157-175, 1900.

ANOVA and correlation

- Field, et. al., Discovering statistics with R, Chapter 10, 2012.

Mutual information

- Cover, Thomas, Elements of information theory, John Wiley & Sons, Ltd., Chapter 2, 2005
- Ross, Mutual Information between Discrete and Continuous Data Sets, PLoS ONE 9(2): e87357, 2014.
- Kraskov, Stoegbauer, Grassberger, Estimating mutual information, Physical Review E 69: 066138, 2004.

5 Univariate Feature Selection

In the previous chapter, we introduced statistical tests as methods for ranking variables and then selecting the top-ranked features. In this chapter, we will continue exploring methods for ranking variables. We will cover two approaches: the first ranks variables based on the performance metric of a machine learning model trained with a single feature. After that, we'll discuss target mean encoding as a proxy for assessing the importance of variables.

In this chapter, we will discuss:

- Single feature models.
- Target mean encoding.

5.1 Single feature models

One way to rank variables is by evaluating the performance of a classifier or regression model trained on each individual variable. The procedure works as follows:

1. Train one classifier or regression model per feature.
2. Determine the performance of each model.
3. Rank the features based on model performance.
4. Select the top-ranking features.

The described procedure is illustrated in the following image, where we have a dataset with 4 variables, we train 4 decision tree models using each of the features, then determine, in this case, each model's ROC-AUC, and finally we rank the features. Here, we select those features whose models returned ROC-AUC values greater than 0.55:



Figure 5.1: Selection using single feature models.

Advantages of using single feature models for feature selection

Using single feature models to assess feature importance is very flexible. We can use any machine learning model to assess the value of the features, which allows us to evaluate features with both linear and non-linear relationships with the target.

> **Tip**
>
> If we want to train random forests or gradient boosting machines, selecting features based on single feature decision trees is usually enough; we don't need the added complexity of the ensembles. If we want to train linear models, then we should rank and select features based on linear or logistic regression.

Machine learning models can be trained with numerical and categorical variables, which makes it possible to compare and select among both types of variables simultaneously. In addition, we can rank features based on any performance metric, such as ROC-AUC or accuracy for classification, or mean squared error (MSE) or $R2$ for regression.

> **Important**
>
> Remember that the encoding performed to the categorical variable may affect its importance.

Limitations of single feature models

The use of single feature models has some limitations. First, we need to train as many models as features, which makes the process computationally expensive, certainly more than filter methods based on statistical tests. Second, the choice of threshold to select the top-ranked features is arbitrary. Additionally, single-feature classifiers or regression models do not account for feature interactions, which means that, if a feature is important in combination with a second feature, this interaction will be lost.

More importantly, single feature models will rank identical or highly correlated features similarly, hence, we will probably select them both, introducing redundant variables in the final dataset, at the expense of variables that could add value but are ranked lower.

Python implementation

We'll start by selecting features based on single feature classifiers or regression models utilizing Feature-engine. After that, we'll implement the selection algorithm manually utilizing pandas and Scikit-learn to better understand how this selection procedure works under the hood. The first demos will be using a classification dataset, followed by demos using a regression dataset.

Let's begin by importing the libraries, functions, and classes. We will work with 2 datasets available in Scikit-learn; the breast cancer dataset for classification and the California housing data for regression.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer, fetch_california_housing
from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.metrics import roc_auc_score, mean_squared_error

from feature_engine.selection import SelectBySingleFeaturePerformance
```

Classification

We start by loading the breast cancer dataset and separating it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Feature-engine

Feature-engine has the class `SelectBySingleFeaturePerformance()` to select features based on single feature models. We can specify:

- The machine learning model that we want to use for selection.
- The cross-validation framework.
- The performance metric to optimize.
- The threshold to select features.

> **Tip**
>
> If we set the `threshold` to `None`, `SelectBySingleFeaturePerformance()` will select those features that return single feature models whose performance is greater than the mean performance of all models. Alternatively, we can specify an arbitrary value for `threshold`.

Here, we'll select features based on single feature decision trees, determining the ROC-AUC of each model with 3 fold cross-validation, and selecting those features for which the model has a performance greater than the mean performance of all single feature models combined:

```
sel = SelectBySingleFeaturePerformance(
    estimator=DecisionTreeClassifier(random_state=10),
    scoring="roc_auc",
    cv=3,
    threshold=None,
)
```

With fit(), we trigger the single feature model training and feature selection:

```
sel.fit(X_train, y_train)
```

We can display the features that will be removed by executing:

```
sel.features_to_drop_
```

This will return the following output with the features that resulted in models with performance that was lower than the mean performance of all features:

```
['mean texture',
 'mean smoothness',
 'mean symmetry',
 'mean fractal dimension',
 'texture error',
 'smoothness error',
 'compactness error',
 'concavity error',
 'concave points error',
 'symmetry error',
 'fractal dimension error',
 'worst texture',
```

```
 'worst smoothness',
 'worst symmetry',
 'worst fractal dimension']
```

We can also return the performance of each single feature model by executing:

```
sel.feature_performance_
```

This will return the following dictionary with the ROC-AUC value for each single feature model:

```
{'mean radius': 0.8051021129248817,
 'mean texture': 0.603808918097661,
 'mean perimeter': 0.8171860645890751,
 'mean area': 0.8124867500529999,
 'mean smoothness': 0.6064235743057028,
 'mean compactness': 0.7132004805314112,
 'mean concavity': 0.7906155042046499,
 'mean concave points': 0.8486679386615789,
 'mean symmetry': 0.5672037311850753,
 'mean fractal dimension': 0.5348031941205569,
 'radius error': 0.7361317221397782,
 'texture error': 0.48441806232775075,
 'perimeter error': 0.7451416860999224,
 'area error': 0.7631262808282101,
 'smoothness error': 0.5168539325842696,
 'compactness error': 0.5755423644972087,
 'concavity error': 0.6033142534096531,
 'concave points error': 0.6228181753939651,
 'symmetry error': 0.5144512755282312,
 'fractal dimension error': 0.553812451416861,
 'worst radius': 0.8500459331496008,
 'worst texture': 0.6172708642498764,
 'worst perimeter': 0.8922337643982757,
 'worst area': 0.8529079217016465,
 'worst smoothness': 0.5829623348173274,
 'worst compactness': 0.6945092219631123,
 'worst concavity': 0.7851741926365629,
 'worst concave points': 0.8753444986220056,
 'worst symmetry': 0.5352625256165642,
 'worst fractal dimension': 0.5759310296092149}
```

`SelectBySingleFeaturePerformance()` uses cross-validation to examine model performance, and therefore, it also obtains the dispersion of the performance (i.e., the standard deviation), which we can display like this:

```
sel.feature_performance_std_
```

In the following dictionary, we see the features as keys and the standard deviation of the single feature models' performance as values:

```
{'mean radius': 0.01619964238671276,
 'mean texture': 0.017829018048097646,
 'mean perimeter': 0.032604745140086044,
 'mean area': 0.017878664980592367,
 'mean smoothness': 0.01953097919513281,
 'mean compactness': 0.0362930880383841,
 'mean concavity': 0.011558761123656296,
 'mean concave points': 0.03128326672622833,
 'mean symmetry': 0.015997158770331525,
 'mean fractal dimension': 0.02655161842468518,
 'radius error': 0.027163527446792114,
 'texture error': 0.018660943620689725,
 'perimeter error': 0.03558117345885875,
 'area error': 0.0092186457982406,
 'smoothness error': 0.015043228385997549,
 'compactness error': 0.027518591461539163,
 'concavity error': 0.053296416284582725,
 'concave points error': 0.032242733198683725,
 'symmetry error': 0.011851218219054776,
 'fractal dimension error': 0.04533844724871236,
 'worst radius': 0.032983908501455404,
 'worst texture': 0.03577118816746036,
 'worst perimeter': 0.007920153895212535,
 'worst area': 0.026815495178058275,
 'worst smoothness': 0.0257885347746508,
 'worst compactness': 0.04254559426016175,
 'worst concavity': 0.011268020284886205,
 'worst concave points': 0.018198000292806406,
 'worst symmetry': 0.016866335378444994,
 'worst fractal dimension': 0.029070994339610624}
```

We can capture both dictionaries into a pandas dataframe and make a bar plot with each single feature model's mean performance and standard deviation:

```
r = pd.concat([
        pd.Series(sel.feature_performance_),
        pd.Series(sel.feature_performance_std_)
    ], axis=1
)
r.columns = ['mean', 'std']

r.sort_values(by="mean", ascending=False)['mean'].plot.bar(
    yerr=[r['std'], r['std']], subplots=True, figsize=(10,5))

plt.title("Single feature model Performance")
plt.ylabel('roc-auc')
plt.xlabel('Features')
plt.show()
```

The previous commands return the following image, where we see the mean performance of each single feature model together with their standard deviation:

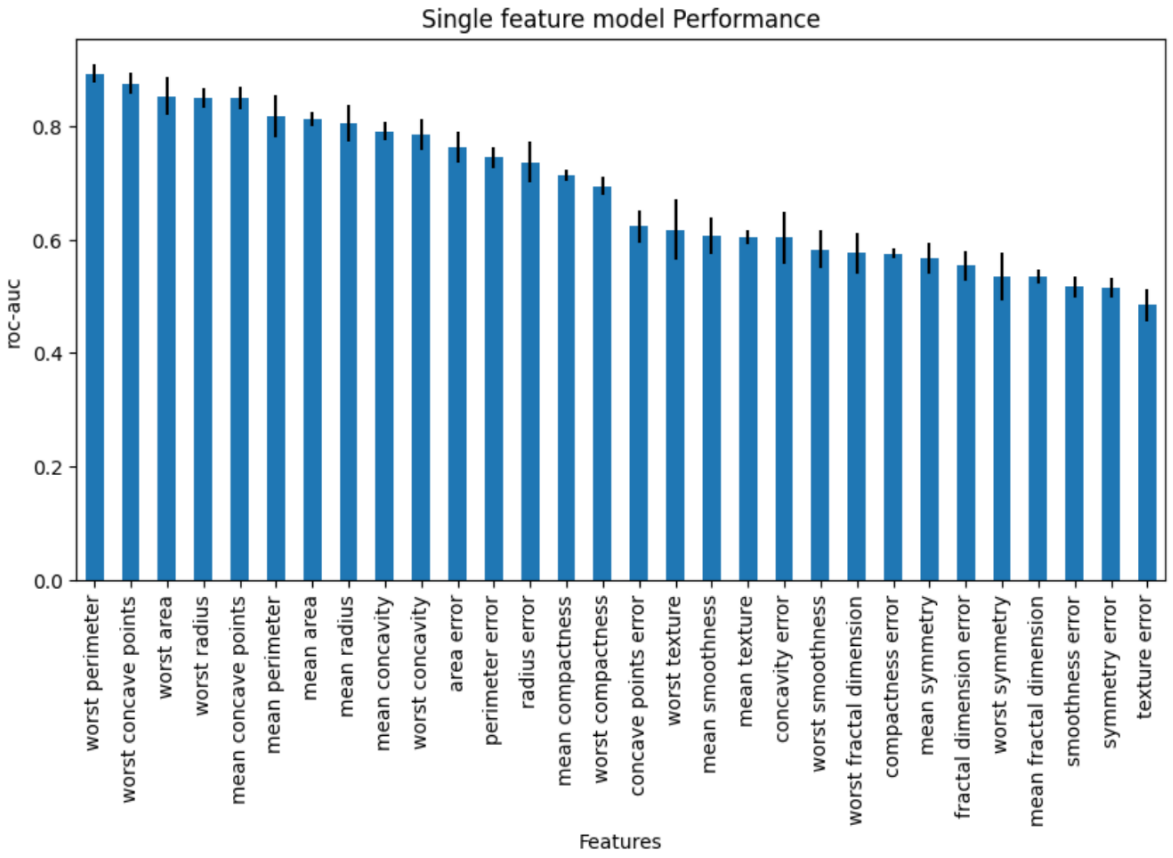


Figure 5.2: Performance returned by the single feature classifiers.

> **Tip**
>
> Some of the features returned classifiers with a ROC-AUC value close to 0.5, which essentially indicates a random guess. Therefore, when using ROC-AUC, we could select features that result in classifiers with ROC-AUC values greater than 0.5, instead of using the mean ROC-AUC of all features, as we did in this demo.

With `transform()`, we remove the features from the data:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

And that's it! We've now reduced the datasets to the selected features.

Pandas and Scikit-learn

Feature-engine trains a machine learning model per feature using cross-validation to determine its performance, and then selects the features based on a threshold. We'll carry out all of those steps manually (except the cross-validation) using pandas and Scikit-learn.

Let's start by training a decision tree classifier per feature and then obtain and store its ROC-AUC:

```
# Here we store the roc-auc values.
roc_values = []

# Iterate over each feature in the dataset.
for feature in X_train.columns:

    # Train a decision tree classifier.
    clf = DecisionTreeClassifier(random_state=0)
    clf.fit(X_train[feature].to_frame(), y_train)

    # Obtain the predictions of the decision tree.
    y_scored = clf.predict_proba(X_test[feature].to_frame())

    # Calculate and store the roc-auc.
    roc_values.append(roc_auc_score(y_test, y_scored[:, 1]))

# Display the result.
roc_values[0:10]
```

The output of the previous code consists of a list with the ROC-AUC values for the first 10 single feature classifiers:

```
[0.8463312368972746,
 0.5570230607966458,
 0.8484276729559749,
 0.8561844863731657,
 0.5912997903563941,
 0.7340670859538784,
 0.8773584905660378,
 0.850104821802935,
 0.5628930817610063,
 0.4763102725366876]
```

A ROC-AUC value of 0.5 indicates a random decision. By executing `len(roc_values[roc_values > 0.5])`, we obtain the number of features that returned models with ROC-AUC values greater than 0.5, which is `28` features.

We can now select the variables that returned models with ROC-AUC greater than 0.5:

```
selected_features = roc_values[roc_values > 0.5].index
```

By executing `selected_features`, we can see the list of features that returned classifiers with performance greater than 0.5:

```
Index(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'radius error', 'texture error',
       'perimeter error', 'area error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst compactness', 'worst concavity',
       'worst concave points', 'worst symmetry', 'worst fractal dimension'],
      dtype='object')
```

Finally, we can retain those variables in the data:

```
X_train = X_train[selected_features]
X_test = X_test[selected_features]
```

Regression

The procedure to select features based on single feature regression models is identical to that of single feature classifiers. We just need to train a regression algorithm for every feature and utilize a regression model evaluation metric instead.

For completeness, let's select features from a regression dataset. Let's load the California housing data and separate it into a training and a testing set:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Feature-engine

We'll set up `SelectBySingleFeaturePerformance()` to select variables using a decision tree model for regression with 3-fold cross-validation, optimizing the mean squared error (MSE).

> **Note**
>
> Scikit-learn minimizes the performance metrics; for this reason, we use the negative of the MSE.

```
sel = SelectBySingleFeaturePerformance(
    estimator=DecisionTreeRegressor(random_state=10),
    scoring='neg_mean_squared_error',
    cv=3,
    threshold=None,
)
```

With `fit()`, we trigger the training of the single feature regression models and the selection of the variables:

```
sel.fit(X_train, y_train)
```

By executing `sel.features_to_drop_`, we display the variables that will be removed from the data:

```
['AveRooms', 'AveBedrms', 'Population', 'AveOccup']
```

Let's create a bar plot with the mean and standard deviation of the MSE for each feature:

```
r = pd.concat([
        pd.Series(sel.feature_performance_),
        pd.Series(sel.feature_performance_std_)
    ], axis=1
)
r.columns = ['mean', 'std']

r.sort_values(by="mean", ascending=False)['mean'].plot.bar(
    yerr=[r['std'], r['std']], subplots=True, figsize=(10,5))

plt.title("Single feature model Performance")
```

```
plt.ylabel('negative mse')
plt.xlabel('Features')
plt.show()
```

The previous code returns the following image, where we can see the features ranked based on the negative of the MSE:

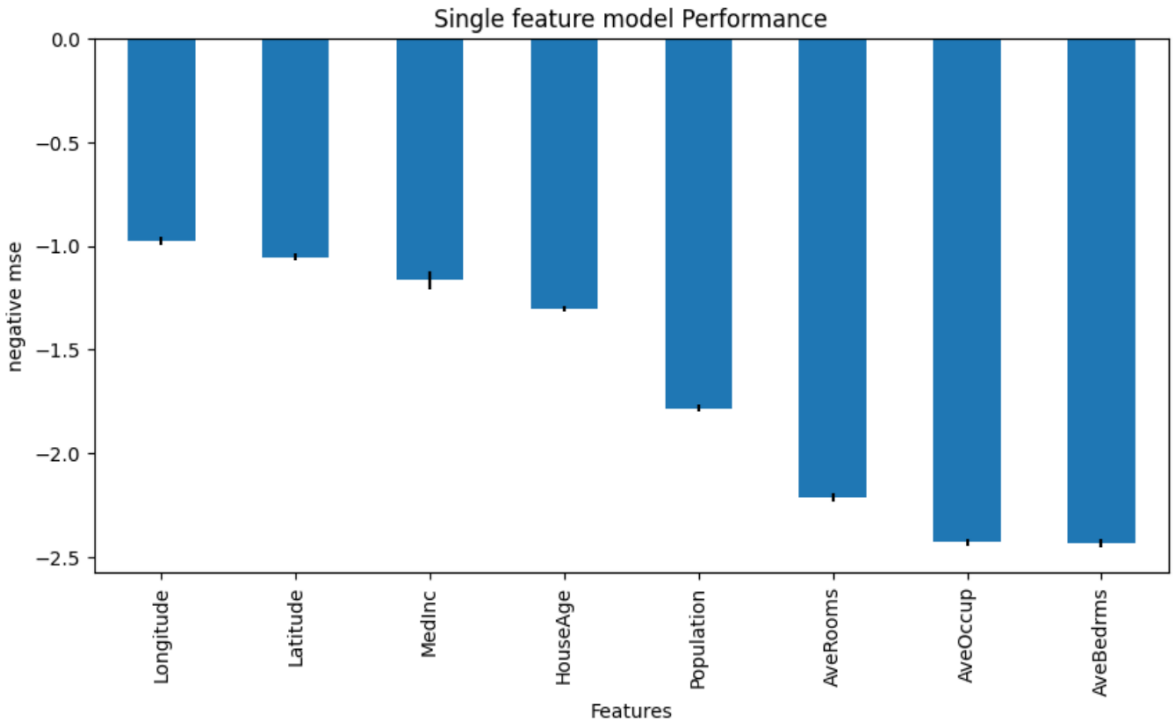


Figure 5.3: Single feature regression model performance.

We can now go ahead and remove the features:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

And that's it! We've now reduced the datasets to the selected features.

Pandas and Scikit-learn

We can select features using single feature regression models manually by utilizing pandas and Scikit-learn. In the following code block, we train a decision tree regressor per feature, obtain the MSE, and store the values in a list:

```
# Here we store the mse values.
mse_values = []

# Iterate over each variable.
for feature in X_train.columns:

    # Train a decision tree.
    clf = DecisionTreeRegressor(random_state=0)
    clf.fit(X_train[feature].to_frame(), y_train)

    # Make a prediction using the decision tree.
    y_scored = clf.predict(X_test[feature].fillna(0).to_frame())

    # Determine the mse and store it.
    mse_values.append(mean_squared_error(y_test, y_scored))


# Display the result.
mse_values[0:10]
```

The previous code returns a list with the MSE values of the first 10 single feature regression models:

```
[1.1895059201655,
 1.2936868384687787,
 2.1619855195998774,
 2.2993323904987784,
 1.6449156583299118,
 2.413230964239309,
 1.0023595654456663,
 0.9584366389417938]
```

Let's capture the MSE values in a pandas Series, add the variable names in the index, sort the features based on the MSE, and make a bar plot:

```
mse_values = pd.Series(mse_values)
mse_values.index = X_train.columns
mse_values.sort_values(ascending=False).plot.bar(figsize=(20,8))
```

```
plt.ylabel('mse')
plt.title('Univariate performance')
```

In the following image, we see the MSE returned by each single feature regression model:

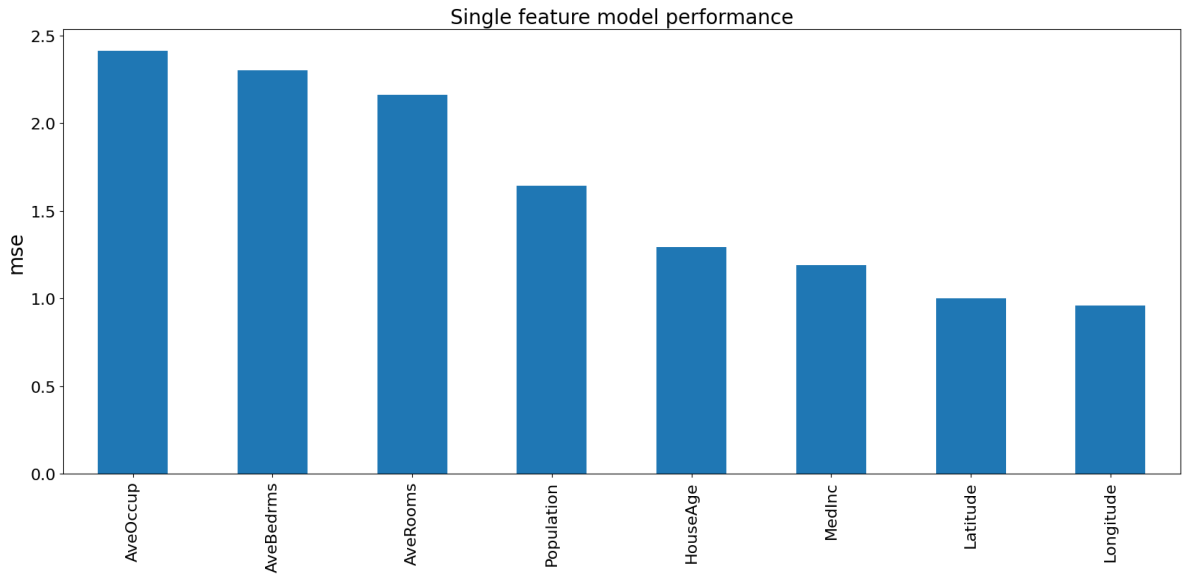


Figure 5.4: Single feature regression model performance.

> **Note**
>
> Remember that for regression, the smaller the MSE, the better the model performance is.

Finally, let's select those features that returned regression models with MSE greater than the mean MSE of all single feature models:

```
selected_features = mse_values[mse_values < np.mean(mse_values)].index
```

By executing `selected_features`, we see the selected variables:

```
Index(['MedInc', 'HouseAge', 'Latitude', 'Longitude'], dtype='object')
```

Now, we can reduce the datasets to the selected variables:

```
X_train = X_train[selected_features]
X_test = X_test[selected_features]
```

That's it! We have selected variables based on single feature classifiers and regression models.

5.2 Target encoding

Selecting features by target encoding was introduced by data scientists from the University of Melbourne in the KDD 2009 data science competition. This feature selection procedure ranks the predictive power of each feature by replacing the feature values with the target posterior probability and using the new vector to predict the target.

The idea behind this method is to use the mean value of the target at each level of the variable as an estimate for the target value. For categorical variables, the mean target value per category is used as the prediction. For continuous variables, the values are first discretized into intervals, and then each interval is replaced with the mean target value.

This selection method works as follows:

1. Split the data into training and test sets.
2. Sort continuous variables into discrete intervals.
3. Using the training set, calculate the mean target value per category or interval.
4. Replace the variables' categories and intervals in the test set with the mean target values from *step 3*.
5. Calculate a performance metric using the encoded variable as the prediction and the target as the ground truth.

> **Note**
>
> The train-test split from *step 1* can be replaced with cross-validation. In fact, this is what Feature-engine's implementation does under the hood: it calculates the interval limits and mean target value per category using the training folds, and then evaluates the feature importance using the left out fold, finally taking the average.

To illustrate this method with an example, let's say we have the variable *city* with values *London*, *Manchester*, and *Bristol*, and we want to predict the fraud rate. We take the training set and determine the fraud rate per city, say 0.6 for London, 0.1 for Manchester, and 0.2 for Bristol. Next, we replace London, Manchester, and Bristol in the testing set with 0.6, 0.1, and 0.2, and using these values and the target, we determine a performance metric, like the accuracy. For

the sake of completion, let's say the accuracy for fraud rate obtained using the encoded *city* variable is 0.8. Then, we use this accuracy to rank the feature.

> **Note**
>
> We can use any classification or regression metric to rank the features (*step 5*).

If we have a continuous variable instead, like *income*, with values varying between 10 and 100 (in thousands), we first sort the variable into intervals, say 10-20, 20-30, 30-40, and so on. Then, we calculate the mean fraud per interval using the training set, and replace the intervals with the mean target value in the test set. Finally, we use the encoded variable to determine the fraud accuracy. For the sake of completion, let's say it is 0.7; then the feature *income* ranks after the feature *city*.

> **Note**
>
> We can sort numerical variables into equal-width or equal-frequency intervals before performing the encoding (*step 2*). Keep in mind that the way in which we construct the intervals may affect the importance of the feature.

Advantages of using the target posterior probability as a feature ranking method

Using the target posterior probability to select features has several advantages:

- **Speed**: calculating the mean and quantiles (for the intervals) is computationally efficient.
- **Robust to outliers**: Extreme values in continuous variables are incorporated into the external intervals, preventing them from skewing the distribution.
- It does not require categorical encoding.
- Allows the comparison of categorical with numerical variables.
- Captures both linear and non-linear relationships between variables and the target.

> **Important**
>
> Selecting features based on target mean encoding is possible with binary classification and continuous targets, but not with multi-class targets.

Python implementation

In this section, we'll implement feature selection by target encoding in Python. We'll start by using the open-source Python library Feature-engine. After that, we'll implement the selection algorithm manually utilizing pandas and Scikit-learn, to better understand how this selection procedure works under the hood.

Let's begin by importing the libraries, functions, and classes:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from feature_engine.encoding import RareLabelEncoder
from feature_engine.selection import SelectByTargetMeanPerformance
```

Next, we load and prepare the Titaic data that contains both categorical and numerical variables:

```
variables = ['pclass', 'survived', 'sex', 'age',
             'sibsp', 'parch', 'fare', 'cabin', 'embarked']

data = pd.read_csv('https://www.openml.org/data/get_csv/16826755/phpMYEkMl',
                   usecols=variables,
                   na_values='?',
                   dtype={'fare': float, 'age': float},
                   )

data.dropna(subset=['embarked', 'fare'], inplace=True)
data['age'] = data['age'].fillna(data['age'].mean())
```

```
def get_first_cabin(row):
    try:
        return row.split()[0]
    except:
        return 'N'


data['cabin'] = data['cabin'].apply(get_first_cabin).str[0]
encoder = RareLabelEncoder(variables='cabin', n_categories=2)
data = encoder.fit_transform(data)

data.head()
```

In the following image, we see the first 5 rows of the Titanic dataset:

	pclass	survived	sex	age	sibsp	parch	fare	cabin	embarked
0	1	1	female	29.0000	0	0	211.3375	Rare	S
1	1	1	male	0.9167	1	2	151.5500	C	S
2	1	0	female	2.0000	1	2	151.5500	C	S
3	1	0	male	30.0000	1	2	151.5500	C	S
4	1	0	female	25.0000	1	2	151.5500	C	S

Figure 5.5: First rows of the Titanic dataset.

Lets' separate the data into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('survived', axis=1),
    data['survived'],
    test_size=0.3,
    random_state=0,
)
```

Feature-engine

To select features using the target posterior distribution with Feature-engine, we use `SelectByTargetMeanPerformance()`. Here, we set up `SelectByTargetMeanPerformance()` to discretize continuous variables into 5 intervals of equal-frequency, and select features using 2-fold cross-validation and the ROC-AUC as evaluation metric.

```
sel = SelectByTargetMeanPerformance(
    bins=5,
    strategy='equal_frequency',
    scoring='roc_auc',
    cv=2,
    threshold=None,
    regression=False,
)
```

> **Tip**
>
> If we leave `threshold` to None, `SelectByTargetMeanPerformance()` will select features whose performance is greater than the mean performance of all features.

With `fit()`, we trigger the discretization of the continuous variables, the replacement of intervals and categories with the target posterior probability (that is, the target mean), and the calculation of the performance metric using the encoded variable as prediction and the target as truth:

```
sel.fit(X_train, y_train)
```

We can explore the features that will be removed by executing:

```
sel.features_to_drop_
```

The output is:

```
['age', 'sibsp', 'parch', 'embarked']
```

We can also explore the performance attributed to each variable by executing:

```
 sel.feature_performance_
```

In the following output, we see the mean ROC-AUC returned by encoding each feature with the target mean:

```
{'pclass': 0.6551893232380663,
 'sex': 0.7491001943282519,
 'age': 0.5382483653274581,
 'sibsp': 0.5183970828336429,
 'parch': 0.5068146044866522,
 'fare': 0.6753892717910286,
 'cabin': 0.6385761238147736,
 'embarked': 0.5630695122556864}
```

From 2024, `SelectByTargetMeanPerformance()` also returns the standard deviation of the feature performance, which we can display by executing:

```
sel.feature_performance_std_
```

Below, we see the output of this command:

```
{'pclass': 0.026413779804051962,
 'sex': 0.013795301386788084,
 'age': 0.010406279303332377,
 'sibsp': 0.010957939301932451,
 'parch': 0.003453652453834666,
 'fare': 0.015961607539624112,
 'cabin': 0.011021780413324356,
 'embarked': 0.0099981941737844 47}
```

We can capture the mean and standard deviation of the feature importance values in a pandas DataFrame and create a bar plot:

```
r = pd.concat([
    pd.Series(sel.feature_performance_),
    pd.Series(sel.feature_performance_std_)
], axis=1
)
r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True)

plt.title("FFeature performance based on target encoding")
plt.ylabel('ROC-AUC')
plt.xlabel('Features')
plt.show()
```

In the following image, we can see the mean and standard deviation of the importance assigned to each feature:

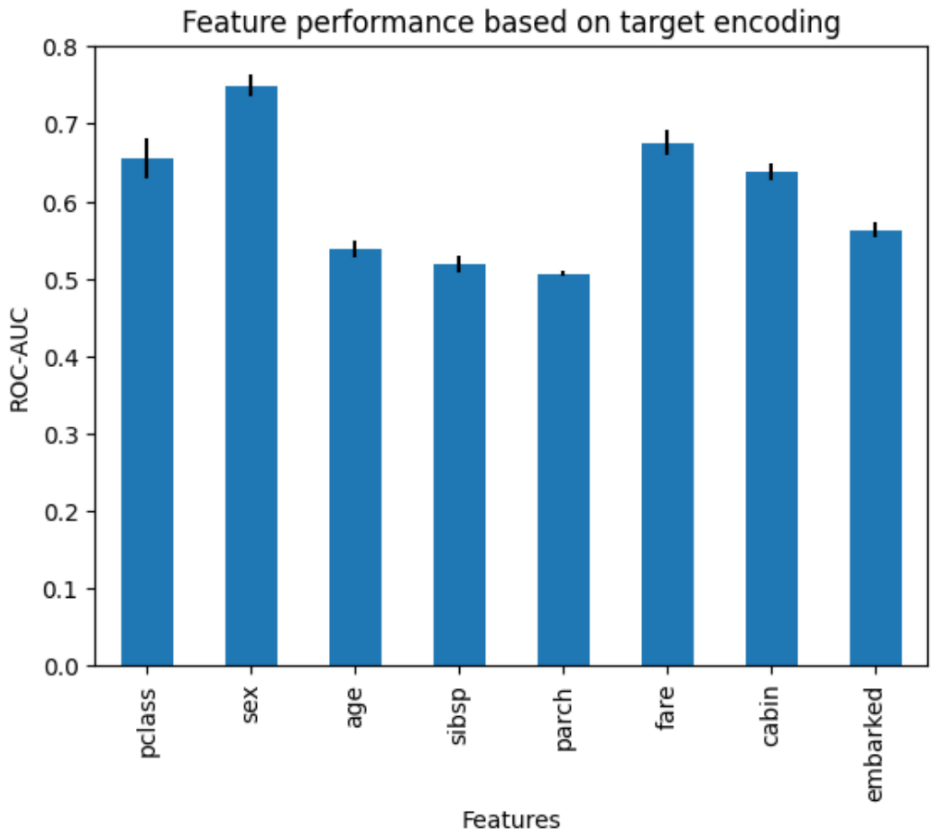


Figure 5.6: ROC-AUC resulting from the mean encoded features.

We can now go ahead and remove the features from the data:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

We can display the resulting dataframe by executing `X_train.head()`, which returns the following image:

	pclass	sex	age	sibsp	parch	fare	cabin	embarked
842	3	male	29.813199	1	0	19.9667	N	S
868	3	male	43.000000	0	0	6.4500	N	S
429	2	male	44.000000	0	0	13.0000	N	S
480	2	male	25.000000	1	2	41.5792	N	C
1308	3	male	29.000000	0	0	7.8750	N	S

Figure 5.7: Dataframe with the selected variables.

And that's it! We've now reduced the dataset using the target posterior probability as proxy for feature importance.

Manual implementation

To better understand the inner workings of the selection algorithm, we will implement the selection process manually. We start by selecting among categorical variables. The Titanic dataset contains 4 categorical variables, which are `Sex`, `Pclass`, `Cabin`, and `Embarked`.

Let's separate the original data into training and testing sets containing only the categorical variables:

```
X_train, X_test, y_train, y_test = train_test_split(
    data[['pclass', 'sex', 'embarked', 'cabin']],
    data['survived'],
    test_size=0.3,
    random_state=0)
```

In the next code block, we create a function that first makes a copy of the train and test sets. Next, utilizing the training set, it finds the mean value of the target per category. Finally, it replaces the categories with the mean target value in both datasets, returning dataframes with the encoded variables.

```
def mean_encoding(df_train, df_test, target):

    # Make a copy of the original dataframes.
    df_train_temp = df_train.copy()
    df_test_temp = df_test.copy()

    # Iterate over each variable.
```

```
    for col in df_train.columns:

        # Make a dictionary of categories: target-mean pairs.
        target_mean_dict = target.groupby(df_train[col]).mean().to_dict()

        # Replace the categories by the mean of the target.
        df_train_temp[col] = df_train[col].map(target_mean_dict)
        df_test_temp[col] = df_test[col].map(target_mean_dict)

    # Return the encoded datasets.
    return df_train_temp, df_test_temp
```

Let's replace the categories in the training and testing sets with the target mean value using the previous function:

```
X_train_enc, X_test_enc = mean_encoding(X_train, X_test, y_train)

X_train_enc.head()
```

In the following image, we see that the strings were replaced by the target mean values:

	pclass	sex	embarked	cabin
842	0.243902	0.199664	0.338534	0.295875
868	0.243902	0.199664	0.338534	0.295875
429	0.416667	0.199664	0.338534	0.295875
480	0.416667	0.199664	0.545946	0.295875
1308	0.243902	0.199664	0.338534	0.295875

Figure 5.8: Categorical variable values are replaced by the target mean.

Next, utilizing list comprehension, we calculate the ROC-AUC for each encoded feature, using the encoded variable as the predictions:

```
roc_values = [
    roc_auc_score(y_test, X_test_enc[feature]) for feature in X_test.columns
]
```

By executing `roc_values` we see the ROC-AUC per feature:

```
[0.6303893819900525,
 0.7841636668407024,
 0.5733423099117914,
 0.6118408397680746]
```

We can capture the previous list in a pandas Series, add the variable names, sort the values, and print the result:

```
m1 = pd.Series(roc_values, index=X_test.columns).sort_values(ascending=False)
```

By executing `m1` we see the ROC-AUC obtained for each of the encoded features:

```
sex         0.784164
pclass      0.630389
cabin       0.611841
embarked    0.573342
dtype: float64
```

All features show a ROC-AUC greater than 0.5, so in principle, we could retain them all.

In the next paragraphs, we will repeat the procedure but this time for numerical variables. It requires one additional step, which is dividing the continuous variable into intervals. We will work with the numerical variables `Age` and `Fare`.

Let's separate the data into training and testing sets containing only the numerical variables.

```
X_train, X_test, y_train, y_test = train_test_split(
    data[['age', 'fare']],
    data['survived'],
    test_size=0.3,
    random_state=0,
)
```

Let's divide `Age` into 5 bins of equal number of observations. We use pandas `qcut()` to find the limits of the intervals. The method returns the discretized variable. With `retbins= True`, we return the limits of the intervals as well.

```
X_train['age_binned'], intervals = pd.qcut(
    X_train['age'],
    q = 5,
    labels=False,
```

```
    retbins=True,
    precision=3,
    duplicates='drop',
)

X_train[['age_binned', 'age']].head(10)
```

In the following output, we see the discretized variable on the left, and the original variable on the right. Each interval was assigned an integer value.

	age_binned	age
842	2	29.813199
868	4	43.000000
429	4	44.000000
480	1	25.000000
1308	1	29.000000
455	4	63.000000
117	3	30.000000
484	3	34.000000
296	3	39.000000
262	4	50.000000

Figure 5.9: Dataframe with the original and discretized variable Age.

We can explore the number of unique intervals by executing `X_train['age_binned'].nunique()`, which should return `5`. We can also print out the unique interval values with:

```
X_train['age_binned'].unique()
```

which returns:

```
array([2, 4, 1, 3, 0], dtype=int64)
```

Let's now discretize the variable `Age` in the test set using the interval limits that we determined from the training set and stored in the variable `intervals`:

```
X_test['age_binned'] = pd.cut(x = X_test['age'], bins=intervals, labels=False)
```

> **Note**
>
> Note that we use `pd.cut` instead of `pd.qcut` to discretize the testing set because we are passing the interval limits.

Now, let's discretize the variable `Fare`. First, we discretize the variable in the training set, and then we use the learned limits to sort the variable values into intervals in the testing set:

```
X_train['fare_binned'], intervals = pd.qcut(
    X_train['fare'],
    q=5,
    labels=False,
    retbins=True,
    precision=3,
    duplicates='drop',
)

X_test['fare_binned'] = pd.cut(
    x = X_test['fare'], bins=intervals, labels=False,
)
```

The test set shows some missing data points that were introduced because the variables contained values that were not present in the train set. Feature-engine handles this automatically by extending the limits of the first and last interval to minus and plus infinite. For simplicity, I will just replace the missing data with 0:

```
X_train = X_train.fillna(0)
X_test = X_test.fillna(0)
```

Using the function we created previously, we can replace the intervals in both variables with the target mean value per interval:

```
binned_vars = ['age_binned', 'fare_binned']

X_train_enc, X_test_enc = mean_encoding(
    X_train[binned_vars], X_test[binned_vars], y_train)
```

Using list comprehension, we can obtain the ROC-AUC values using the encoded variables as predictions:

```
roc_values = [
    roc_auc_score(y_test, X_test_enc[feature]) for feature in binned_vars
]
```

Let's capture the ROC-AUC values into a pandas series, add the variable names, and sort the values:

```
m1 = pd.Series(roc_values, index=binned_vars).sort_values(ascending=False)
```

By executing `m1` we obtain the ROC-AUC per numerical feature:

```
fare_binned    0.666964
age_binned     0.497335
dtype: float64
```

According to the target mean feature selection method, the variable `Age` has no predictive value, because it returns a ROC-AUC value close to 0.5. So, we would select the variable `Fare`, and remove the variable `Age`.

> **!** Important
>
> Keep in mind that the way in which we discretize continuous variables (equal-width or equal-frequency intervals) and the number of intervals used for the discretization may enhance or mask the variable's association with the target, and hence its predictive value.

And that's it! You now know how to select features based on target mean encoding.

5.3 References

- Guyon et. al., An Introduction to Variable and Feature Selection, JMLR, 3:1157-1182, 2003.
- Miller et. al., Predicting customer behaviour, JMLR Workshop and Conference Proceedings 7:45-55, 2009.

6 Wrapper Methods

Wrapper methods *wrap* the feature selection procedure around a predictive model. These methods use the regression or classification model as part of the function that evaluates the performance of different feature subsets.

The idea behind wrapper methods is simple: generate all possible subsets of features, evaluate each subset using the machine learning model, and select the subset that results in the best-performing model.

The main challenge with wrapper methods is that, for *n* features, the number of possible feature subsets is 2^n. As a result, evaluating every possible combination of features becomes computationally impractical for larger datasets.

To reduce the number of feature combinations to test, researchers developed alternative strategies to generate subsets, with the 2 most popular being forward search and backward search. In forward search, we start by evaluating all possible models containing 1 feature, and in subsequent rounds we add 1 feature at a time, sequentially. In backward search, we start by training a model with all features, and then remove 1 feature at a time, sequentially.

In the rest of the chapter, we will discuss the three main wrapper algorithms:

- Exhaustive search.
- Forward search.
- Backward search.

6.1 Exhaustive search

The Exhaustive Feature Selection (EFS) algorithm identifies the best subset of features from all possible combinations, based on a performance metric for a given machine learning algorithm.

For example, if the dataset contains 4 features, the EFS algorithm will evaluate the following feature subsets:

- All possible combinations of 1 feature.
- All possible combinations of 2 features.
- All possible combinations of 3 features.

- All 4 features.

EFS trains a machine learning model for each feature subset with cross-validation and evaluates its performance. For 4 features, EFS will examine 2^4 feature subsets, that is, 16 feature combinations. This means that the algorithm will train 16 machine learning models with cross-validation. It's easy to see how this method becomes computationally costly as the number of features increases.

In the following paragraphs, we will implement EFS using the open-source Python library `MLXtend`, which to my knowledge, is the only currently maintained library that supports this method.

Python implementation

In this example, we'll use EFS to select features from the breast cancer dataset. Let's begin by importing the libraries, functions, and classes:

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS
```

Let's load the breast cancer dataset and separate it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

To set up an exhaustive search using MLXtend, we need to define the following parameters:

1. The machine learning algorithm.
2. The performance metric, for example the ROC-AUC.
3. The cross-validation framework.
4. The number of minimum features a subset can have.
5. The maximum number of features a subset can have.

Points 4 and 5 are intended to reduce the number of feature combinations to test, hence, reducing the search time.

> **Tip**
>
> To examine all possible feature combinations, set the minimum number of features to 1 and the maximum to the total features in the dataset.

We'll set up the EFS algorithm to train and evaluate random forests using the ROC-AUC with 2-fold cross-validation:

```
efs = EFS(
    estimator=RandomForestClassifier(n_estimators=3, random_state=0),
    min_features=1,
    max_features=4,
    scoring='roc_auc',
    cv=2,
)
```

With `fit()`, we trigger the search:

```
efs = efs.fit(X_train, y_train)
```

EFS will take several minutes to run. After that, we can find the selected feature subset by executing `efs.best_feature_names_`, which returns the following features: `('compactness error', 'worst area', 'worst concavity', and 'worst symmetry')`.

> **Note**
>
> Setting up EFS for classification and regression is quite similar. The only difference is that you need to use the appropriate model for either regression or classification, along with a suitable evaluation metric.

In the attribute `efs.subsets_`, the transformer stores the evaluated subsets, the score for each cross-validation fold, and the mean score across all folds.

In this demo, we evaluated all possible combinations of 1 to 4 features out of 30 features. That means that EFS evaluated 31,930 feature subsets. You can obtain this number by executing `len(efs.subsets_.keys())`.

Let's print out the details of the first subset by executing:

```
efs.subsets_[0]
```

In the following output, we see a feature subset containing only 1 feature: `mean radius`, along with the ROC-AUC returned by the models trained using this feature on each cross-validation fold, as well as the average ROC-AUC across folds:

```
{'feature_idx': (0,),
 'cv_scores': array([0.83105044, 0.86616541]),
 'avg_score': 0.8486079287582265,
 'feature_names': ('mean radius',)}
```

Let's display a different subset:

```
efs.subsets_[60]
```

This subset consists of 2 features, `mean texture` and `mean area`, and we see the performance (ROC-AUC) of the model trained using these features in each cross-validation fold as well as the average:

```
{'feature_idx': (1, 3),
 'cv_scores': array([0.86057056, 0.90953947]),
 'avg_score': 0.8850550192906221,
 'feature_names': ('mean texture', 'mean area')}
```

Finally, we can go ahead and remove the features that were not selected from the data:

```
X_train_t = efs.transform(X_train)
X_test_t = efs.transform(X_test)
```

The final result consists of numpy arrays containing the 4 selected features. By executing `X_test_t` we can see the transformed data:

```
    array([[2.265e-02, 8.444e+02, 5.106e-01, 3.585e-01],
           [8.082e-03, 6.329e+02, 1.390e-01, 2.444e-01],
           [9.238e-03, 6.889e+02, 6.260e-02, 2.136e-01],
           [1.377e-02, 8.197e+02, 1.565e-01, 2.636e-01],
         ...
           [2.277e-02, 2.232e+03, 6.810e-01, 3.643e-01]])
```

That's it! We've now concluded the feature selection using exhaustive search.

> **!** Important
>
> Note that the features were selected using random forests, making them suitable for this model's performance. Different models may require different feature subsets. In other words, exhaustive search is not model agnostic.

6.2 Forward feature selection

In forward selection (FS), variables are progressively added to larger subsets. The algorithm starts by training all possible single-variable machine learning models. It then selects the feature that results in the best-performing classifier or regression model. In the next step, it trains models for all combinations of the feature from the previous step with the remaining variables in the dataset, one at a time. The pair of features that results in the best performing model is selected. This process continues, adding one feature at a time to the subset from the previous step until a stopping criterion is met. By progressively evaluating promising features, forward feature selection is more efficient than exhaustive search.

To illustrate this with an example, let's start with a dataset containing 4 features. In the first step, we train 4 single feature classifiers, and select the feature that results in the model with the highest ROC-AUC. In the second round: we add every single remaining feature to the winning feature from step one, and select the feature pair that results in the best performing decision tree. Next, we add every remaining feature to the winning feature pair and select the three features resulting in the best-performing decision tree, and so on:

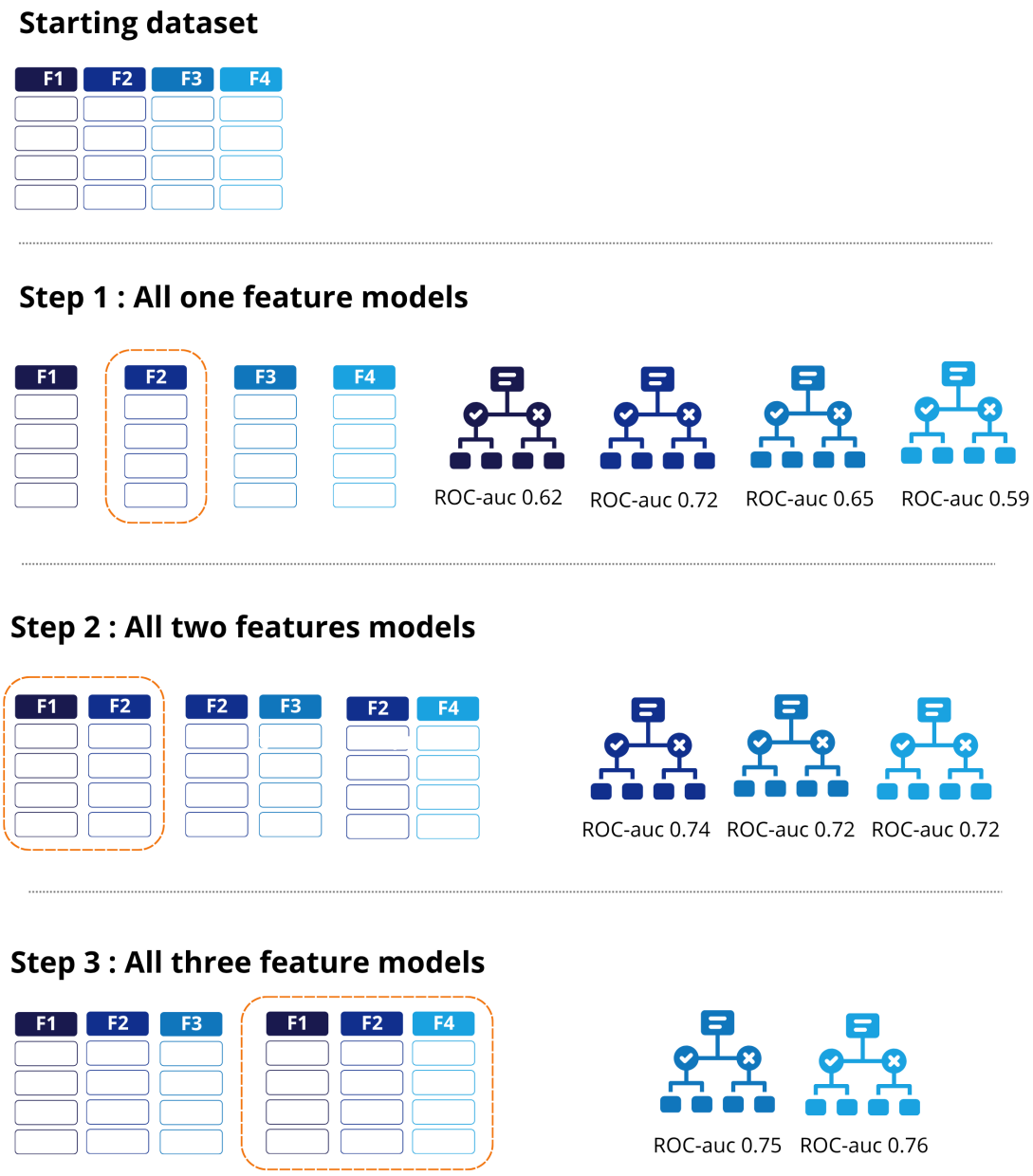


Figure 6.1: Diagram illustrating the forward selection procedure. Orange boxes highlight the best feature subset obtained at each round.

FS has the advantage that, by starting with smaller feature subsets, it is more computationally efficient than other wrapper methods; models with fewer features are faster to train. However, for this same reason, it does not consider feature interactions, or at least not until sufficient features have been added to the subset.

In FS, we need a criterion to stop the search. The most obvious stopping condition consists of stopping the search when adding more features does not improve model performance. This approach has the advantage of focusing the search on model performance. However, the threshold for performance improvement is arbitrarily set by the user and hard to determine. Alternatively, we can stop the search after selecting a predefined number of features.

In the next section, we will implement FS with Scikit-learn and MLXtend. With both Python implementations we can stop the search after selecting a predefined number of features. With Scikit-learn we can also stop the search by using a threshold on model performance improvement.

> **Note**
>
> FS, like all wrapper methods, is not model agnostic. That means that the features selected for a machine learning model may not be suitable for a different model.

Python implementation

In this demo, we'll use FS to select features from the breast cancer dataset. Let's start by importing the classes, functions and libraries:

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
```

Let's load the breast cancer dataset and separate it into a train and a test set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Scikit-learn

Let's import the `SequentialFeatureSelector` from Scikit-learn:

```
from sklearn.feature_selection import SequentialFeatureSelector as SFS
```

We'll set up `SequentialFeatureSelector` to select features for random forests based on FS and use the ROC-AUC as a performance metric with 3-fold cross validation. We'll stop the search when the ROC-AUC does not increase beyond 0.001.

```
sfs = SFS(
    estimator=RandomForestClassifier(n_estimators=5, random_state=0),
    n_features_to_select='auto',
    tol=0.001,           # Threshold for model performance.
    direction='forward', # Direction of the selection procedure.
    scoring='roc_auc',   # Metric to evaluate.
    cv=3,                # Cross-validation fold.
)
```

With `fit()`, we trigger the forward search:

```
sfs = sfs.fit(X_train, y_train)
```

We can display the names of the selected variables by executing `sfs.get_feature_names_out()`, which returns the following variables:

```
array(['mean smoothness', 'mean concavity', 'worst texture',
       'worst perimeter', 'worst concavity'], dtype=object)
```

Finally, we can reduce the data to the selected variables with `transform()`:

```
X_train_t = sfs.transform(X_train)
X_test_t = sfs.transform(X_test)
```

The previous command returns numpy arrays containing the selected features only.

> **Tip**
>
> We can enforce Scikit-learn's `SequentialFeatureSelector` to return pandas dataframes by settings its transform output to pandas as follows: `SFS(estimator=RandomForestClassifier()).set_output(transform="pandas")`.

MLXtend

Next, let's carry out forward search with MLXtend. Let's import the `SequentialFeatureSelector`:

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

We'll select features for random forests evaluating the ROC-AUC with 3-fold cross-validation, and we'll stop the search after identifying the best 10 features:

```
sfs = SFS(
    estimator=RandomForestClassifier(n_estimators=5, random_state=0),
    k_features=10, # Number of features to select.
    forward=True,  # Direction of  the search.
    verbose=1,     # Display intermediate steps' output.
    scoring='roc_auc',
    cv=3,
)
```

With `fit()`, we trigger the forward search:

```
sfs = sfs.fit(X_train, y_train)
```

We set up `verbose=1`, which makes SFS display information about the search as it proceeds:

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:    0.9s finished
Features: 1/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  29 out of  29 | elapsed:    0.8s finished
Features: 2/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  28 out of  28 | elapsed:    0.8s finished
Features: 3/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  27 out of  27 | elapsed:    0.8s finished
Features: 4/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  26 out of  26 | elapsed:    0.7s finished
Features: 5/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  25 out of  25 | elapsed:    0.7s finished
```

```
Features: 6/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:    0.6s finished
Features: 7/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  23 out of  23 | elapsed:    0.7s finished
Features: 8/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  22 out of  22 | elapsed:    0.7s finished
Features: 9/10[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  21 out of  21 | elapsed:    0.6s finished
Features: 10/10
```

We can find the selected features by executing `sfs.k_feature_names_`, which returns the following variables:

```
('mean perimeter',
 'mean compactness',
 'mean fractal dimension',
 'compactness error',
 'concave points error',
 'worst texture',
 'worst perimeter',
 'worst smoothness',
 'worst concave points',
 'worst symmetry')
```

Finally, we can reduce the data to the selected features:

```
X_train_t = sfs.transform(X_train)
X_test_t = sfs.transform(X_test)
```

The resulting datasets are numpy arrays containing the selected variables. We can check them out by executing `X_test_t`:

```
array([[8.864e+01, 1.469e-01, 7.325e-02, ..., 1.574e-01, 2.051e-01,
        3.585e-01],
       [8.410e+01, 5.205e-02, 5.584e-02, ..., 1.289e-01, 6.005e-02,
        2.444e-01],
       [8.959e+01, 5.581e-02, 5.586e-02, ..., 1.034e-01, 8.216e-02,
```

```
        2.136e-01],
       ...,
       [7.894e+01, 6.307e-02, 5.808e-02, ..., 1.292e-01, 1.070e-01,
        3.110e-01],
       [8.791e+01, 1.047e-01, 6.641e-02, ..., 1.451e-01, 7.407e-02,
        2.710e-01],
       [1.307e+02, 1.666e-01, 6.082e-02, ..., 1.438e-01, 2.247e-01,
        3.643e-01]])
```

And that's it! We've now selected features using forward search.

6.3 Backward feature selection

In backward selection, we start with the full set of variables and progressively eliminate the least promising ones. The process begins by fitting a machine learning model using all the features in the dataset and evaluating its performance. In the next step, it generates all possible feature subsets that exclude one feature, training a model for each subset: it then selects the feature subset that results in the best-performing model. After that, it creates all possible subsets removing an additional feature, training a model per subset, and selects the subset that results in the best performing model, and so on:

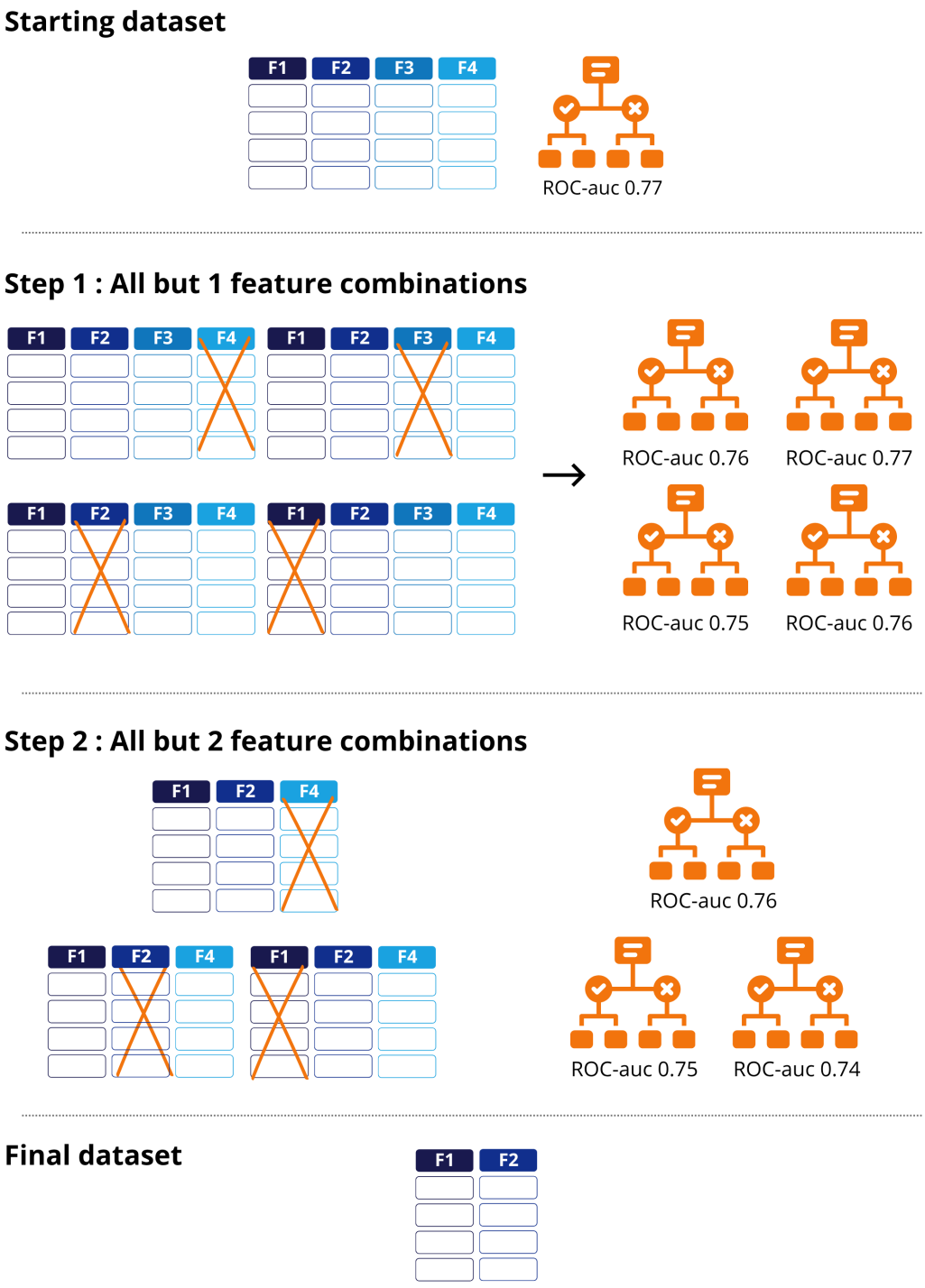


Figure 6.2: Diagram illustrating the backward selection procedure. Orange boxes highlight the best feature subset obtained at each round.

In backward selection, we need a criterion to stop the search. We can stop removing features if the performance of the classifier or regression model degrades beyond a certain threshold. The advantage of this approach is that it focuses the search on model performance. However, the threshold for model degradation is arbitrarily set by the user and often hard to determine. Alternatively, we can stop the search after reaching a predefined number of features.

Backward selection is less computationally efficient than forward selection because it starts by training machine learning models with all features. However, this also allows the search to consider feature interactions, which are ignored, initially, in a forward search. In practice, both forward and backward selection have been used successfully.

In the next section, we will implement backward selection using Scikit-learn and MLXtend. With both Python implementations we can stop the search after selecting a predefined number of features. With scikit-learn, we can also use a threshold on model performance to stop the search.

> **Note**
>
> Backward selection, like all wrapper methods, is not model agnostic. That means that the subset selected for a certain model may not be suitable for a different machine learning algorithm..

Python implementation

We'll use backward selection to select features from the California housing dataset. Let's start by importing the libraries, functions and classes:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
```

Let's load the California housing dataset and separate it into a training and a testing set:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Scikit-learn

Let's import the `SequentialFeatureSelector` from Scikit-learn:

```
from sklearn.feature_selection import SequentialFeatureSelector as SFS
```

Let's select features for random forests using backward search by evaluating r^2 using 3-fold cross-validation. We'll stop the search when the r^2 decreases beyond 0.001:

```
sfs = SFS(
    estimator = RandomForestRegressor(n_estimators=5, random_state=10),
    n_features_to_select="auto", # Number of features to select.
    tol=0.001,                   # Maximum decrease in model performance.
    direction='backward',        # Direction of the selection procedure.
    scoring='r2',                # Metric to measure model performance.
    cv=3,                        # Cross-validation framework.
).set_output(transform="pandas")
```

> **Tip**
>
> Note how we enforce the output of SFS to a pandas dataframe by setting the output of transform to pandas. Alternatively, the default result is a numpy array.

With `fit()` we trigger the search:

```
sfs = sfs.fit(X_train, y_train)
```

We can display the selected features by executing `sfs.get_feature_names_out()` which returns the following variables:

```
array(['MedInc', 'HouseAge', 'AveBedrms', 'AveOccup', 'Latitude',
       'Longitude'], dtype=object)
```

Now we can reduce the datasets to the selected variables:

```
X_train_t = sfs.transform(X_train)
X_test_t = sfs.transform(X_test)
```

The result are pandas dataframes with the selected features.

MLXtend

Let's now carry out backward selection with MLXtend. Let's import the transformer:

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

Let's select features for random forests evaluating r^2 with 3-fold cross-validation. We'll stop the search after selecting a subset of 5 features.

```
sfs = SFS(
    estimator=RandomForestRegressor(
        n_estimators=5, random_state=10),
    k_features=5,
    forward=False,
    verbose=1,
    scoring='r2',
    cv=3,
)
```

With `fit()` we trigger the search:

```
sfs = sfs.fit(X_train, y_train)
```

We set `verbose=1`, which makes SFS display information as it proceeds with the search:

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   8 out of   8 | elapsed:    6.8s finished
Features: 7/5[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   7 out of   7 | elapsed:    5.7s finished
Features: 6/5[Parallel(n_jobs=1)]: Using backend
    SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   6 out of   6 | elapsed:    3.9s finished
Features: 5/5
```

We can display the selected variables by executing `sfs.k_feature_names_`, which returns: `('MedInc', 'HouseAge', 'AveOccup', 'Latitude', 'Longitude')`.

Let's reduce the data to the selected variables:

```
X_train_t = sfs.transform(X_train)
X_test_t = sfs.transform(X_test)
```

MLXtend returns numpy arrays with the selected features. As an example, we can see the first and last rows of the resulting test set by executing `X_test_t`:

```
array([[   4.1518    ,   22.        ,    4.18059299,   32.58      ,
         -117.05      ],
       [   5.7796    ,   32.        ,    3.02097902,   33.92      ,
         -117.97      ],
       [   4.3487    ,   29.        ,    2.91011236,   38.65      ,
         -121.84      ],
       ...,
       [   3.6296    ,   16.        ,    1.88631579,   34.2       ,
         -118.61      ],
       [   5.5133    ,   37.        ,    3.00847458,   33.9       ,
         -118.34      ],
       [   4.7639    ,   36.        ,    2.90545455,   37.66      ,
         -122.44      ]])
```

That's it! We've now selected a subset of 5 features using backward elimination.

6.4 References

- Guyon et. al., An Introduction to Variable and Feature Selection, JMLR, 3:1157-1182, 2003
- Kohavi et. al., Wrappers for feature subset selection, Artificial Intelligence, 97: 273-324, 1997.

7 Embedded Methods

Embedded methods "embed" the selection procedure in the training or induction of the predictive model. In other words, the search for an optimal subset of features is built into the construction of the classifier or the regression algorithm, and as such, embedded approaches train only **one** machine learning model to select features.

A typical embedded feature selection workflow involves:

- Training a machine learning model.
- Deriving feature importance.
- Selecting the top ranking features.

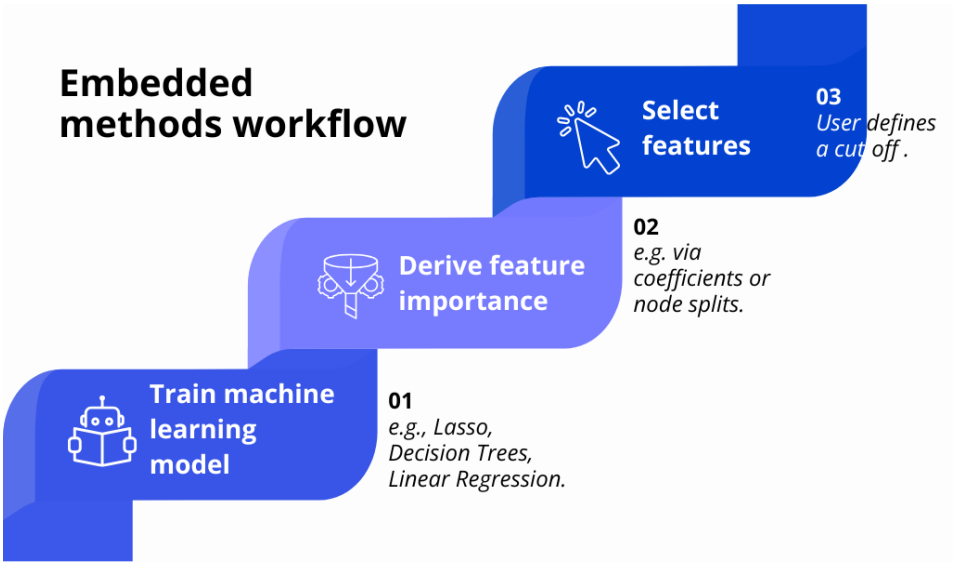


Figure 7.1: Embedded methods - workflow.

During the fitting process of some classifier or regression models, a value of importance is assigned to the input features. We can then use this importance to rank the variables and select the most important ones. The two main embedded methods for feature selection are the Lasso regularization, which is utilized when fitting linear models, and the information gain, which is utilized while training decision trees. In this chapter, we will cover these methods in more detail.

In this chapter, we'll discuss:

- Lasso regularization
- Decision trees and tree ensembles

7.1 Lasso regularization

Linear regression models aim to predict the outcome based on a linear combination of the predictor variables, given by:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + e$$

In the former equation, x_i represents the various predictor variables, with *i* ranging from 1 to *n*, β_i represent the coefficients that multiply those predictors, and *e* represents an error term.

The values of the coefficients are usually determined by minimizing the squared difference between the real and the predicted value of *y*, as follows:

$$minimize\{\sum_{j=1}^{N}(y_j - (\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n)_j)^2\}$$

Here, *j* represents each observation. This is called the least-square loss, and can be re-written as:

$$minimize\{\sum_{j=1}^{N}(y_j - \beta_0 - \sum_{i=1}^{n}\beta_i x_{ij})^2\}$$

where *i* refers to each variable, *n* is the total number of variables, *j* refers to each observation, and *N* is the total number of observations.

If *n* is large, that is, if the dataset has a lot of features, linear models are likely to overfit the data. To prevent this, the search for the optimal coefficients is done with regularization.

There are two main regularization procedures: the **Ridge** and the **Lasso** regularization. With the Lasso or *l1*-regularized regression, the coefficients are estimated by minimizing the following equation:

$$minimize\{\sum_{j=1}^{N}(y_j - \beta_0 - \sum_{i=1}^{n}\beta_i x_{ij})^2 + \lambda \sum_{i=1}^{n}|\beta_i|\}$$

This is the same least-square loss equation plus a constraint on the values of the coefficients given by $\lambda \sum_{i=1}^{n} |\beta_i|$, where λ is the strength of the regularization, or the strength of the constraint.

The Ridge, or *l2*-regularized regression, estimates the coefficients by minimizing:

$$minimize\{\sum_{j=1}^{N}(y_j - \beta_0 - \sum_{i=1}^{n}\beta_i x_{ij})^2 + \lambda \sum_{i=1}^{n}\beta_i^2\}$$

where the constraint on the coefficients is given by the sum of the squared values of β instead of their absolute value.

Both regularization procedures shrink the coefficients of the linear model to reduce bias, or in other words, to prevent overfitting. The greater the value of λ, the more the coefficients are shrunk.

It turns out that the Lasso regularization is special: it has the ability to set some of the coefficients to zero. In other words, Lasso returns sparse solutions and, as such, offers the intrinsic property of providing feature selection. If the coefficients of some features are 0, we can safely remove those features from the data, as they do not contribute to the prediction.

In the following image, we see the values of the coefficients for 15 features of the breast cancer dataset (*y-axis*), estimated by a Lasso regression with increasing constraints (λs), which in the plot are called penalties (*x-axis*). As the value of the penalty increases, more coefficients are set to zero (see the green, and then red and blue lines).

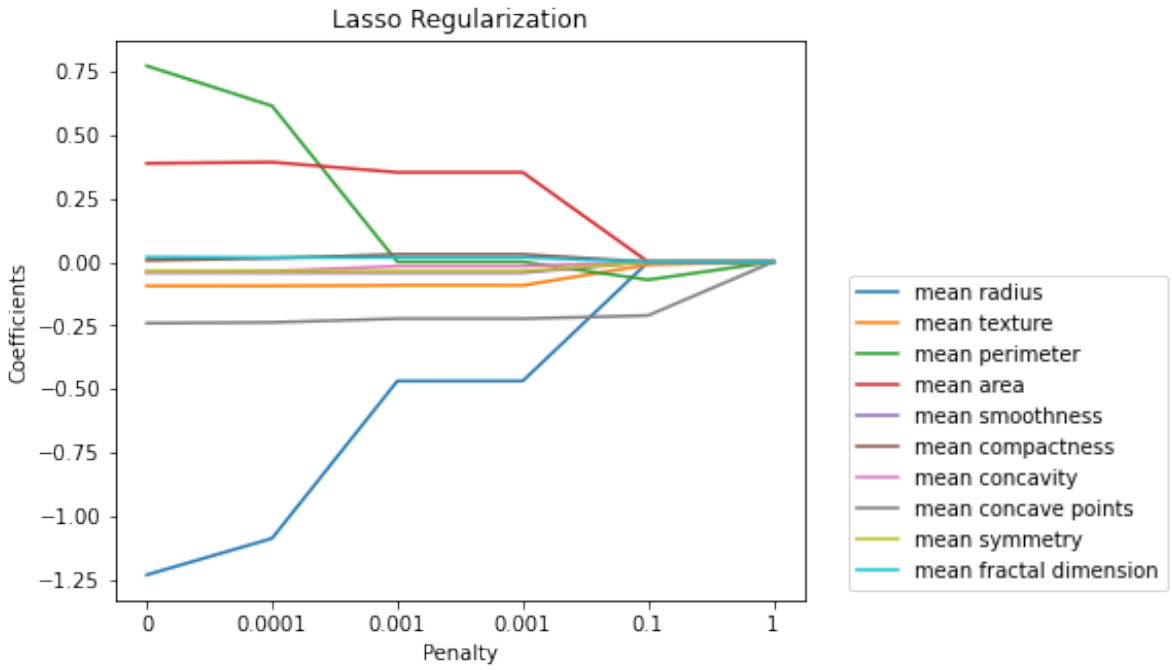


Figure 7.2: Lasso regularization shrinks coefficients to zero.

In contrast, the Ridge regularization does not have that property, or at least not until the penalty is very big, as can be witnessed in the following image:

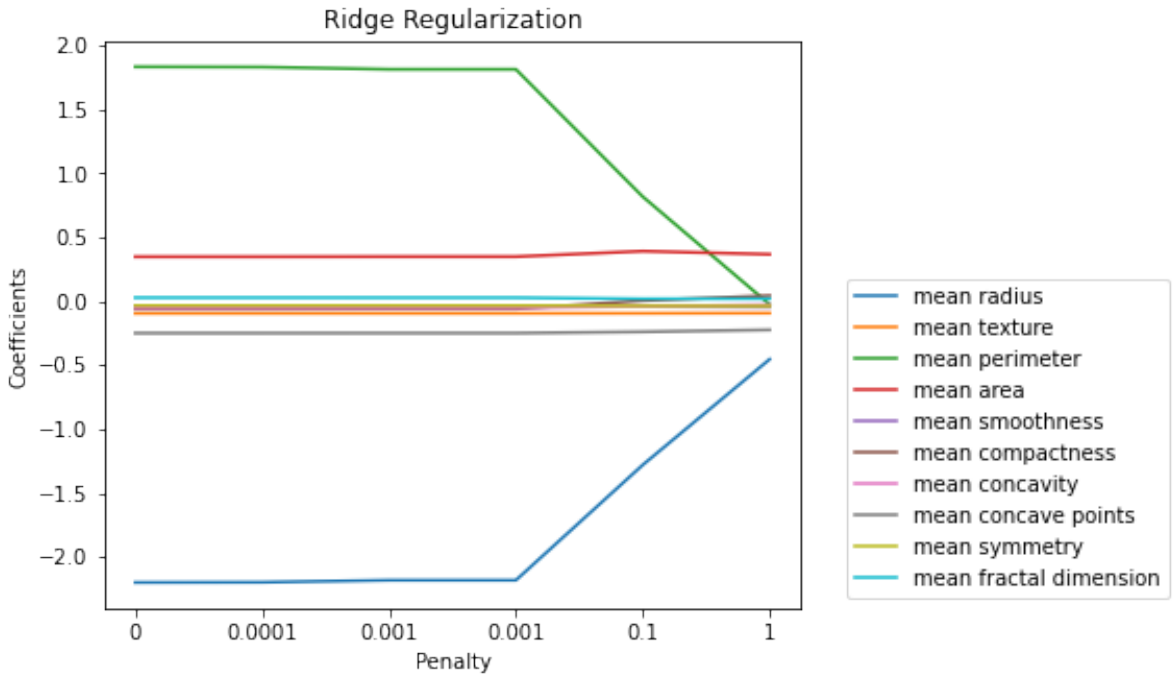


Figure 7.3: Ridge regularization does not shrink coefficients to zero.

A mathematical demonstration of how Lasso has the ability to shrink coefficients to zero exceeds the scope of the book. I invite you to find the details in the references at the end of this section.

It is important to highlight that the coefficients in a linear regression, regardless of the regularization procedure, are influenced by the magnitude of the variables. Thus, before carrying out feature selection we **should put the features in similar scales**.

Python implementation

In this section, we'll show how to select features using Lasso for logistic regression and then, for a linear regression model. Let's begin by importing the libraries, functions, and classes:

```
from sklearn.datasets import load_breast_cancer, fetch_california_housing
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import Lasso, LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Logistic regression

We'll start by using the breast cancer dataset with the aim of predicting if a tumor is benign or malignant. Let's load the data and separate it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

To select features with Lasso when training a logistic regression model from scikit-learn, we need to select `"l1"` as the penalty. We'll *wrap* the logistic regression model with Scikit-learn's `SelectFromModel` transformer, which will automatically remove the features whose coefficients are shrunk to 0:

```
selector = SelectFromModel(
    LogisticRegression(
        C=0.5,
        penalty='l1', # Select the Lasso (l1) penalty.
        solver='liblinear',
        random_state=10),
)
```

We mentioned previously that the coefficients of the linear models are influenced by the magnitude of the features. Therefore, we will standardize the features before carrying out the selection. To do this, we'll put together the standard scaler and the selection transformer within a pipeline:

```
pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("selector", selector),
])
```

With `fit()`, the pipeline will first standardize the data and then train a logistic regression. After that, it will identify the features whose coefficients are different from 0:

```
pipe.fit(X_train, y_train)
```

By executing `pipe.get_feature_names_out()` we display the name of the selected features:

```
array(['mean texture', 'mean concave points', 'mean symmetry',
       'radius error', 'compactness error', 'fractal dimension error',
       'worst radius', 'worst texture', 'worst perimeter', 'worst area',
       'worst smoothness', 'worst concavity', 'worst concave points',
       'worst symmetry'], dtype=object)
```

We can remove the features from the training and testing sets like this:

```
X_train_selected = pipe.transform(X_train)
X_test_selected = pipe.transform(X_test)
```

If we now execute `X_train_selected.shape, X_test_selected.shape`, we obtain the shapes of the reduced datasets: `((426, 14), (143, 14))`.

Go ahead and change the value of the penalty (C) to see if the result changes.

> **Tip**
>
> The value of C, and thus, the feature subset, can be treated as a hyperparameter and optimized as part of hyperparameter tuning with cross-validation.

Linear regression

Let's now select features for a regression dataset. Let's import the California housing dataset with the aim of predicting house prices, and separate it into a training and a testing set:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

To select features using Lasso, we need to wrap the Lasso model with the `SelectFromModel` transformer, which will select those features whose coefficients differ from 0:

```
sel_ = SelectFromModel(Lasso(alpha=0.01, random_state=10))
```

In linear models, the magnitude of the features may affect the selection process, so we'll scale the features before training the model. To do this, we'll put together the selection transformer and the standard scaler within a pipeline, and enforce its output to pandas dataframes:

```
pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("selector", sel_),
]).set_output(transform="pandas")
```

With `fit()`, the pipeline will first scale the data, then train the Lasso model, and finally select those features whose coefficients are different from 0:

```
pipe.fit(X_train, y_train)
```

We can obtain the name of the selected features by executing `pipe.get_feature_names_out()` (try it out), and we can reduce the datasets to the selected features as follows:

```
X_train_selected = pipe.transform(X_train)
X_test_selected = pipe.transform(X_test)
```

That's it, we have now selected features utilizing the ability of the Lasso regularization to shrink coefficients to zero.

7.2 Decision trees and tree ensembles

Decision tree algorithms predict an outcome by making sequential partitions of the data based on the values of the input features. A decision tree model consists of a series of nodes. In the first node, we find the entire dataset (see dots in the following image). Child nodes contain subsets of the data from the previous node.

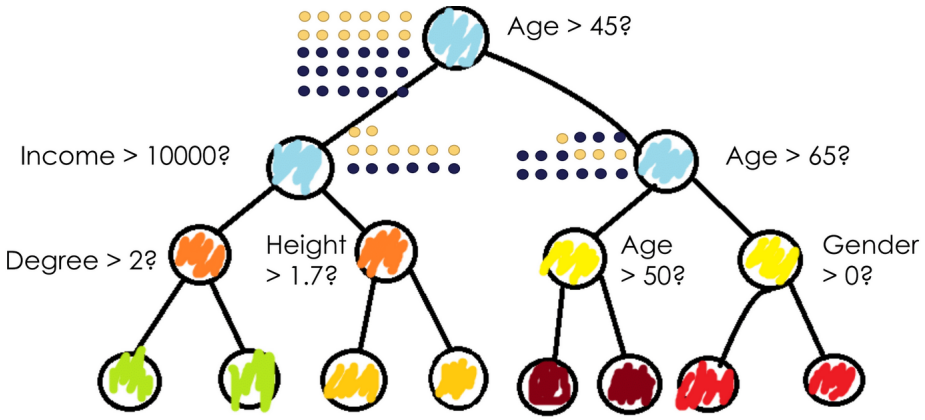


Figure 7.4: Diagram of a decision tree depicting the different nodes, the features and values used to create the partitions, and the total data and partitions in first and second nodes. Image taken from my Machine Learning Interpretability course.

At each node, a data partition occurs. To find the best partition, the algorithm tests every value of every feature and selects the feature (and the value) that generates the best partition.

There are different metrics that can be used to determine the "best" partition. In classification trees, the induction algorithm minimizes the Gini index, the entropy, or the misclassification error.

Gini ➔ $$H(X_m) = \sum_k p_{mk}(1 - p_{mk})$$

Entropy ➔ $$H(X_m) = -\sum_k p_{mk} \log(p_{mk})$$

Misclassification ➔ $$H(X_m) = 1 - \max(p_{mk})$$

Figure 7.5: Metrics used to evaluate partitions in classification trees. Taken from scikit-learn's documentation.

In regression trees, the induction algorithm minimizes the mean squared error, the mean absolute error, or the Poisson deviance.

Sum of squares ➔ $$H(Q_m) = \frac{1}{n_m} \sum_{y \in Q_m} (y - \bar{y}_m)^2$$

Poisson deviance ➔ $$H(Q_m) = \frac{1}{n_m} \sum_{y \in Q_m} (y \log \frac{y}{\bar{y}_m} - y + \bar{y}_m)$$

Mean absolute error ➔ $$H(Q_m) = \frac{1}{n_m} \sum_{y \in Q_m} |y - median(y)_m|$$

Figure 7.6: Metrics used to evaluate partitions in regression trees. Taken from scikit-learn's documentation.

The importance of each feature is calculated as the total reduction in the value of the metric used to evaluate the partition brought by that feature throughout the tree. For example, if a feature is used to partition data in node 1 and then again in node 3, the significance of that feature is determined by the sum of the metric reductions at both nodes.

For a given node *m* with *left* and *right* child nodes, the impurity reduction is calculated as:

$$Gain_m = impurity_m - (weight_{left} \times impurity_{left} + weight_{right} \times impurity_{right})$$

The value $Gain_m$ refers to the information gained (or impurity reduction) by partitioning the data at that specific node. The values of $impurity_{left}$ and $impurity_{right}$ are given by the metric of choice evaluated at each child. The weights are defined by the fraction of the samples in *m* that are present in the child nodes. Thus, if N is the number of observations,

$$weight_{left} = \frac{N_{left}}{N_m}$$

$$weight_{right} = \frac{N_{right}}{N_m}$$

The total impurity reduction attributed to a given feature in a tree is the sum of the impurity reductions resulting from splits on that feature at each node, divided by the total impurity reduction across the entire tree:

$$Importance_f = \frac{\sum_m Gain_m}{\sum_f \sum_m Gain_m}$$

In the previous equation, $Gain_m$ represents the reduction in impurity resulting from the feature in question at node *m*, and thus $\sum_m Gain_m$ is the total impurity reduction caused by that feature. $\sum_f \sum_m Gain_m$ represents the total impurity reduction in the tree, accounting for all features that split the data at various tree nodes.

Tree ensembles

Tree ensembles are collections of decision trees created through boosting or bagging. Random Forests are created by bagging decision trees, whereas gradient boosting machines are trained by boosting decision trees.

Random forests train multiple classification trees using bootstrapped data partitions and feature subsets. The final prediction in random forests results from the average of all trees' predictions.

Gradient boosting machines, on the other hand, build trees sequentially. Each tree minimizes the difference between the current tree's predictions and the residuals of the previous tree. The final prediction results from the weighted sum of the trees' outputs.

Regardless of the way in which the ensembles are created (that is, bagging or boosting), feature importance is calculated for each tree and then averaged:

$$Importance_f = \frac{1}{T} \sum Importance_f$$

Here, T is the number of trees and *f* is the feature in question.

Like this, ensembles based on decision trees can assign feature importance during induction of the model. We can then select the features with the highest importance.

> **Note**
>
> Gradient boosting machines train sequential trees that predict the difference between the current tree's predictions and the residuals of the previous trees. Hence, regardless of whether the trees are for classification or regression, the induction process minimizes a regression metric, in particular, the mean squared error.

Python implementation

In this demo, we'll select features from the breast cancer dataset using the importance derived from random forests. However, the process is identical if we want to derive importance from decision trees or gradient boosting machines; we just need to swap the models.

Let's begin by importing the libraries, functions, and classes:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import load_breast_cancer, fetch_california_housing
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split
```

Let's load the dataset and separate it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

To select features based on their importance derived from a random forest classifier (or any other decision tree or tree-ensemble from scikit-learn), we need to wrap the model with the `SelectFromModel` transformer:

> **Note**
>
> `SelectFromModel` will select features whose importance is greater than the average importance of all features. This can be modified through the parameter `threshold`, in which case, the transformer will select features whose importance is higher than the threshold.

```
sel_ = SelectFromModel(
    RandomForestClassifier(n_estimators=10, random_state=10))
sel_.fit(X_train, y_train)
```

> **Note**
>
> For regression models, the code implementation is the same, we just need to swap the `RandomForestClassifier` for the `RandomForestRegressor`.

Scikit-learn automatically selects features whose importance is greater than the mean importance of all features in the data. But we could as well change this cut-off point to some other arbitrary number.

By executing `sel_.get_support()` we obtain a boolean vector with `True` for the features that were selected, as shown in the following output:

```
array([ True, False, False,  True, False, False,  True,  True, False,
       False,  True, False, False, False, False, False, False, False,
       False, False, False, False,  True,  True, False, False, False,
       True, False, False])
```

We can make a list with the selected features as follows:

```
selected_feat = X_train.columns[(sel_.get_support())]
```

By executing `len(selected_feat)` we obtain the number of selected features: `8`. By executing `selected_feat` we obtain the names of the selected variables:

```
Index(['mean radius', 'mean area', 'mean concavity', 'mean concave points',
       'radius error', 'worst perimeter', 'worst area',
       'worst concave points'],
      dtype='object')
```

> **Note**
>
> A simpler way to get the name of the selected features is to execute `sel_.get_feature_names_out()`.

Let's make a bar plot with the feature importance:

```
pd.Series(
    sel_.estimator_.feature_importances_.ravel(),
    index=X_train.columns).plot.bar(figsize=(10,5),
    )
plt.ylabel('Feature importance')
plt.show()
```

In the following image, we see the importance assigned to each feature by random forests:

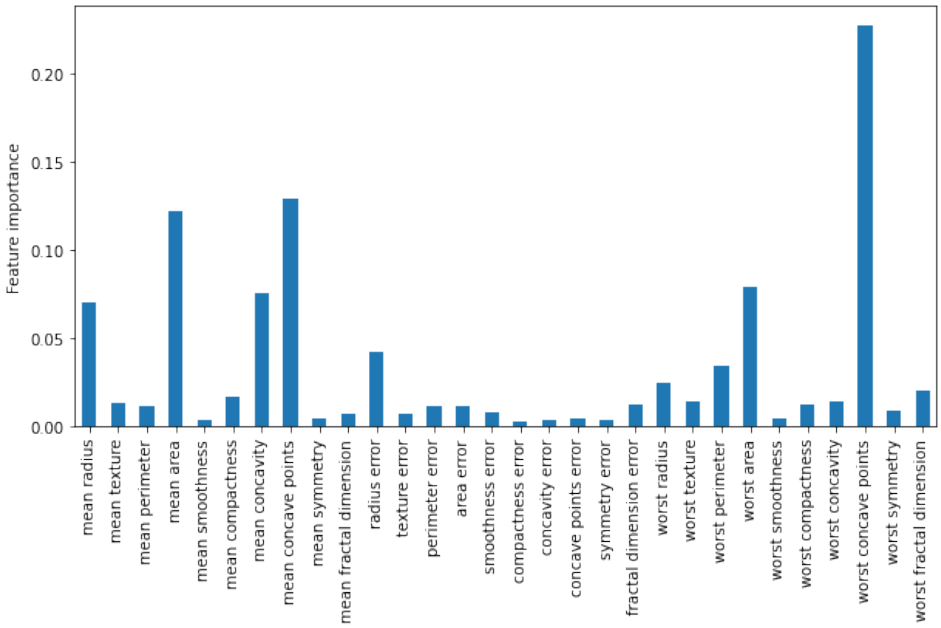


Figure 7.7: Feature importance derived from random forests.

Finally, we can reduce the datasets to the selected variables:

```
X_train_selected = sel_.transform(X_train)
X_test_selected = sel_.transform(X_test)
```

And that's it, we have now selected features based on the importance obtained from decision trees.

> **Note**
>
> In this demo, we derived the importance from random forests, however, the same code implementation can be used for gradient boosting machines and decision trees. If you want to select features from xgboost or lightGBMs, you can use their scikit-learn compatible API to select features as I show here.

7.3 References

Lasso regularization:

- Tibshirani R, Regression Shrinkage and Selection via the Lasso, J. R. Statistics Society, 58: 267-288, 1996.
- Hastie, Tibshirani, Wainwright, Statistical Learning with Sparsity, The Lasso and Generalizations, CRC Press, Taylor and Francis Group, 2015.
- For a mathematical demonstration of the Lasso property visit this link: https://bit.ly/3zWBO8L
- For a visual demonstration of the Lasso property visit this link: https://bit.ly/3QrWNXY

Decision tree importance:

- Breiman, et. al., Classification and Regression Trees, Wadsworth, Belmont, CA, 1984.
- Breiman, Random Forests, Machine Learning 45:5–32, 2001.

8 Recursive Feature Elimination

The feature importance derived from linear models or decision trees (embedded methods, see *Chapter 7*) can be influenced by the other features in the dataset. Some features may show less importance when it is derived using the entire dataset with respect to the importance they would show when evaluated individually.

In particular, in random forests, the feature importance is more or less evenly distributed among correlated features, hence, they show less relevance than what they would, were their correlated counterparts not present in the dataset. In fact, removing the correlated features can lead to an increase in the importance of (some of) the remaining features.

To mitigate the effect of correlation on the importance derived through embedded (and other) methods, and avoid selecting redundant features, we can remove features recursively, re-assessing the importance of the remaining variables at each round. This way, if feature importance was 'masked' by the presence of correlated features, when we remove them, the importance may increase, and we may get a better representation of the true influence of a feature towards the outcome or target variable. This process is called Recursive Feature Elimination (RFE).

In this chapter, we will discuss 2 implementations of RFE. In the first one, feature importance is derived through embedded methods. In the second implementation of RFE, feature importance is calculated as the drop in model performance resulting from removing the feature.

In summary, we will cover:

- Recursive feature elimination - embedded importance
- Recursive feature elimination - performance degradation

8.1 Recursive feature elimination - embedded importance

To better evaluate the importance of correlated (or otherwise related) features when using embedded methods (see *Chapter 7*), we can remove features recursively, and re-assess the importance of the remaining variables. The importance of features correlated with those removed should increase, providing a more accurate assessment of their true relationship with the target, while avoiding the selection of redundant variables.

> **Note**
>
> Recursive feature elimination requires training a model at each round, so it is more computationally expensive than embedded methods.

This implementation of Recursive Feature Elimination (RFE) proceeds as follows:

1. Train a machine learning model from which we can infer feature importance.
2. Derive feature importance and rank the features.
3. Remove the feature (or features) with the least importance.
4. Repeat 1 to 3 until a stopping criteria is met.

This implementation of RFE stops after selecting a pre-defined number of features (*step 4*).

> **Note**
>
> To accelerate the feature selection procedure, we can remove several features at each iteration (*step 3*). The trade-off is the quality of the resulting feature subset.

RFE has the advantage of accounting for changes in importance after a feature (or a small subset of features) is removed. As a result, it is better suited for handling highly correlated datasets. However, because it trains multiple classifiers or regressors, it is more computationally expensive than embedded methods.

Python implementation

RFE based on embedded methods is supported by scikit-learn. Let's import the functions and classes:

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE
from sklearn.model_selection import train_test_split
```

Let's load the breast cancer dataset and separate it into a training and testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's carry out RFE removing 2 features at each iteration and stopping the search when there are 8 features remaining in the dataset:

```
sel_ = RFE(
    RandomForestClassifier(n_estimators=10, random_state=10),
    n_features_to_select=8,
    step=2,
).set_output(transform="pandas")
```

> **Note**
>
> Scikit-learn's implementation of RFE works with models that can attribute feature importance during training, that is, linear models and decision tree based models.

With `fit()` we trigger the feature selection process:

```
sel_.fit(X_train, y_train)
```

Let's capture the names of the selected features in an array:

```
selected_feat = X_train.columns[(sel_.get_support())]
```

By executing `selected_feat` we display the name of the selected variables:

```
Index(['mean area', 'mean concave points', 'area error', 'worst texture',
       'worst perimeter', 'worst area', 'worst compactness',
       'worst concave points'],
      dtype='object')
```

> **Note**
>
> A better way of displaying the name of the selected features is by executing `sel_.get_feature_names_out()`.

We can remove the features from the training and testing sets like this:

```
X_train_selected = sel_.transform(X_train)
X_test_selected = sel_.transform(X_test)
```

And that's it. We have selected features recursively based on the importance derived from random forests.

> **Note**
>
> To select features recursively for other machine learning models, simply swap the random forests for a decision tree, linear model or ensemble of your choice.

8.2 Recursive feature elimination - performance degradation

Recursive Feature Elimination (RFE) is a feature selection technique that starts by training a model using the entire dataset, and then, iteratively removes the least relevant feature(s). The relevance of the features can be assessed in various ways. Here, we'll determine feature relevance as the decrease in model performance resulting from removing the feature.

> **Note**
>
> This implementation of RFE provides a faster alternative to *backward feature selection*, which we discussed in (*Chapter 6*). Backward selection removes every feature at each round to determine the best subset. RFE, on the other hand, first ranks features based on importance, and then removes only **one** feature per iteration, the one with the least importance. As a result, RFE trains less machine learning models compared to backward search, and is therefore faster.

This implementation of RFE consists of the following steps:

1. Train a machine learning model using all of the features.
2. Rank the features according to their importance derived from the model in 1.
3. Discard the least important feature and train a new machine learning model.
4. Calculate the change in model performance resulting from removing the feature.
5. If the performance decreases beyond a threshold, keep the feature from 3, otherwise, remove it.
6. Repeat steps 3-5 until all the features have been evaluated.

> **Note**
>
> This implementation of recursive feature elimination differs from backward elimination in that features are assessed and, potentially removed, in a sequence. The sequence is determined by the feature importance, which is derived from the machine learning model. Thus, this search algorithm trains fewer machine learning models.

The threshold for model performance degradation (*step 5*) is key in deciding whether a feature is retained or removed, and it is determined by the user. An optimal threshold is sometimes hard to assess and will directly influence the subset of selected features.

> **Note**
>
> Note that this implementation of RFE evaluates all features and therefore does not require a stopping criteria.

Python implementation

RFE based on model performance is supported by Feature-engine. In this demo, we will use RFE to select features that predict house price using the california housing dataset. We'll first implement RFE using the open-source Python library Feature-engine, and then we'll implement the process manually, to better understand the logic of this selection process.

> **Note**
>
> The procedure for applying RFE to a classification model is very similar to what we show in this demo. We just need to replace the machine learning model with a classifier and choose a suitable performance metric.

Let's import the libraries, functions, and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.base import clone
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import GradientBoostingRegressor
```

```
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split

from feature_engine.selection import RecursiveFeatureElimination
```

Let's load the California housing dataset and separate it into a training and a testing set:

```
X, y = fetch_california_housing(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's set up a gradient boosting machine for regression. We will use it to evaluate the features during the search.

```
model = GradientBoostingRegressor(
    n_estimators=5,
    random_state=10,
)
```

Feature-engine

Let's set up a RFE search to select features for the previous gradient boosting machine. We'll use the R^2 to evaluate the drop in model performance resulting from removing each feature, using 2-fold cross-validation. We will remove those features that result in a decrease in R^2 greater than 0.001.

```
sel = RecursiveFeatureElimination(
    estimator = model,
    scoring = 'r2',
    threshold = 0.001,
    cv=2,
)
```

With `fit()`, we trigger the search:

```
sel.fit(X_train, y_train)
```

`RecursiveFeatureElimination()` saves a few parameters learned during the search. By executing `sel.initial_model_performance_`, we obtain the performance of the gradient boosting machine trained using the entire dataset: `0.3639885983830904`.

`RecursiveFeatureElimination()` also stores the importance of the features derived from the gradient boosting machine trained using all features:

```
r = pd.concat([
    pd.Series(sel.feature_importances_),
    pd.Series(sel.feature_importances_std_)
], axis=1
)
r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True, figsize=(15,5))

plt.title("Feature importance - derived from the GBM")
plt.ylabel('Feature importance')
plt.xlabel('Features')
plt.show()
```

In the following image, we can see the mean feature importance derived from the GBM and its standard deviation:

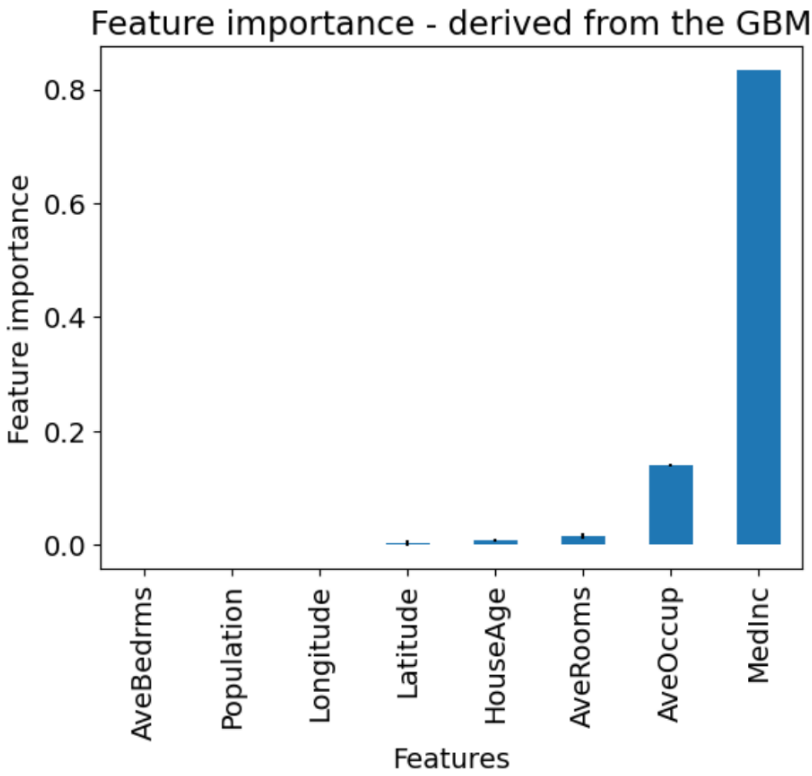


Figure 8.1: Feature importance derived from the gradient boosting machine.

We can also plot the changes in R^2 that resulted from the elimination of each feature:

```
r = pd.concat([
    pd.Series(sel.performance_drifts_),
    pd.Series(sel.performance_drifts_std_)
], axis=1
)
r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True, figsize=(10,5))

plt.title("Performance change after removing each feature")
plt.ylabel('Mean performance change')
plt.xlabel('Features')
plt.show()
```

In the following plot, we see the change in the mean R^2 and its standard deviation resulting from feature elimination:

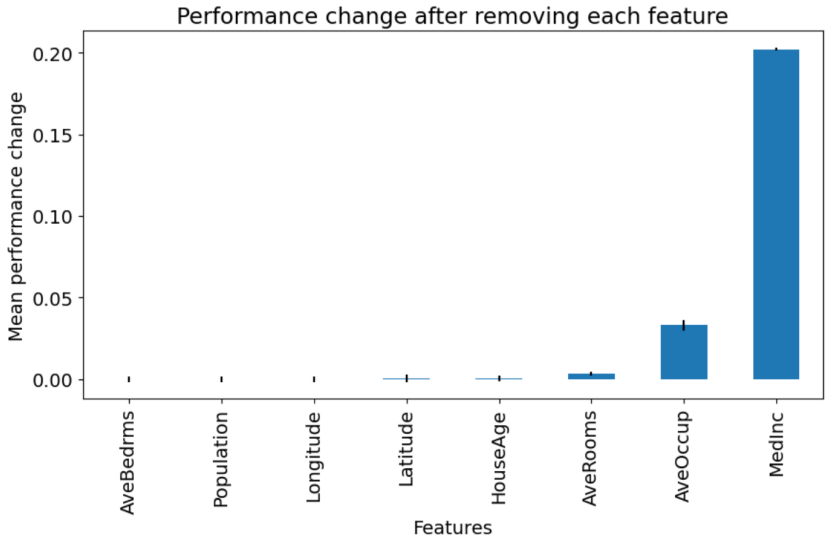


Figure 8.2: Feature importance calculated as the drop in performance resulting from removing the feature: mean and standard deviation.

By executing `sel.features_to_drop_` we obtain the features that will be removed from the data: `['HouseAge', 'AveBedrms', 'Population', 'Latitude', 'Longitude']`.

Let's now reduce the dataset to the selected variables:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

And that's it! We now reduced the training and testing sets to the selected variables.

> **Tip**
>
> In theory, this RFE implementation is suitable to select features for all models, because feature importance is estimated as the drop in model performance and does not depend on an embedded method. However, in Feature-engine's implementation, features are initially ranked based on their embedded importance, making this implementation dependent on models that support embedded methods. This may change in the future, so check out Feature-engine's documentation regularly.

Manual implementation

To understand the logic behind RFE, let's implement a simpler search manually, without cross-validation. First, we train a regression model using the entire dataset, and calculate its performance. Note that we fit the model to the train set, and calculate the performance on the test set:

```
model.fit(X_train, y_train)

y_pred_test = model.predict(X_test)
r2_full = r2_score(y_test, y_pred_test)

print(f'R2 all features = {r2_full}')
```

The previous code block returns the following print statement:

```
R2 all features = 0.3564875693230235
```

Next, let's sort the features based on the performance derived from the regression model, the least important feature first:

```
features = pd.Series(model.feature_importances_,
                     index=X_train.columns).sort_values(ascending=True)

features.head()
```

In the following output, we see the 5 features with the least importance:

```
AveBedrms     0.000000
Population    0.000000
Latitude      0.000000
Longitude     0.000000
HouseAge      0.009201
dtype: float64
```

Next, we make a list with the ranked features:

```
features = list(features.index)
```

Finally, we eliminate each feature recursively, based on their importance, and we train a new regression model. If the performance of the new model is worse than the previous one (beyond a threshold), we keep the feature, otherwise, we remove it:

```
# The threshold for model performance drop.
tol = 0.001

# To store the features to remove.
features_to_remove = []

# Loop over every feature in order of importance.
for feature in features:

    # Initialize model.
    model_int = clone(model)

    # Fit model with all variables except the removed features.
    model_int.fit(
        X_train.drop(features_to_remove + [feature], axis=1), y_train)

    # Obtain prediction using the test set.
    y_pred_test = model_int.predict(
        X_test.drop(features_to_remove + [feature], axis=1))

    # Calculate r2.
    r2_int = r2_score(y_test, y_pred_test)

    # Determine drop in r2.
    diff_r2 = r2_full - r2_int

    # Assess r2.
    if diff_r2 < tol:
        # If we remove the feature, we need to set the new r2
        # to the one based on the remaining features.
        r2_full = r2_int

        # Append the feature to remove to the list.
        features_to_remove.append(feature)

print(f'Number of features to remove: {len(features_to_remove)}')
```

In the following output, we see that 4 features should be removed based on the search:

```
Number of features to remove: 4
```

Let's make a list with the features to keep:

```
features_to_keep = [x for x in features if x not in features_to_remove]
```

Finally let's assess the performance of a regression model trained on the selected features:

```
model_final = clone(model)

# Fit the model with the selected features.
model_final.fit(X_train[features_to_keep], y_train)

# Obtain predictions
y_pred_test = model_final.predict(X_test[features_to_keep])

# Calculate r2.
r2_final = r2_score(y_test, y_pred_test)
print(f'r2 selected features= {r2_final}')
```

In the following output we see that the R^2 returned by the model trained on fewer features:

```
r2 selected features= 0.3564875693230233
```

The performance of this model is very similar to the performance of the model trained utilizing the entire dataset, yet the model is easier to interpret, faster and requires less storage, because it requires less features to produce the predictions.

8.3 References

- Guyon, et. al., Gene selection for cancer classification using support vector machines, Mach. Learn., 46, 389–422, 2002.

9 Alternative Feature Selection Methods

Throughout the book, we discussed classical feature selection methods that generally fall into one of the filter, wrapper, or embedded categories. There are, however, additional features selection procedures that can't be easily classified in any of these groups. Some of these methods combine aspects of embedded and wrapper methods, and are thus faster than wrapper methods and potentially better than embedded methods at selecting features. An example of this is recursive feature addition. Other feature selection techniques assess feature importance by measuring the drop in model performance after randomly permuting the feature values, or select features comparing their importance to that of random variables. We'll cover these methods in this chapter.

In this chapter, we will discuss the following feature selection methods:

- Recursive feature addition
- Feature permutation
- Probe features selection
- Minimum Redundancy Maximum Relevance (MRMR)
- Boruta

9.1 Recursive feature addition

Recursive Feature Addition (RFA) starts with an empty set of features and iteratively adds features so long there is an increase in model performance. It is similar to forward search (see *Chapter 6*), but instead of testing all features at each iteration before deciding which one to add, RFA first ranks features based on importance, and then tests the features following their importance rank. Like this, RFA is faster than forward search, because it trains less models.

In RFA, we first train a model using all features and then rank the features based on their model derived feature importance (see embedded methods in *Chapter 7*). After that, we train a machine learning model using the most important feature and determine its performance using a metric like the ROC-AUC or the mean squared error. In subsequent rounds, we add the remaining features, one at a time, in order of importance, re-train the model and determine if its performance increased. Those features that resulted in an increase in model performance are selected.

In summary, RFA consists of the following steps:

1. Train a machine learning model using all features and derive their importance.
2. Rank the features by importance.
3. Train a model using the most important feature and determine its performance.
4. Add the next most important feature and re-train the model.
5. If adding the feature results in an increase in model performance keep the feature.
6. Repeat steps 4–5 until all features have been evaluated.

> **Note**
>
> The model from *step 1* should be a model from which we can infer feature importance, like linear or logistic regression, or tree based models, so that we can rank the features. The model from *steps 3* and *4*, however, should be the model for which we want to select the features.

In *step 5*, features are selected if they result in an increase in model performance beyond a threshold that is arbitrarily set by the user. This threshold is key in deciding whether a feature is retained or removed. Bigger thresholds result in a more stringent selection.

Python implementation

We will apply RFA to select features that predict malignant tumours using the breast cancer dataset. We'll start by using the open-source Python library Feature-engine, which has `RecursiveFeatureAddition` that applies RFA as we discussed in the previous section. After that, we'll apply RFA manually, using pandas and scikit-learn, to better understand the logic of this selection process.

> **Note**
>
> In this demo, we apply RFA to select features for a classification model. To select features with RFA for a regression model we just need to replace the classifier with a regression model and choose a suitable performance metric, like the mean squared error or R^2.

Let's import the libraries, functions, and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.base import clone
from sklearn.datasets import load_breast_cancer
```

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

from feature_engine.selection import RecursiveFeatureAddition
```

Let's load the breast cancer dataset and separate it into training and testing sets:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's set up the classifier; we will use a gradient boosting machine:

```
model = GradientBoostingClassifier(
    n_estimators=5,
    random_state=10,
)
```

Feature-engine

Let's set up the RFA algorithm to identify the best features for the gradient boosting machine. We'll examine model performance using the ROC-AUC with 2-fold cross-validation and retain features that result in an increase in ROC-AUC greater than 0.0001.

> **Note**
>
> With Feature-engine, we use the same model to derive feature importance (*step1*) and then select the features (*steps 3* and *4*). If we want to use different models, we need to implement RFA manually.

```
rfa = RecursiveFeatureAddition(
    estimator=model,
    scoring='roc_auc',
    threshold=0.0001,
    cv=2,
)
```

The search begins when we call `fit()`:

```
rfa.fit(X_train, y_train)
```

`RecursiveFeatureAddition()` stores various parameters learned during the feature search. For example, by executing `rfa.initial_model_performance_`, we obtain the performance of the classifier trained utilizing all the features in the data: `0.9676361015186747`.

`RecursiveFeatureAddition()` also stores the feature importance (mean and standard deviation) derived from the classifier. Let's make a bar plot with these values:

```
r = pd.concat([
    pd.Series(rfa.feature_importances_),
    pd.Series(rfa.feature_importances_std_)
], axis=1)

r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True, figsize=(15,5))

plt.title("Feature importance - derived from the GBM")
plt.ylabel('Feature importance')
plt.xlabel('Features')
plt.show()
```

In the following figure, we can see the mean feature importance and standard deviation obtained with cross-validation:

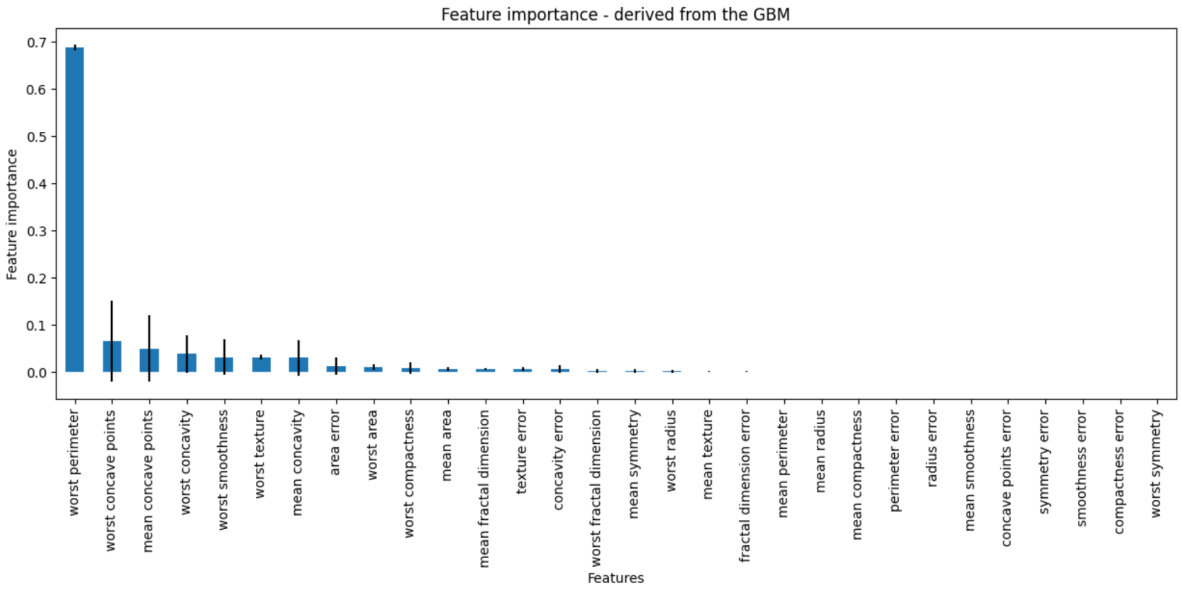


Figure 9.1: Feature importance derived from the gradient boosting machine trained using all features: mean and standard deviation.

RFA selects those features that resulted in an increase in model performance. We can plot the changes in model performance resulting from adding each feature as follows:

```
r = pd.concat([
    pd.Series(rfa.performance_drifts_),
    pd.Series(rfa.performance_drifts_std_)
], axis=1)

r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True, figsize=(15,5))

plt.title("Performance change caused by feature")
plt.ylabel('Mean performance change')
plt.xlabel('Features')
plt.show()
```

In the following image, we see that only 2 of the features resulted in an increase in the ROC-AUC of the model:

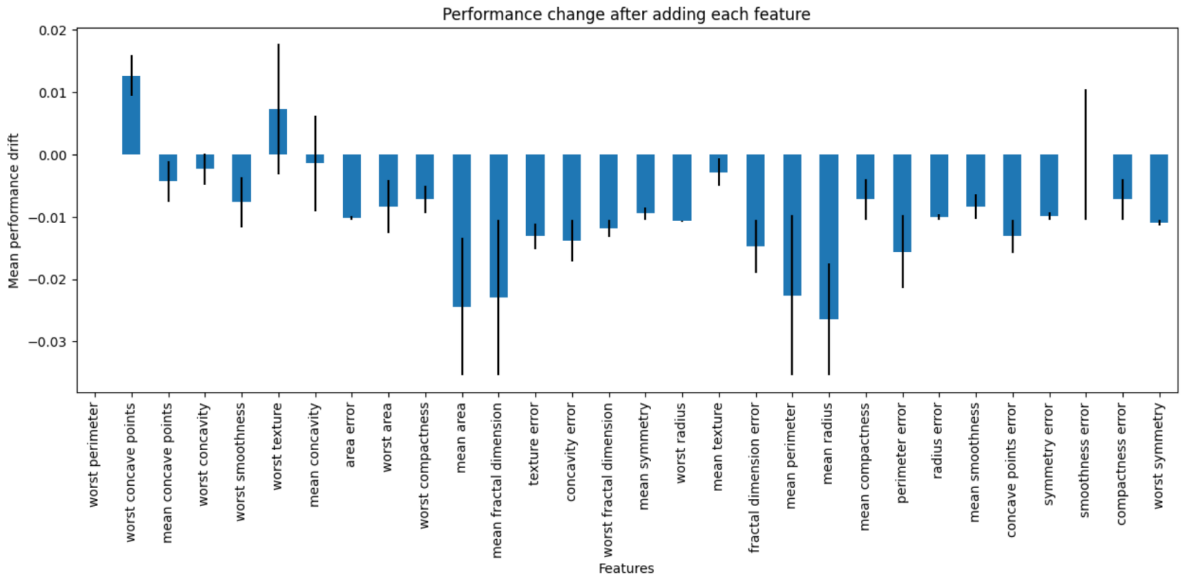


Figure 9.2: Change in the performance of the gradient boosting machine resulting from adding each feature to the dataset.

As a result, RFA selected 3 features: the most important variable identified by the importance derived from the GBM, and the 2 additional features that resulted in an increase in model performance, when added to the subset.

By executing `rfa.features_to_drop_`, we obtain the features that will be removed from the data:

```
['mean radius',
 'mean texture',
 'mean perimeter',
 'mean area',
 'mean smoothness',
 'mean compactness',
 'mean concavity',
 'mean concave points',
 'mean symmetry',
 'mean fractal dimension',
 'radius error',
 'texture error',
 'perimeter error',
 'area error',
 'smoothness error',
```

```
 'compactness error',
 'concavity error',
 'concave points error',
 'symmetry error',
 'fractal dimension error',
 'worst radius',
 'worst area',
 'worst smoothness',
 'worst compactness',
 'worst concavity',
 'worst symmetry',
 'worst fractal dimension']
```

We can now reduce the data to the selected features:

```
X_train_t = rfa.transform(X_train)
X_test_t = rfa.transform(X_test)
```

We can display the resulting dataframe by executing `X_test_t.head()`:

	worst texture	worst perimeter	worst concave points
512	29.66	113.30	0.20510
457	34.23	91.29	0.06005
439	19.31	96.53	0.08216
298	25.26	105.80	0.07530
37	22.81	84.46	0.05013

Figure 9.3: Dataframe with the selected features.

Let's now train a gradient boosting machine using only the selected features and determine its performance:

```
model.fit(X_train_t, y_train)
roc_auc_score(y_test, model.predict_proba(X_test_t)[:, 1])
```

The previous code outputs a value of ROC-AUC of `0.9922431865828093` which is even better than that returned by the model trained using the entire dataset. Like this, the model with the selected features has a similar performance to that with all features, but it is easier to interpret, faster to execute, and easier to maintain.

Manual implementation of RFA

To understand the logic behind the RFA algorithm, let's implement a simpler version manually. First, we'll fit a model using all the features in the data, so that we can derive feature importance and rank the features. Next, we will add one feature at a time and determine its impact on model performance utilizing the ROC-AUC.

Let's set up the classifier; we will use a gradient boosting machine:

```
model = GradientBoostingClassifier(
    n_estimators=5,
    random_state=10,
)
```

Now, we fit the model using all the features and determine its performance:

```
model.fit(X_train, y_train)

y_pred_test = model.predict_proba(X_test)[:, 1]
roc_full = roc_auc_score(y_test, y_pred_test)

print(f'ROC AUC full dataset = {roc_full}')
```

The output of the previous code is:

```
ROC AUC full dataset = 0.9731656184486374
```

Next, we sort the features according to their importance in decreasing order:

```
features = pd.Series(
    model.feature_importances_,
    index=X_train.columns).sort_values(ascending=False)

features.head()
```

We can see the 5 most important features in the following output:

```
mean concave points    0.754473
worst area             0.049151
worst radius           0.044597
worst texture          0.035570
worst perimeter        0.029786
dtype: float64
```

We capture the ordered features in a list:

```
features = list(features.index)
```

Next, we train a single variable classifier using the most important feature and store its performance. Note that we fit the model to the training set and obtain the performance using the test set:

```
model_one_feature = clone(model)
model_one_feature.fit(X_train[features[0]].to_frame(), y_train)

y_pred_test = model_one_feature.predict_proba(
    X_test[features[0]].to_frame())[:, 1]

roc_first = roc_auc_score(y_test, y_pred_test)
print(f'ROC AUC one feature = {roc_first}')
```

The output of the previous code block is the following:

```
ROC AUC one feature = 0.9548218029350105
```

Next, we will add one feature at a time, following their importance, and train a new machine learning model. We will compare the performance of the new model with that of the model without the feature. If the performance increases beyond 0.0001, we will keep the feature.

```
tol = 0.0001

# List to store the features we should keep.
features_to_keep = [features[0]]

# Loop over all the features following their importance.
for feature in features[1:]:

    # Initialize model.
    model_int = clone(model)

    # Fit model after adding the feature to test.
    model_int.fit(
        X_train[features_to_keep + [feature] ], y_train)

    # Obtain prediction using the test set.
```

```
    y_pred_test = model_int.predict_proba(
        X_test[features_to_keep + [feature] ])[:, 1]

    # Determine model performance with the roc-auc.
    roc_int = roc_auc_score(y_test, y_pred_test)

    # Determine the increase in the roc-auc.
    diff_roc = roc_int - roc_first

    # Assess the change in the roc-auc.
    if diff_roc >= tol:
        # if the change in the roc-auc value is higher than the threshold,
        # keep the feature and re-adjust the roc-auc to the new value.
        roc_first = roc_int

        # Append the feature to keep to the list.
        features_to_keep.append(feature)

print(f'number of features to keep: {len(features_to_keep)}')
```

The previous code block returns the following print statement as output, indicating that the method selected 5 features:

```
number of features to keep: 5
```

Finally, let's determine the performance of a model trained on the selected features:

```
model_final = clone(model)

# Fit the model with the selected features.
model_final.fit(X_train[features_to_keep], y_train)

# Obtain predictions.
y_pred_test = model_final.predict_proba(X_test[features_to_keep])[:, 1]

# Calculate roc-auc.
roc_final = roc_auc_score(y_test, y_pred_test)
print(f'ROC AUC selected features = {roc_final}')
```

The previous code block returns the following roc-auc value:

```
ROC AUC selected features = 0.9926624737945493
```

The ROC-AUC of the model with 5 features is not very different from the model trained using all the features. Thus, with just 5 features, we obtain similar performance as with the entire dataset, yet, the model trained using 5 features is faster and easier to interpret.

9.2 Feature permutation

A popular method to evaluate and select features consists of randomly shuffling (permuting) the values of a variable and determining how that permutation affects the performance of the machine learning algorithm. If the feature is important, randomly shuffling its values before obtaining a prediction should cause a noticeable decrease in model performance. If the feature is irrelevant, permuting its values should have no effect.

The procedure works as follows:

1. Train a machine learning model using all features and determine its performance.
2. Shuffle one feature and make a new prediction using the model from *step 1*.
3. Compare the performance of the prediction from *step 2* with the one from *step 1*.
4. If the performance drops beyond a threshold, keep the feature, otherwise remove it.
5. Repeat steps 2-4 until all features have been shuffled.

One of the advantages of this method is that it does not rely on feature importance derived from a machine learning classifier or regression model. Thus, this procedure can be applied to select features for all machine learning models, including those for which no feature importance can be derived, like, for example, nearest neighbours.

> **Note**
>
> Assessing the importance of features by permutation was introduced by Leo Breinman in his article Random Forests (2001), as a way to explore the mechanisms of the algorithm.

A second advantage of using permutation to select features is that it is fast, because we train only 1 model, and then use this model to obtain predictions after shuffling features.

A criticism that is given to this method, however, is that by randomly permuting the values of a feature, we may obtain feature combinations that would not exist in a real dataset. For example, if one variable is *temperature*, and the other one *country*, by randomly shuffling the values of *country*, we could obtain subzero temperatures in tropical countries, or over 40 in the poles, which is becoming increasingly common with climate change, but, you get the gist.

Python implementation

In this demo, we will select features by random permutation using the open-source library Feature-engine. After that, we'll apply the selection procedure manually to better understand how it works.

Let's begin by importing the libraries, functions and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.base import clone
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

from feature_engine.selection import SelectByShuffling
```

Let's load the breast cancer dataset and separate it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Let's set up the classifier for which we want to select features:

```
rf = RandomForestClassifier(
    n_estimators=10,
    random_state=2909,
)
```

Feature-engine

To select features by permutation, we use `SelectByShuffling` from Feature-engine. Let's set up the transformer to train the previous random forests and then select features by permuting their values and evaluating the model performance using the ROC-AUC with 3-fold cross-validation:

```
sel = SelectByShuffling(
    estimator=rf,
    scoring='roc_auc',
    threshold=0,
    cv=3,
    random_state=1,
)
```

With `fit()` we trigger the training of random forests and the feature selection process:

```
sel.fit(X_train, y_train)
```

`SelectByShuffling()` stores a few parameters learned during the search. By executing `sel.initial_model_performance_` we obtain the performance of the classifier fit to the entire dataset: `0.9822274044237158`.

We can also plot the mean change in performance resulting from the permutation of each feature:

```
r = pd.concat([
    pd.Series(sel.performance_drifts_),
    pd.Series(sel.performance_drifts_std_)
], axis=1
)
r.columns = ['mean', 'std']

r['mean'].plot.bar(yerr=[r['std'], r['std']], subplots=True, figsize=(15,5))

plt.title("Performance change caused by feature permutation")
plt.ylabel('Mean performance change')
plt.xlabel('Features')
plt.show()
```

In the following plot, we see the changes in model performance resulting from shuffling each variable:

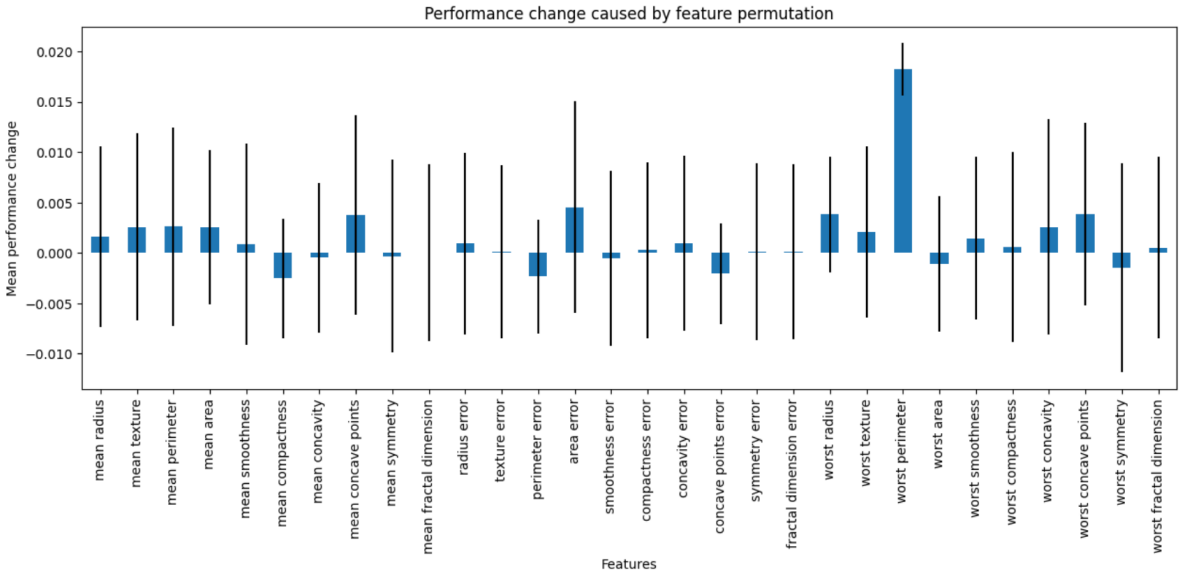


Figure 9.4: Performance change resulting from feature permutation.

> **Note**
>
> `SelectByShuffling()` determines the performance change by comparing the performance after shuffling with the performance before shuffling: `performance drift = model performance - performance after feature permutation`. If the feature is important, the `performance after feature permutation` will be low, and the drift will be high (and positive). If the feature is redundant, then `performance after feature permutation` should be roughly that obtained without permutation, and the drift close to 0.

In the previous image, we see that only 1 feature seems to be important: `worst perimeter`, since it is the only one whose permutation triggered model performance degradation. Shuffling all other features resulted in a performance change close to 0 (which is why the dispersion seems high, look at the scale on *y-axis*).

By executing `sel.features_to_drop_`, we obtain the variables that will be removed:

```
['mean smoothness',
 'mean compactness',
 'mean concavity',
 'mean symmetry',
 'mean fractal dimension',
```

```
 'radius error',
 'texture error',
 'perimeter error',
 'smoothness error',
 'compactness error',
 'concavity error',
 'concave points error',
 'symmetry error',
 'fractal dimension error',
 'worst area',
 'worst compactness',
 'worst symmetry',
 'worst fractal dimension']
```

Next, let's reduce the data to the selected variables:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

Manual implementation

To understand the logic behind feature selection by random permutation, let's implement it manually. First, we train a random forest using the entire dataset and obtain its performance:

```
rf.fit(X_train, y_train)

print('train auc score: ',
      roc_auc_score(y_train, (rf.predict_proba(X_train.fillna(0)))[:, 1]))
print('test auc score: ',
      roc_auc_score(y_test, (rf.predict_proba(X_test.fillna(0)))[:, 1]))
```

In the following output, we can see the ROC-AUC scores for training and testing sets:

```
train auc score:  0.9998822226933315
test auc score:  0.9908805031446541
```

Next, we shuffle a feature and obtain the model's prediction, comparing its performance to that of the model before shuffling. If the performance decreases beyond a threshold, we retain the feature:

```
# ROC-AUC of the model before shuffling features.
train_roc = roc_auc_score(y_train, (rf.predict_proba(X_train))[:, 1])

# List to capture the performance drift.
performance_shift = []

# Evaluate every feature.
for feature in X_train.columns:

    X_train_c = X_train.copy()

    # Shuffle feature.
    shuffle = X_train_c[feature].sample(frac=1, random_state=10)
    shuffle.index = X_train_c.index
    X_train_c[feature] = shuffle

    # Obtain prediction with shuffled feature and calculate roc-auc.
    shuff_roc = roc_auc_score(y_train, rf.predict_proba(X_train_c)[:, 1])

    # Calculate performance change.
    drift = train_roc - shuff_roc

    # Store the drop in roc-auc.
    performance_shift.append(drift)
```

By executing `performance_shift`, we can see the change in the ROC-AUC caused by the permutation of each feature:

```
[0.0,
 0.0002826655360045338,
 -0.00010599957600176957,
 7.06663840011057e-05,
 2.35554613337019e-05,
 1.177773066685095e-05,
 0.0002591100746708319,
 0.0017313264080277557,
 1.177773066685095e-05,
 0.0,
 5.888865333425475e-05,
 1.177773066685095e-05,
 0.00020022142133657717,
 7.06663840011057e-05,
```

```
 -4.71109226674038e-05,
 2.35554613337019e-05,
 0.00010599957600165855,
 4.71109226674038e-05,
 1.177773066685095e-05,
 4.71109226674038e-05,
 0.00012955503733547147,
 0.0004004428426730433,
 0.007290415282783402,
 1.177773066685095e-05,
 0.00012955503733547147,
 0.0,
 0.0005299978800085148,
 0.0004004428426730433,
 7.06663840011057e-05,
 0.0]
```

Let's transform the list into a pandas Series and sort the values in descending order:

```
feature_importance = pd.Series(
    performance_shift, index=X_train.columns).sort_values(ascending=False)

feature_importance.head()
```

Below we see the features that caused the greatest decrease in performance:

```
worst perimeter         0.007290
mean concave points     0.001731
worst concavity         0.000530
worst concave points    0.000400
worst texture           0.000400
dtype: float64
```

We can determine the number of features that caused a drop in performance by executing `feature_importance[feature_importance>0].shape[0]`. We will see that `24` features caused a performance decrease.

We can now print the name of those features:

```
feature_importance[feature_importance>0].index
```

Below we see the names of the important variables:

```
Index(['worst perimeter', 'mean concave points', 'worst concavity',
       'worst concave points', 'worst texture', 'mean texture',
       'mean concavity', 'perimeter error', 'worst radius', 'worst smoothness',
       'concavity error', 'area error', 'mean area', 'worst symmetry',
       'radius error', 'concave points error', 'fractal dimension error',
       'mean smoothness', 'compactness error', 'texture error',
       'mean symmetry', 'mean compactness', 'symmetry error', 'worst area'],
      dtype='object')
```

Finally, let's capture the predictive features in a list and train a new model with them:

```
# Selected features.
selected_features = feature_importance[feature_importance > 0].index

# Train random forests using the selected features.
rf = clone(rf)
rf.fit(X_train[selected_features], y_train)

# Print roc-auc from training and testing sets.
print(
    'train auc score: ',
    roc_auc_score(y_train, (rf.predict_proba(X_train[selected_features]))[:,
                                                                          1]))
print(
    'test auc score: ',
    roc_auc_score(y_test, (rf.predict_proba(X_test[selected_features]))[:, 1]))
```

In the following output, we see that the performance of the model fit using the selected features is very similar to the performance of the model trained using the entire dataset:

```
train auc score:  0.9999882222693333
test auc score:  0.9959119496855346
```

We see that by reducing the number of features we obtain a model that performs similarly, but is faster to train, and easier to maintain and interpret.

9.3 Probe features selection

An intuitive way to determine if a feature is important, consists of adding random variables to the dataset, which we'll call *probes*, and then compare the feature importance to that of the

probes. Because the probes are random, we know that they have **no impact** on the model's predictions. Hence, their importance value conveys a benchmark: we'll select those features whose importance is greater than that of the probes.

On face value, this method is solid: we introduce probes, which we know are redundant, and like that, we have a benchmark; we are not in the darkness any more. By comparing the feature importance to the benchmark, we can select features that we know are, at least, better than random.

Another advantage of introducing probes is that we can compare feature importance to that of the probes by using pretty much any selection process that we discussed in the book. So in theory, there is little to argue against adding probes to the dataset.

In practice, however, there are a few important things that we need to consider. To start, we mentioned previously that correlation masks feature importance; features that are important may show reduced relevance if there are correlated variables in the dataset. Hence, depending on how much the importance is reduced, we may, in some cases, lose some relevant features when comparing them to the probes.

A perhaps more important issue is that cardinality does impact the importance of features, in particular when selecting features for tree-based models. If our datasets contain discrete variables, we should add discrete probes. If our datasets contain continuous variables, we should add continuous probes, and if our datasets have a mix of continuous and discrete variables, we should add continuous and discrete probes and compare them like to like, which starts getting a bit more complex to implement.

We can add probes to the data and then select those features whose importance is greater than the probes. That means that we need a method to derive feature importance. Fortunately, we've discussed many methods throughout the book. We could, for example, rank features by using statistical tests, like ANOVA or mutual information (see *Chapter 4*). We could rank features based on single feature models or target encoding (see *Chapter 5*). We could use embedded methods (see *Chapter 7*). All of these methods need an arbitrary threshold to select features, which is in general, provided by the user. With probes, we don't need to provide a threshold any more, because now, the probes provide the benchmark.

Python implementation

In this demo, we'll select features by comparing their importance to that of probes using the open-source library Feature-engine. At the time of writing, Feature-engine supports the creation of continuous probes with a normal or uniform distribution, and of discrete probes with a binomial distribution. We can add as many probes as we like.

Feature-engine determines feature importance through embedded methods or single feature models, though more possibilities may be supported in the future. After ranking variables with

any of the importance methods, Feature-engine will select those features whose importance is greater than the mean importance of all probes.

> **Tip**
>
> After determining the importance of various types of probes, you'll notice that discrete probes usually show a lower importance value. So, while Feature-engine will take the average of all probes, in practice, it might be wiser to compare discrete variables to discrete probes and continuous variables to continuous probes, which, at the time of writing, needs to be done manually.

Let's start by importing the libraries, functions and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from feature_engine.selection import ProbeFeatureSelection
```

Let's load the breast cancer dataset and split it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=3)
```

All of the variables in the dataset are continuous, which we can quickly confirm by executing `X_train.hist(bins=50, figsize=(10,10))` (try it out). Therefore, we'll select features by adding continuous probes with a normal distribution. We'll determine feature importance by using an embedded method: the importance derived from random forests:

```
sel = ProbeFeatureSelection(
    estimator=RandomForestClassifier(random_state=10),
    collective=True,
    n_probes=3,
    distribution="normal",
    cv=3,
```

```
    random_state=150,
)
```

> **Note**
>
> With the parameter `collective=True` we inform `ProbeFeatureSelection` to use an embedded method to determine feature importance. When `False`, it will use single feature models instead, and then we need to pass a suitable performance metric, like the ROC-AUC for classification, or the mean squared error for regression. In both cases, the importance is determined with cross-validation to obtain the mean and standard deviation.

With `fit()` we trigger the addition of the probes, the calculation of feature importance, and the selection of the features:

```
sel.fit(X_train, y_train)
```

> **Note**
>
> Feature-engine will select features whose importance is greater than the mean importance of all probes, ignoring the standard deviation, at least at the time of writing.

We can go ahead and examine the distribution of the added probes like this:

```
sel.probe_features_.hist(bins=30)
plt.title("Distribution of the probe features")
plt.show()
```

In the following image, we see the distribution of the three probes added to the dataset:

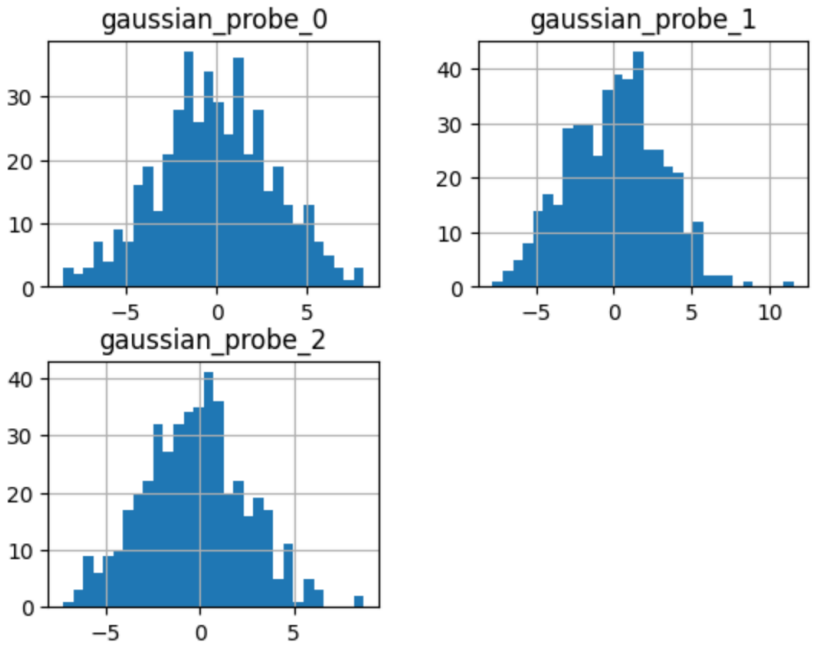


Figure 9.5: Random probes with a normal distribution added to the dataset.

We can also plot the feature importance including that of the probes as follows:

```
r = pd.concat([
    sel.feature_importances_,
    sel.feature_importances_std_
], axis=1)

r.columns = ["mean", "std"]

r.sort_values("mean", ascending=False)["mean"].plot.bar(
    yerr=[r['std'], r['std']], subplots=True, figsize=(15,6)
)
plt.title("Feature importance derived from the random forests")
plt.ylabel("Feature importance")
plt.show()
```

In the following plot, we see the feature importance derived from random forests. As expected, we can see that the probes show the least importance, and some of the features show importance values close to those of the probes:

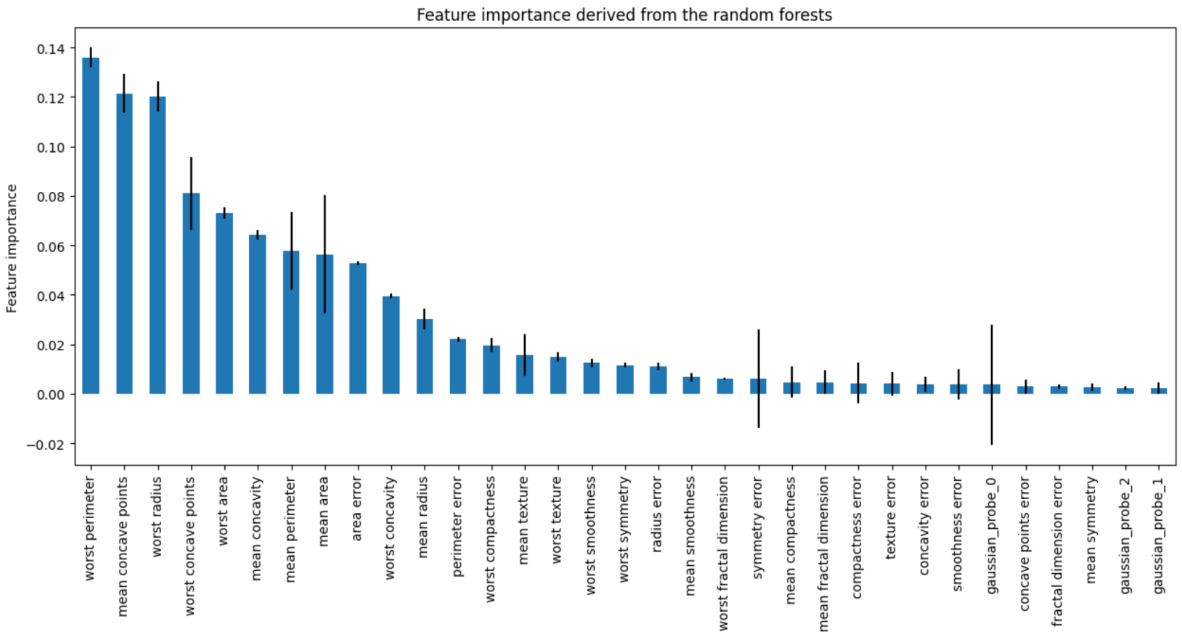


Figure 9.6: Feature importance derived from random forests.

We can display the features that will be removed like this:

```
sel.features_to_drop_
```

We see that only 1 feature will be removed, which means that almost every feature has a relevance greater than random features.

```
['mean symmetry']
```

Although, in principle, all features show importance greater than the probes, by looking at the bar plot we see that many of the importance values are within the dispersion (error bars) of the probes, and hence, they are not significantly different from random. Hence, we could remove more variables than those selected by `ProbeFeatureSelection`.

Let's wrap up by reducing the datasets to the selected feature subset:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

And that's it! We've now selected features using probes.

9.4 Minimum Redundancy Maximum Relevance (MRMR)

In the ideal world, we want to select features that have a strong relationship with the target variable (maximum relevance) but weak relationship to other features (minimum redundancy). The question then boils down to how to determine relevance, how to assess redundancy, and how to combine both values to determine if a feature should be selected.

The Minimum Redundancy Maximum Relevance (MRMR) algorithm obtains a measure of relevance and a measure of redundancy, and then it assigns an importance score to each feature based on the *difference* or *ratio* between relevance and redundancy. After that, it selects the features with the highest scores.

MRMR works as follows:

1. Determine the relevance of all predictor variables and select the feature with highest relevance.
2. Determine the redundancy between the remaining features and the feature selected in *step 1*.
3. Calculate the importance score as the ratio or difference between relevance and redundancy and select the feature with the maximum value.
4. Determine the *mean* redundancy between the remaining features and the features selected so far.
5. Calculate the importance score and select the feature with the maximum value.
6. Repeat 4 and 5 until the desired number of features is reached.

MRMR is an iterative algorithm. At each iteration, it determines the mean redundancy between the remaining features and the features that were selected in previous rounds. With the redundancy, it obtains a new measure of importance, and then it selects the feature with highest importance.

MRMR has 3 strategies to determine feature relevance, or in other words the strength of the relationship between predictors and target:

- F-statistic: derived from ANOVA if the target is discrete or correlation if the target is categorical.
- The mutual information.
- The importance derived from random forests.

MRMR has 2 strategies to obtain redundancy, or in other words, the relationship among predictors variables: Pearson's correlation coefficient and mutual information.

MRMR combines the values of relevance and redundancy as follows:

Table 9.1: Relevance and redundancy combinations to assess feature importance by MRMR.

Method	Relevance	Redundancy	Scheme
MID	Mutual information	Mutual information	Difference
MIQ	Mutual information	Mutual information	Ratio
FCD	F-Statistic	Correlation	Difference
FCQ	F-Statistic	Correlation	Ratio
RFCQ	Random Forests	Correlation	Ratio

With a value of relevance and a value of redundancy for each feature, MRMR ranks variables and selects the highest ranking feature at each iteration.

At each round, a new value of redundancy needs to be determined, because there is another feature in the selected subset. The search ends after selecting a pre-defined number of features.

> **Note**
>
> MRMR is fast to compute if using the F-statistic and correlation, which is what made this method popular (that, and the fact that it was used by Uber). However, these statistical values are suitable to evaluate linear relationships. Mutual information inherently detects linear and non-linear associations, but for continuous features it takes longer to compute, impacting the speed of selection with MRMR.

Python implementation

We'll implement MRMR using the open-source library Feature-engine. Let's start by importing the libraries, functions and classes:

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from feature_engine.selection import MRMR
```

Let's load the breast cancer dataset and split it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
y = y.map({0:1, 1:0})

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=3)
```

We'll set up the MRMR algorithm to determine relevance using the F-statistic, redundancy with Pearson's correlation coefficient, and rank features by using the ratio between the two:

```
sel = MRMR(method="FCQ", regression=False)
```

> **Tip**
>
> The parameter `method` informs `MRMR()` how to determine relevance and redundancy, and how to combine both values to select the features. For more details, check out Feature-engine's documentation: https://feature-engine.trainindata.com/en/latest/api_doc/selection/MRMR.html

With `fit()` we trigger the search:

```
sel.fit(X_train, y_train)
```

`MRMR()` stores the relevance in one of its attributes, so we can create a plot as follows:

```
pd.Series(sel.relevance_, index=sel.variables_).sort_values(
    ascending=False).plot.bar(figsize=(15, 4))
plt.title("Relevance")
plt.show()
```

In the following image, we see the feature relevance, that is, their relationship with the target, here calculated as the F-statistic derived from ANOVA:

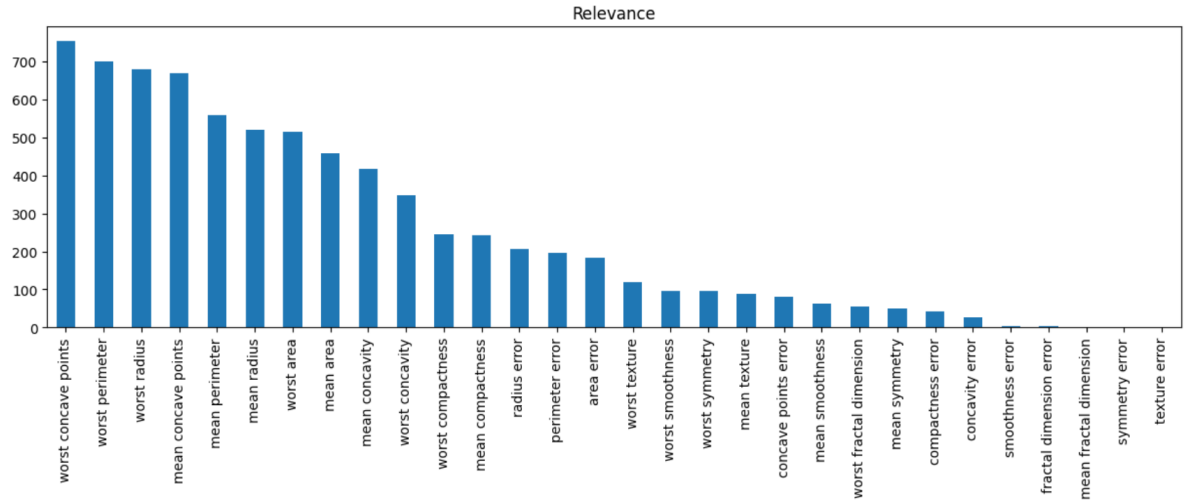


Figure 9.7: Feature relevance to the target assessed as the f-statistic derived from ANOVA.

> **Note**
>
> While features get one value for relevance throughout the search, the value for redundancy changes round after round, because the redundancy is calculated as the mean association with all the features that were selected in previous rounds. Each round adds a new feature, and hence, we need to determine a new value of redundancy for the remaining variables. That's why we haven't come up with a way to store this information in the transformer.

We can display the features that will be removed as follows:

```
sel.features_to_drop_
```

In the following output, we see the features that were not selected:

```
['mean radius',
 'mean texture',
 'mean area',
 'mean smoothness',
 'mean compactness',
 'mean concavity',
 'mean symmetry',
 'mean fractal dimension',
 'radius error',
```

```
 'texture error',
 'perimeter error',
 'area error',
 'smoothness error',
 'compactness error',
 'concavity error',
 'concave points error',
 'symmetry error',
 'fractal dimension error',
 'worst texture',
 'worst smoothness',
 'worst compactness',
 'worst concavity',
 'worst symmetry',
 'worst fractal dimension']
```

> **!** Important
>
> `MRMR()` will select 20% of the features by default. If we want to select a specific number of features, we must specify so through the parameter `max_features` when setting up the transformer.

Finally, we can reduce the datasets to the selected features like this:

```
X_train_t = sel.transform(X_train)
X_test_t = sel.transform(X_test)
```

And that's it! We've now selected features using MRMR.

9.5 Boruta

I debated for a long time whether to include Boruta in this book. The reason is that Boruta is designed to find **all** relevant features in a dataset, rather than the optimal subset that produces the best-performing model. As a result, it doesn't necessarily lead to simpler models.

However, given Boruta's popularity in former data science competitions and its continued demand online, I decided to cover it, allowing you to judge whether it's a tool worth using.

I'll begin by introducing the method, explaining why I believe it's not ideal for most machine learning applications, and then demonstrate its Python implementation.

Boruta's mechanism

Boruta is a feature selection method designed to identify **all** features that have a significant association with the target variable. The algorithm works by creating randomized copies of each feature (called shadow features) and adding them to the original dataset. It then trains a random forest classifier on this extended dataset. Next, Boruta compares the importance score of each original feature against the importance scores of the shadow features using a statistical test. If the original feature's importance significantly exceeds that of the randomized counterparts, the algorithm retains the feature; otherwise, it discards it.

The Boruta algorithm works as follows:

1. Create a randomized copy of the dataset (called shadow features) and add it to the original data.
2. Train a random forest model using the original and shadow features.
3. Obtain feature importance for original and shadow features.
4. Use a statistical test to identify features whose importance is significantly greater than that of the maximum of the shadow attributes.
5. Select the features in 4.
6. Repeat steps 1-5 with the remaining features.

Each random forest training step (*step 2*) is called an iteration.

In *step 4*, we test for significance using a two-tailed binomial test. Specifically, we track how often a feature's importance exceeds that of the maximum of the shadow features across iterations. For example, if a feature outperforms the shadows in 3 out of 5 iterations, we use this ratio (3/5) in a binomial test to determine whether the result is statistically significant.

> Tip
>
> In short, if we train 10 random forests, and the feature is important, we expect its importance to be greater than the maximum of the shadow features more than 5 times. This can be statistically determined by the binomial distribution test.

The process is done in iterations for two reasons:

1. With huge datasets, removing features at early iterations helps speed computations to assess the remaining features.
2. In random forests, correlation among features may mask true importance. By removing features early, we can help unmask important features that may have been correlated to those removed.

The Boruta algorithm stops when one of the following happens:

- All features have been examined and deemed important or unimportant.
- A maximum number of iterations is reached.

> **Note**
>
> In Step 4, a feature's importance is compared to the maximum shadow feature importance. There are three possible outcomes:
>
> 1. If the feature's importance is significantly greater, it is selected.
> 2. If the importance is not significantly different, the feature is removed.
> 3. If the result is inconclusive, the feature proceeds to the next iteration for re-evaluation.

Boruta evaluates feature significance by performing a binomial test for each feature. Since this involves multiple hypothesis testing (we are evaluating many features simultaneously), the algorithm uses either the Bonferroni correction or the Benjamini-Hochberg procedure to limit false positives.

Additionally, because the same feature undergoes repeated testing across iterations, we can apply the Bonferroni correction **again** to further reduce false discoveries.

Limitations

Boruta is slow; training multiple random forests takes time. In addition, to assess thousands of features we need random forests with numerous trees to ensure that all features are included in a tree at least once. That increases computation time even further. Moreover, we need to allow a sufficient number of iterations to be able to say if a feature's importance is significantly greater than that of the shadows, which again, increases computation time even further.

Finally, Boruta was specifically designed to identify **all** potentially relevant features, often resulting in large retained feature sets. This contrasts with typical machine learning objectives, where the goal is to minimize the feature space to improve model interpretability and simplicity while maintaining predictive performance.

> **Tip**
>
> Boruta was developed in a scientific research context with the explicit goal of identifying all potentially relevant features (e.g., genes or chemical compounds) associated with a target variable, rather than optimizing predictive performance through feature subset reduction. This explains its particular value in exploratory research settings where comprehensive

feature identification is prioritized, while making it less optimal for traditional machine learning applications where selection of reduced feature spaces is typically preferred.

Python implementation

To wrap up, we'll implement Boruta using the open-source Python library `BorutaPy`. Let's start by importing the libraries, functions and classes:

> **Tip**
>
> At the time of writing, Boruta is not available in Feature-engine, but that may change in the future, so keep checking the documentation.

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

from boruta import BorutaPy
```

Let's load the breast cancer dataset and split it into a training and a testing set:

```
X, y = load_breast_cancer(return_X_y=True, as_frame=True)
y = y.map({0:1, 1:0})

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=3)
```

Let's set up the random forest classifier that we want to use for the feature selection process:

```
rf = RandomForestClassifier(
    max_depth=4,
    random_state=0,
)
```

We'll set up the Boruta algorithm to select features whose importance is greater than the maximum importance of any of the shadow features:

```
sel = BorutaPy(
    estimator = rf,
    n_estimators='auto',
    perc=100,
    alpha=0.05,
    two_step=False,
    max_iter=20,
    verbose=2,
    random_state=0,
)
```

There are a number of important parameters in this implementation:

- `estimator` is the machine learning model to use in the selection. In general, it'll be random forests.
- `n_estimators` is the number of models in the ensemble.
- `perc` determines which value of importance of the shadow features to use. If `100`, then a feature is deemed important if its importance is greater than the **maximum** importance of the shadow features. If we set it to 90, then, a feature will be deemed important if it is greater than the 90th percentile importance of the shadow features. The lower the percentage, the more conservative the selection (i.e., we select more features).
- `alpha` is the value of hypothesis rejection. In general, we use 0.05.
- `two_step` indicates how to apply the correction for multiple tests. If `False`, then it will carry out Bonferroni correction for testing multiple features. If `True`, it will use the Benjamini-Hochberg correction for the multi-feature comparison, and the Bonferroni correction for testing the same feature multiple times.
- `max_iter` is the number of ìterations or rounds of testing.

> **Note**
>
> To guarantee thorough evaluation of all features, Boruta requires sufficiently large random forests. The BorutaPy implementation can automatically determine the optimal number of trees when the `n_estimators` parameter is set to `"auto"`. For manual configuration, ensure `n_estimators` is large enough to provide adequate feature coverage.

With `fit()` we trigger the feature selection process:

```
sel.fit(X_train, y_train)
```

> **Note**
>
> During the search, we'll see the number of features selected at each iteration. Note however, that we need a sufficient number of iterations before the binomial distribution test can reliably assess if a feature is significantly important. Hence, in the first 5 rounds, you'll probably see that Boruta does not accept or reject any feature.
> I made a pull request (PR) to BorutaPy to defer the testing of features to the fifth iteration. If it is eventually merged, you'll notice that the features are tested with the binomial distribution from round 5. At the time of writing, they are tested from the first iteration.

We can find the selected features as follows:

```
sel.support_
```

`True` marks the features that were selected and `False` those that will be removed.

```
array([ True,  True,  True,  True,  True,  True,  True,  True, False,
       False,  True, False,  True,  True, False, False,  True, False,
       False, False,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True])
```

I mentioned previously that at each iteration, features are selected as important, removed as non-important or passed to the next round for further evaluation. We can find those features for which Boruta couldn't determine if they were truly significant as follows:

```
sel.support_weak_
```

`True` marks the 5 features for which the results are inconclusive:

```
array([False, False, False, False, False, False, False, False,  True,
        True, False,  True, False, False, False,  True, False,  True,
       False, False, False, False, False, False, False, False, False,
       False, False, False])
```

> **Important**
>
> If many features remain undetermined (i.e., Boruta cannot confidently classify them as significant or not), consider increasing the `max_iter` parameter to allow for additional evaluation rounds. This gives the algorithm more opportunities to resolve borderline cases.

Since we are working with dataframes, we can obtain a dataframe as a result as follows:

```
X_train_t = sel.transform(X_train, return_df=True)
X_test_t = sel.transform(X_test, return_df=True)
```

If we now execute `X_test_t.shape`, we'll see that only `22` from the 30 features were selected.

> **Note**
>
> If we are not working with dataframes, then we need to make sure that we set `return_df` to `False` when applying `transform()`.

BorutaPy offers a ranking for features. We can plot the features' ranking as follows:

```
ranking = pd.Series(sel.ranking_, index=X_train.columns)

ranking.sort_values().plot.bar()
plt.ylabel("ranking")
plt.xlabel("features")
plt.title("ranking of features")
plt.show()
```

In the following output, we see the 22 selected features with the ranking of 1, the non-selected features with the ranking 3, and the features that were inconclusive with a ranking of 2:

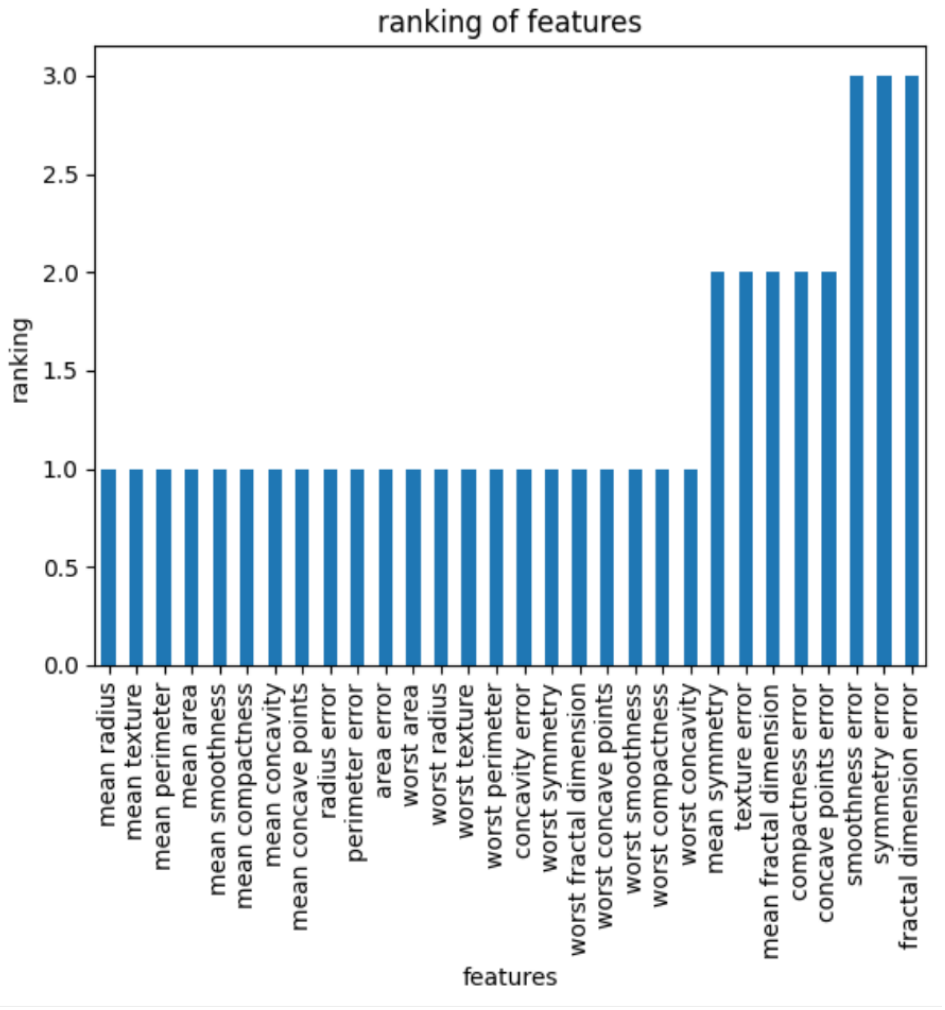


Figure 9.8: Feature ranking based on whether a feature was selected, discarded or inconclusive.

> **Note**
>
> Boruta was specifically designed to identify all statistically relevant features rather than to rank them by importance. Consequently, while BorutaPy provides a feature ranking, this ordering should not be overinterpreted—the algorithm's primary value lies in its selection capability.

And that's it! We've now selected features using Boruta.

9.6 References

- Breiman, Random Forests, Machine Learning 45:5–32, 2001.
- Stoppiglia, et al., “Ranking a Random Feature for Variable and Feature Selection”. JMLR: 1399-1414, 2003.
- Kursa, et al., “Boruta - A System for Feature Selection”, Fundamenta Informaticae 101(4):271-285, 2010.
- Kursa and Rudnicki, ” Feature Selection with the Boruta Package”, J. Stat. Soft., 36(11): 1–13, 2010.

More from the author

Congratulations!

Feature selection is one of those skills that quietly separates good data scientists from great ones. By working through this book, you've added a rigorous, practical toolkit to your arsenal that most practitioners simply don't have. Well done.

If this book has been useful, the best thing you can do is share it — with a colleague, on LinkedIn, or by leaving a review on Trustpilot. It makes a real difference.

Keep going

This book is part of a growing library of advanced, hands-on machine learning titles. If you want to go deeper, the next two books are waiting:

- Python Feature Engineering Cookbook — over 70 recipes for engineering features in a professional and elegant manner using Python open-source tools. Available on Packt (https://www.packtpub.com/en-us/product/python-feature-engineering-cookbook-9781835883587)
- Imbalanced Data, Myth, Mistakes and Modern Solutions — cut through the noise. This book puts popular techniques like resampling to the test, exposes the myths, and hands you a proven framework for tackling imbalanced data head-on (https://www.trainindata.com/p/imbalanced-data-myths-mistakes-solutions-book).

And if you learn better through video and code-along exercises, all of these topics — and more — are covered in depth at https://www.trainindata.com/courses.

I look forward to seeing you again!