Machine Learning With Python for Beginners

A Step-By-Step Guide With Hands-On Projects

JAMIE CHAN

Machine Learning with Python for Beginners

A Step-by-Step Guide with Hands-On Projects

by Jamie Chan

https://www.learncodingfast.com/machine-learning

Preface

Machine learning is becoming mainstream in recent years and has revolutionized every aspect of our lives. From facial recognition on social media platforms to route suggestions by digital maps, it is at the heart of countless technological breakthroughs.

With the increase in the use of machine learning, there has been a surge in demand for individuals with relevant skills. Perhaps you would like to add machine learning features to your projects, or you are required to pick up some machine learning skills for your job. Whatever your reason is for learning machine learning, this book aims to cover the major concepts in a step-by-step fashion and provide you with an excellent base to explore further.

The book uses a hands-on approach and includes many code examples for you to try out. In addition, you'll get a chance to practice what you learned with three hands-on projects.

You can download all the code examples, images, and data files used in this book at https://www.learncodingfast.com/machine-learning.

I sincerely hope you find this book useful. If you have any questions or suggestions regarding the book, feel free to reach out to me at jamie@learncodingfast.com.

More books by Jamie

https://www.learncodingfast.com/recommend/

Python Series

Learn Python in One Day and Learn It Well (2nd Edition)
Learn Python in One Day and Learn It Well (2nd Edition) – Workbook

General Programming Series

Learn C# in One Day and Learn It Well
Learn Java in One Day and Learn It Well

Web Programming Series

Learn CSS in One Day and Learn It Well
Learn PHP in One Day and Learn It Well
Learn SQL in One Day and Learn It Well

Contents

Chapter 1 - Introduction

Welcome to the world of machine learning! Whether you are an aspiring data scientist or just curious about machine learning, this book is designed to help you grasp the fundamental concepts of machine learning in a systematic and step-by-step fashion.

The book aims to be as beginner-friendly as possible. However, note that it has some prerequisites.

Firstly, the book assumes that you are familiar with statistical measures such as the mean, median, mode, variance, and standard deviation. If you have forgotten these concepts, you can refer to https://www.learncodingfast.com/machine-learning for a quick recap.

Next, you need to be comfortable with basic Python, especially with the concept of object-oriented programming in Python. It'll also be great if you are familiar with Python lists and dictionaries, as we'll be working with enhanced versions of lists and dictionaries in this book.

If you are new to Python, I strongly recommend reading my introductory book "Learn Python in One Day and Learn It Well (2nd edition)" before proceeding. If you are already comfortable with Python, let's move on.

1.1 What is Machine Learning?

In recent years, artificial intelligence (AI) algorithms have become widely available and have fundamentally changed fields ranging from business analytics to healthcare. AI is an umbrella concept that refers to any technique that enables computers to mimic human behavior; machine learning is a subset of AI.

Machine learning is concerned with giving computers the ability to *perform a task without being explicitly programmed*.

As an example, suppose we want to sort emails into promotional and non-promotional emails. In conventional programming, we can do this using a set of hard-coded rules or conditional statements. For instance, one possible rule is to classify an email as promotional if it contains the words “Discount”, “Sale”, or “Free Gift”. We can also classify an email as non-promotional if the email address includes “.gov” or “.edu”.

The problem with such an approach is that it is challenging to come up with the rules. For instance, while most emails from addresses that contain “.edu” are likely to be non-promotional (such as an email from your thesis supervisor), it is also possible for educational institutions to send promotional emails advertising their courses.

It is almost impossible to come up with a set of rules that considers all possible scenarios. This is where machine learning can come in handy. Machine learning can super-charge the sorting program by identifying each email’s unique attributes and autonomously derive robust rules to automate the sorting process, thereby preventing the need for manually engineered rules.

For a machine to do that, we need to provide it with data. The goal is for the machine to learn the rules directly from the data, using what are known as machine learning algorithms.

In a nutshell, machine learning algorithms are made up of formulas and procedures derived from mathematical concepts in linear algebra, calculus, probability, statistics, and other fields. These formulas and

procedures are implemented in programming code and used to perform calculations on our data.

After performing the calculations, the algorithm typically generates an output known as a model. The process of generating the model is known as training the model. This model describes the rules, numbers, and any other algorithm-specific data structures that our machine learned from the data. Our machine can then use the model to perform its task.

Two of the most common tasks machine learning models perform are classification (e.g., classifying emails into promotional and non-promotional) and prediction (e.g., predicting stock prices). Other tasks include making recommendations, image recognition, and natural language processing.

As an example of how machine learning works, let us briefly discuss one of the most fundamental algorithms in machine learning: the simple linear regression algorithm. In simple linear regression, the task is to establish a linear relationship between an independent variable (x) and a dependent variable (y) and to use the relationship to make predictions. We can represent this relationship as a linear equation of the form $y = a + bx$.

The linear regression algorithm aims to find the best possible values for the coefficients “a” and “b” by performing calculations on the data provided. Once the calculations are done, the linear regression algorithm returns a model, including the values of “a” and “b”. Our machine can then use the equation $y = a + bx$ (with known values of “a” and “b”) to make predictions.

Machine learning models range from simple to complex. Some complex models involve neural networks and are part of a subset of machine learning known as deep learning. These models are inspired by the structure of our brains and involve networks with multiple connected layers. We will not be covering deep learning models in this book.

1.2 Approaches to Machine Learning

There are three main approaches to machine learning: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning is typically employed when we want our machine to make predictions or classifications. In such cases, we need to provide the machine learning algorithm with labeled data.

For example, suppose we want to predict the price of a house based on its location and floor area. We can provide the machine learning algorithm with a CSV file that lists the prices of 10,000 houses, along with the location and floor area of each house.

In machine learning jargon, this collection of data is known as a dataset and we say that the dataset consists of 10,000 instances (also known as observations, samples, or examples). Location and floor area are known as features in the dataset, while price is the target variable. The value of the target variable for each instance is known as the label of that instance.

The machine learning algorithm aims to find a connection between the features (location and floor area) and the target variable (price). Once the link is identified, we can use the resulting relationship to predict housing prices in a new dataset.

In contrast to supervised learning, we have unsupervised learning. In unsupervised learning, the machine learning algorithms are provided with instances without their respective labels. Here, the algorithms aim to identify hidden patterns in the dataset.

As an example, assume we would like to sort a basket full of mixed fruits into their respective varieties (apples, oranges, bananas, etc.). We can use unsupervised learning to cluster them into unique groups based on their features (such as their shape, color, and sweetness).

As we did not label the fruits in our training dataset, the algorithm will not be able to label the groups. For instance, if we have five apples and three

oranges, the machine learning algorithm may cluster the fruits into two groups. However, it will not be able to label group 1 as “apples” or group 2 as “oranges”. All it does is cluster the fruits based on their feature similarities.

Unsupervised learning has been frequently used in business analytics to discover patterns in customers’ shopping habits and segment them based on these patterns.

Last but not least, we have reinforcement learning. Reinforcement learning aims to determine optimal behavior through trial and error by interacting continuously with an environment while assessing each action’s rewards and penalties.

A well-known example of reinforcement learning is the AlphaGo program developed by DeepMind. AlphaGo is a computer program that utilizes reinforcement learning to play a board game called Go. In 2016, it defeated Go champion Lee Sedol 4-1, becoming the first computer program to defeat a world campaign Go player.

Reinforcement learning is an advanced topic in machine learning and will not be covered in this book.

1.3 Life Cycle of a Machine Learning Project

Despite the diverse applications of machine learning, most machine learning projects follow a typical life cycle that includes some (or all) of the steps listed in this section.

We'll give an overview of the steps in this chapter. In subsequent chapters, you'll learn to perform the steps in Python when we discuss different machine learning libraries. For the sake of brevity, we assume that the data is already collected and available for use.

Step 1. Loading the data

The first step to any machine learning project is to load the data. Based on the data at hand, we would need different libraries to load the respective data. For example, for loading CSV files, we need the pandas library. For loading 2D images, we can use the Pillow or OpenCV library.

Step 2. Examine the data

Assuming the data has been loaded correctly, the next step is to examine the data to get a general feel for the dataset. Let us take the case of a simple CSV file-based dataset.

For starters, we can look at the dataset's shape (i.e., the number of rows and columns in the dataset). We can also peek inside the dataset by looking at its first 10 or 20 rows. In addition, we can perform fundamental analysis on the data to generate some descriptive statistical measures (such as the mean, standard deviation, minimum and maximum values).

Last but not least, we can check if the dataset contains missing data. If there are missing values, we need to handle them.

Step 3. Split the Dataset

Before we handle missing values or do any form of computation on our dataset, we typically split it into training and test subsets. A common practice is to use 80% of the dataset for training and 20% for testing.

The training subset is the actual dataset used for training the model. After the training process is complete, we can use the test subset to evaluate how well the model generalizes to unseen data (i.e., data not used to train the model).

It is crucial to treat the test subset as if it does not exist during the training stage. Therefore, we should not touch it until we have finished training our model and are ready to evaluate the selected model.

Step 4. Visualizing the data

After splitting the dataset, we can plot some graphs to better understand the data we are investigating. For instance, we can plot scatter plots to investigate the relationships between the features and the target variable.

Step 5. Data Preprocessing

The next step is to do data preprocessing. More often than not, the data that we receive is not ready to be used immediately.

Some problems with the dataset include missing values, textual and categorical data (e.g., “Red”, “Green”, and “Blue” for color), or range of features that differ too much (such as a feature with a range of 0 to 10,000 and another with a range of 0 to 5).

Most machine learning algorithms do not perform well when any of the issues above exist in the dataset. Therefore, we need to process the data before passing it to the algorithm.

Step 6. Train the Models

After processing the data, we are ready to train our models. Based on the previous steps of analyzing the dataset, we can narrow down appropriate machine learning algorithms and build models using those algorithms.

Step 7. Evaluate the models

After building the models, we need to evaluate our models using different metrics and select the best-performing model for deployment.

At this stage, the machine learning project is more or less complete.

What follows include steps for monitoring and maintaining the system.

1.4 Overview of the Book

Now that we have a good overview of the steps involved in a machine learning project, we are ready to get hands-on with building our models. Here's an overview of what the book covers.

First, we'll learn four Python libraries - NumPy, pandas, Matplotlib, and Scikit-Learn - that are essential to any machine learning project.

The NumPy and pandas libraries are for working with our dataset. The Matplotlib library is for plotting graphs, and the Scikit-Learn library is for machine learning tasks like data preprocessing, model training, and model evaluation.

Next, we'll discuss three common machine learning applications - regression, classification, and clustering. We'll cover one or more algorithms for each application, including a high-level discussion of the mathematics behind the algorithms and how we can apply those algorithms using Scikit-Learn.

After covering the algorithms, we'll move on to more advanced topics, including dimensionality reduction and hyperparameter tuning.

Finally, we'll end the book with three hands-on projects, where you get to apply the concepts covered. Ready?

1.5 Jupyter Notebook

To follow along with the examples in this book, you need to have access to Jupyter Notebook.

Jupyter Notebook is an open-source interactive computational environment that is based on a server-client structure. It includes a web server and a web application that works like an integrated development environment (IDE). This web application allows us to create Jupyter Notebook documents (commonly referred to simply as notebooks or IPYNB files) that consist of code, text, and images.

To use Jupyter notebook, we have two options.

Anaconda

The first is to install it on our system. The easiest way to do that is to install a Python distribution known as Anaconda.

Anaconda comes with over 250 packages pre-installed, including NumPy, pandas, Matplotlib, and Scikit-Learn, all of which we'll be using in this book. In addition, it includes many useful applications and IDEs, such as the Jupyter Notebook application mentioned above.

Google Colaboratory

The second option to access Jupyter Notebook is to use Google Colab (short for Google Colaboratory).

Google Colab is a free cloud-based Jupyter Notebook environment provided by Google that requires no installation and offers free access to online computing resources. However, you need to be connected to the internet when you use Google Colab. This connection is mainly used to run the code and does not consume much data.

We'll be using Google Colab for the rest of this book. If you prefer to install Jupyter Notebook on your system instead, you can visit https://www.learncodingfast.com/machine-learning for installation instructions and more information.

To use Google Colab, head over to https://colab.research.google.com/. You'll need to sign in to your Google account if you are not already signed in. If you do not have a Google account, create one for free at https://accounts.google.com/signup.

After signing in, you'll be presented with a dialogue box. Click on "New notebook" to create a new Jupyter notebook. Next, click on the file's title (refer to the screenshot below) and rename it to *Examples.ipynb*. We'll learn to create the document below:

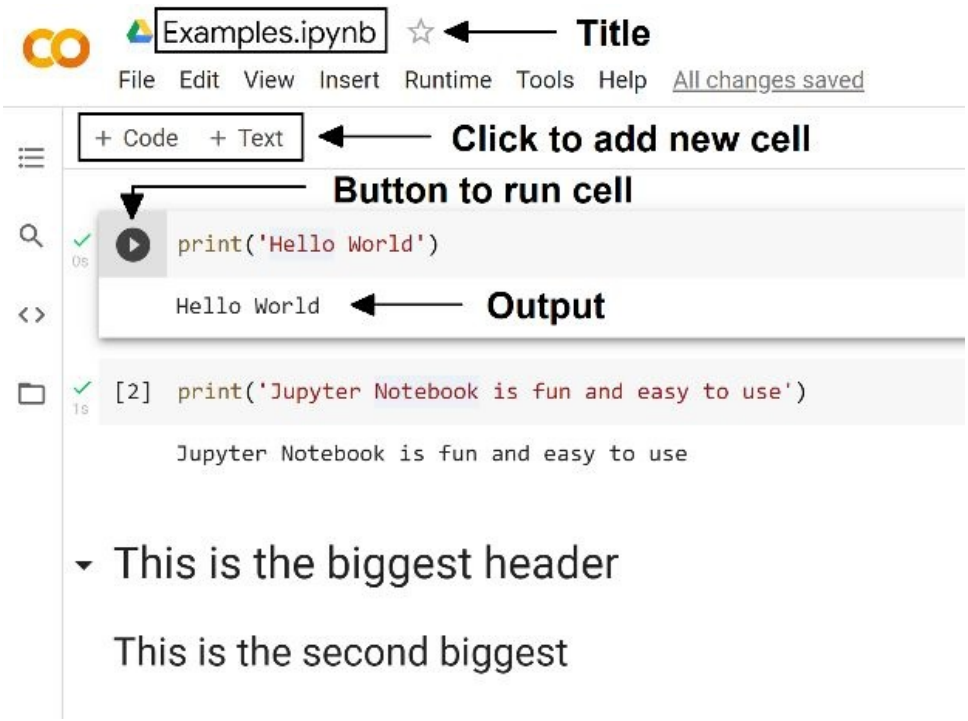

Figure 1.1: Our first Google Colab notebook

This document consists of two code cells and one text cell.

A code cell allows us to add code to the document. We can run the code in the cell by clicking on the button on the left of the cell. Alternatively, we can click inside the cell and press Shift-Enter on our keyboard. If the code produces any output, the output will be displayed below the code.

When we create a new notebook, we get an empty code cell by default. In addition, if we run the last cell in the notebook *by pressing Shift-Enter*, Google Colab adds a new code cell for us. If we need to create a code cell ourselves, the easiest way is to click on the "+ Code" button found at

the top of the document.

Next, we have text cells. A text cell allows us to add formatted text and images to our document. Google Colab supports the use of Markdown, a markup language that is a superset of HTML. To create a text cell, click on “+ Text” at the top of the document.

Now, let’s create the document shown above. To do that, click on the first cell in *Examples.ipynb*, add the following code to it, and run the cell by *pressing Shift-Enter* on your keyboard:

```
print('Hello World')
```

The first cell will likely take a few seconds to run as we need to connect to the online resources. Next, click on the second code cell, add the following code, and run it by pressing Shift-Enter:

```
print('Jupyter Notebook is fun and easy to use')
```

Finally, delete the code cell that is automatically added. To delete a cell, click inside the cell and click on the dustbin icon. Next, create a text cell and add the following text to it:

```
# This is the biggest header
## This is the second biggest
```

Markdown uses number signs (#) followed by a space to add headers. The more number signs you use, the smaller the header. Here, we create two headers, the second smaller than the first. Run the cell by pressing Shift-Enter on your keyboard.

After creating the document, we can save the file by pressing Ctrl-S on our keyboard (or selecting “File > Save” on the top menu); this saves the file in our Google Drive. If you go to https://drive.google.com/ now and log in with the same Google account you used for Google Colab, you’ll see a folder called “Colab Notebooks”. All your notebooks are stored here. You can open any file by double-clicking on it.

That’s all for Jupyter Notebook. Be sure to have access to it before proceeding to the next chapter.

The following chapters include many examples and code snippets. To try

out the examples, create a new code cell in your Jupyter Notebook document, type the code in, and run the cell. Within each chapter, *some code examples depend on code from previous examples of the same chapter*. Whenever you get disconnected from Google Colab, you can run all the cells again by clicking “Runtime > Run all” if you get any error message.

When you run the examples, your results may differ slightly from what you see in this book. These discrepancies are due to the randomness inherent in some machine learning algorithms and the differences in numerical precision in our systems.

You can download the completed notebook for each chapter at https://www.learncodingfast.com/machine-learning. In addition, you can download any data files needed and all the images displayed in this book at the same link.

Chapter 2 - NumPy

Now that we understand how machine learning works and have access to Jupyter Notebook, we are ready to get our feet wet with some actual coding. Before we discuss any machine learning algorithms, we need to be familiar with the NumPy library.

2.1 What is NumPy?

NumPy is a Python library designed to make it easy for us to work with arrays. In basic Python, arrays (also known as lists) have limited functionalities. While we can add and subtract lists, it is not easy to perform advanced mathematical operations on them.

Numpy solves this problem for us. With NumPy, we can perform various complex operations on arrays, including shape manipulation, basic linear algebra, random simulation, and more.

The main data structure in NumPy is an array object known as the ndarray. An ndarray is a multidimensional array of elements, where each element in the array is typically an array itself.

Datasets in machine learning are frequently stored as two-dimensional ndarrays. Therefore, familiarity with NumPy is an essential skill for any data scientist.

2.2 Importing the NumPy Library

To work with NumPy, we need to import it.

First, create a new notebook in Google Colab and name it *Chapter 2 - NumPy.ipynb*. Next, add the following code to the first cell and run it:

```
import numpy as np
```

It is customary to use **`np`** as the alias when importing NumPy; we'll follow the same convention in this book. (Note that this **`import`** statement does not produce any output when you run it.)

2.3 Creating a NumPy Array

After importing NumPy, we can use it to create an ndarray (also known as a NumPy array, or simply an array).

There are two main ways to do it. The first is to convert a Python array-like structure (such as a Python list or tuple) to an ndarray using the **`array()`** function. Let's look at some examples:

```
list1 = [1, 2, 3, 4]
list2 = [[1, 2, 3, 4]]
list3 = [[1, 2, 3, 4], [10, 20, 30, 40], [100, 200, 300, 400]]
arr1 = np.array(list1)
arr2 = np.array(list2)
arr3 = np.array(list3)
print(type(arr1), type(arr2), type(arr3))
```

Here, we first declare and initialize three Python lists (**`list1`**, **`list2`**, and **`list3`**).

Notice that **`list1`** and **`list2`** are very similar, except that **`list1`** has one set of square brackets while **`list2`** has two.

`list1` is a one-dimensional (1D) list with four elements.

`list2`, on the other hand, is a two-dimensional (2D) list with one element. This element - **`[1, 2, 3, 4]`** - is a list itself and consists of four elements.

`list3` is also a 2D list, with three nested lists of four elements each.

After declaring the lists, we pass them to the NumPy **`array()`** function to convert them to ndarrays. We then print the data types of the resulting arrays. If you run the code above, you'll get the following output:

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

This output indicates we have successfully converted **`list1`**, **`list2`**, and **`list3`** to NumPy arrays. We can print the shapes of these ndarrays. The shape of an ndarray (stored in the **`shape`** attribute) is given as a

tuple and tells us the number of elements in it:

```
print(arr1.shape)
print(arr2.shape)
print(arr3.shape)
```

This gives us the following output:

```
(4,)
(1, 4)
(3, 4)
```

`(4,)` tells us that **`arr1`** is a 1D array with four elements, while **`(1, 4)`** tells us that **`arr2`** is a 2D array with one nested array; this nested array has four elements.

Finally, the last line **`(3, 4)`** tells us that **`arr3`** is a 2D array with three nested arrays; each nested array has four elements.

The example above shows how to use the **`array()`** function to convert Python built-in structures to ndarrays. Next, let's learn to create NumPy arrays from scratch. To do that, we can use different predefined functions in the NumPy library. An example is the **`linspace()`** function.

`linspace(start,stop,num)` gives us a NumPy array of **`num`** evenly spaced numbers within the interval of **`start`** (inclusive) to **`stop`** (inclusive). Let's look at an example:

```
arr4 = np.linspace(0, 10, 5)
print(arr4)
```

This example gives us 5 evenly spaced numbers from 0 to 10. If you run the example, you'll get the following output:

```
[ 0.   2.5  5.   7.5 10. ]
```

`linspace()` generates floating-point numbers by default. In addition, Jupyter Notebook prints NumPy arrays without commas. Hence, we get decimal points after the integers (e.g., **`0.`**) and no commas in the output above.

2.4 Selecting Data from a NumPy Array

Similar to what we do with Python lists, we can access elements in a NumPy array using their indexes. Recall that indexes start from 0 and negative indexes start from the back.

```
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1[2])
print(arr1[-1])
```

The code above gives us

```
3
5
```

as the output. The next example shows how we can access elements in a 2D ndarray:

```
arr2 = np.array([['a', 'b', 'c'], ['d', 'e', 'f']])
print(arr2[0])
print(arr2[0][1])
print(arr2[0, 1])
```

Here, **`arr2`** is a 2D ndarray with two elements - **`['a','b','c']`** and **`['d','e','f']`**.

`arr2[0]` gives us the first element in **`arr2`**. In other words, it gives us the array **`['a','b','c']`**.

To access the elements in this array, we can use two sets of square brackets or a comma. For instance, to access the second element in **`arr2[0]`**, we use **`arr2[0][1]`** or **`arr2[0, 1]`**.

If you run the code above, you'll get the following output:

```
['a' 'b' 'c']
b
b
```

Next, we can slice a NumPy array. To do that, we use the **`[start:stop:step]`** notation. This gives us elements from index **`start`** to **`stop`** (including **`start`** but excluding **`stop`**), with a step of

`step`.

`start` and **`step`** have default values of 0 and 1, respectively. The default value for **`stop`** is the length of the array.

```
arr3 = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'])
print(arr3[1:6])
```

Here, we use the slice **`1:6`** to select elements from index 1 (i.e., the element **`'b'`**) to 5. This gives us the following output:

```
['b' 'c' 'd' 'e' 'f']
```

If we add a step of 2 to the slice, we'll get every 2nd element from index 1 to 5. In other words,

```
print(arr3[1:6:2])
```

gives us

```
['b' 'd' 'f']
```

We can also slice a 2D ndarray:

```
arr4 = np.array([[1, 2, 3, 4, 5], [10, 20, 30, 40, 50], [6, 7, 8, 9,
10]])
print(arr4[0:2, 2:4])
```

An easy way to work with 2D arrays is to think of them as tables, with their nested arrays arranged as rows. For instance, if we represent **`arr4`** above as a table, we'll get one with three rows and five columns.

	Columns				
	0	**1**	**2**	**3**	**4**
Row 0	1	2	3	4	5
Row 1	10	20	30	40	50
Row 2	6	7	8	9	10

The slice before the comma selects the rows, while the slice after it selects the columns.

Therefore, **`arr4[0:2, 2:4]`** selects elements from rows 0 to 1

(because of the slice `0:2`) and columns 2 to 3 (because of the slice `2:4`). If we run the code above, we'll get the following output

```
[[ 3  4]
[30 40]]
```

2.5 ndarray Methods

An ndarray comes with many useful methods defined in the **`ndarray`** class. To use these methods, we write the array name, followed by the dot operator and the name of the method. Let's look at some examples.

sum() and mean()

The **`sum()`** and **`mean()`** methods give us the sum and mean of an array, respectively.

Let's use **`sum()`** as an illustration. If we use **`sum()`** without specifying the axis, it returns the sum of all the elements in the array. On the other hand, if we specify **`axis=0`**, it sums the elements in each column, and if we specify **`axis=1`**, it sums the elements in each row. An example is shown below:

```
arr1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr1.sum())
print(arr1.sum(axis=0))
print(arr1.sum(axis=1))
```

This gives us the following output:

```
36
[ 6  8 10 12]
[10 26]
```

`arr1.sum()` gives us **`36`** as 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36.

`arr1.sum(axis=0)` gives us **`[6  8  10  12]`** as 1 + 5 = 6, 2 + 6 = 8, 3 + 7 = 10, and 4 + 8 = 12.

Finally, **`arr1.sum(axis=1)`** gives us **`[10  26]`** as 1 + 2 + 3 + 4 = 10 and 5 + 6 + 7 + 8 = 26.

reshape()

`reshape()` gives a new shape to a NumPy array without changing its data.

Recall that the shape of a *2D array* is represented as a tuple with two values (refer to Section 2.3). For instance, the shape of **`[[1, 2, 3], [4, 5, 6]]`** is (2, 3), which tells us that this array consists of two rows and three columns.

Suppose we have a 1D array with 8 elements and we want to reshape it to a (4, 2) 2D array, the example below shows how we can do it:

```
arr2 = np.array([1, 2, 3, 4, 5, 6, 7, 8])
reshaped_array = arr2.reshape((4, 2))
print(arr2)
print(reshaped_array)
```

If you run this example, you'll get the following output:

```
[1 2 3 4 5 6 7 8]
[[1 2]
[3 4]
[5 6]
[7 8]]
```

The first line shows the original array, which is not changed by the **`reshape()`** method. The remaining lines show the reshaped array, which is a (4, 2) array consisting of 4 rows and 2 columns.

It is necessary to know how to reshape 1D arrays to two-dimensional because certain methods in the Scikit-Learn library require 2D arrays as input. We'll learn more about this library in Chapter 5.

A useful trick when reshaping arrays to two-dimensional is to pass **`(n, -1)`** or **`(-1, n)`** to the **`reshape()`** method and let it infer the shape for us. For instance, for the example above, we can pass **`(4, -1)`** to reshape **`arr2`** to a 2D array with four rows and an unspecified number of columns (denoted by -1). When we do that, **`reshape()`** infers the number of columns and returns a (4, 2) array to us.

We can also pass **`(1, -1)`** to convert a 1D array to a 2D array with one row and an unspecified number of columns. An example is shown below:

```
arr3 = np.array([1, 2, 3, 4, 5])
reshaped_array_2 = arr3.reshape((1, -1))
print(arr3)
print(reshaped_array_2)
```

If you run the code above, you'll get the following output:

```
[1 2 3 4 5]
[[1 2 3 4 5]]
```

`[1 2 3 4 5]` is a 1D array, while **`[[1 2 3 4 5]]`** is a 2D array with one row and five columns.

2.6 NumPy Functions

Next, let's move on to NumPy functions. In the previous section, we learned about methods that are defined *inside* the **`ndarray`** class.

Besides these methods, NumPy comes with its own set of functions (also known as routines) that are defined *outside* the **`ndarray`** class. To use these functions, we pass the NumPy array as an argument to the function.

concatenate()

A commonly used NumPy function is **`concatenate()`**, which can be used to join two or more NumPy arrays.

When we use **`concatenate()`** to join arrays, we can specify whether we want to join along axis 0 (which is the default) or axis 1. Joining along axis 0 combines the rows and requires the arrays to have the same number of columns, while joining along axis 1 combines the columns and requires the arrays to have the same number of rows.

Let's look at some examples:

```
arr1 = np.array([[1, 2], [3, 4], [5, 6]])
arr2 = np.array([[7, 8]])
combined_array = np.concatenate((arr1, arr2))
print(combined_array)
```

Here, **`arr1`** can be viewed as a table with three rows and two columns, while **`arr2`** is a table with one row and two columns.

As the two arrays have the same number of columns, we can concatenate them along axis 0. We do that on the third line by passing a tuple of the two arrays - **`(arr1, arr2)`** - to the function. If you run the code above, you'll get the following output:

```
[[1 2]
[3 4]
[5 6]
[7 8]]
```

The rows in **arr1** and **arr2** have been combined to give us a new array with four rows and two columns.

If we want to add columns to this array (**combined_array**), we need to concatenate it with an array that has 4 rows (i.e., 4 nested arrays). An example is shown below:

```
arr3 = np.array([[9], [10], [11], [12]])
combined_array = np.concatenate((combined_array, arr3), axis=1)
print(combined_array)
```

Here, we concatenate **combined_array** with **arr3** along axis 1 and assign the result back to **combined_array**. If you run the code above, you'll get the following output:

```
[[ 1  2  9]
 [ 3  4 10]
 [ 5  6 11]
 [ 7  8 12]]
```

Wrapper Functions

Some functions in NumPy are wrappers around corresponding methods in the **ndarray** class. An example is the NumPy **reshape()** function. This function is a wrapper around the **reshape()** method in the **ndarray** class.

A wrapper is a function whose main purpose is to call another function or method. In other words, if you look under the hood of the NumPy **reshape()** function, you'll see that it calls the **reshape()** method in the **ndarray** class.

To use methods with wrapper functions, we have two options. Suppose we have an array

```
arr4 = np.array([1, 2, 3, 4, 5, 6])
```

and we want to reshape it, we can use the **reshape()** method directly:

```
reshaped1 = arr4.reshape((2, -1))
print(reshaped1)
```

or use the **reshape()** wrapper function:

```
reshaped2 = np.reshape(arr4, (2, -1))
print(reshaped2)
```

Both give us the same output:

```
[[1 2 3]
[4 5 6]]
```

Chapter 3 - pandas

In the previous chapter, we covered most of the important concepts in NumPy. Next, let’s move on to discuss another essential library for machine learning - the pandas library.

3.1 What is pandas?

pandas is an open-source library built on top of NumPy. It provides us with new data structures (such as Series and DataFrames) and is designed to make it easy for us to work with tabular data.

A pandas Series is a 1-dimensional data structure similar to a Python list or a 1D NumPy array; the main difference is it is labeled. A DataFrame, on the other hand, is a 2-dimensional labeled data structure.

3.2 Importing the pandas Library

To use the pandas library, we need to import it. First, create a new notebook in Google Colab and name it *Chapter 3 - pandas.ipynb*.

Next, add the following code to the first cell and run it:

```
import pandas as pd
```

Similar to using **`np`** for NumPy, it is customary to use **`pd`** as the alias for pandas.

3.3 Creating a pandas Series

To create a pandas Series, we use the **`Series()`** constructor in the pandas library. We can pass array-like structures like Python lists, Python dictionaries, and NumPy arrays to the constructor. Let's look at some examples:

```
list1 = [1, 2, 3, 4, 5]
print(list1, end='\n\n')

series1 = pd.Series(list1)
print(series1, end='\n\n')

series2 = pd.Series(list1, index=['P', 'Q', 'R', 'S', 'T'])
print(series2, end='\n\n')
```

If you run the code above, you'll get the following output:

```
[1, 2, 3, 4, 5]

0    1
1    2
2    3
3    4
4    5
dtype: int64

P    1
Q    2
R    3
S    4
T    5
dtype: int64
```

In the example above, we first create a list called **`list1`**. When we print this list, we get **`[1, 2, 3, 4, 5]`** as the output.

Next, we pass **`list1`** to the **`Series()`** constructor to create a Series and assign the result to a variable called **`series1`**. When we print the value of **`series1`**, we get two columns of values. The column *on the left* is known as the index of the Series.

The individual values in the index are known as labels. For example, the

label for the first element in **`series1`** is **`0`**, the second is **`1`**, and so on.

After printing **`series1`**, we pass **`list1`** to the **`Series()`** constructor again to create another Series. The default index of a Series is a running sequence of numbers. We can change that using the **`index`** parameter.

When creating **`series2`**, we pass **`index=['P', 'Q', 'R', 'S', 'T']`** to the **`Series()`** constructor. As a result, the first element in **`series2`** has a label of **`'P'`**, the second has a label of **`'Q'`**, and so on.

3.4 Creating a pandas DataFrame

Next, let's learn to create a DataFrame. A DataFrame is very similar to a Series, except that it is two-dimensional.

To create a DataFrame, we use the **`DataFrame()`** constructor. Similar to the **`Series()`** constructor, we can pass array-like structures like Python lists, Python dictionaries, and NumPy arrays to this constructor. Let's look at some examples:

```
# Creating from a 2D list
myList = [[1, 2, 3], [4, 5, 6]]
df1 = pd.DataFrame(myList)
print(df1, end='\n\n')

# Creating from a dictionary of lists
myDict = {'A':[1, 2, 3], 'B':[4, 5, 6]}
df2 = pd.DataFrame(myDict)
print(df2, end='\n\n')
```

Here, we create **`df1`** using a 2D list (**`myList`**) and **`df2`** using a dictionary of lists (**`myDict`**).

When we use a 2D list to create a DataFrame, the nested lists in the list form the rows of the DataFrame. In contrast, when we use a dictionary of lists, the lists in the dictionary form the *columns*.

If you run the code above, you'll get the following output:

```
   0  1  2
0  1  2  3
1  4  5  6

   A  B
0  1  4
1  2  5
2  3  6
```

As you can see, a DataFrame is a two-dimensional data structure that comes with row and column labels. For instance, the columns in the last DataFrame above are labeled **`A`** and **`B`**, while the rows are labeled **`0`**, **`1`**, and **`2`**.

We can specify the labels of a DataFrame when creating it. To do that, we use the **`index`** (for row labels) and **`columns`** (for column labels) parameters:

```
myList2 = [[1, 2, 3, 4, 5], [10, 20, 30, 40, 50]]
df3 = pd.DataFrame(myList2, index = ['A', 'B'], columns = ['1st', '2nd',
'3rd', '4th', '5th'])
df3
```

Here, we specify the row labels of **`df3`** as **`['A', 'B']`** and the column labels as **`['1st', '2nd', '3rd', '4th', '5th']`**.

If you run the code above, you'll get the following output:

	1st	2nd	3rd	4th	5th
A	1	2	3	4	5
B	10	20	30	40	50

Figure 3.1: Displaying a DataFrame as a table

Notice that this DataFrame looks different from the previous DataFrames? This is because the last statement in the code snippet above is **`df3`** instead of **`print(df3)`**.

When the last statement in a Jupyter Notebook cell is a variable name, we do not need to use the **`print()`** function to print the variable; Jupyter Notebook does it for us automatically when we run the cell. If the variable is a DataFrame, Jupyter Notebook formats the DataFrame as a table before printing it.

3.5 Using read_csv()

In the previous section, we learned to create a DataFrame using the **`DataFrame()`** constructor. In most machine learning projects, however, we seldom create a DataFrame from scratch. More often than not, the data that we need is stored in a file. We typically use a pandas function to read it.

The examples below are based on a CSV file called *pandasDemo.csv*. To follow along with the examples, you can download the file at https://www.learncodingfast.com/machine-learning.

If you use Anaconda, you need to make sure the CSV file is in the same folder as your IPYNB file.

If you use Google Colab, you need to upload the file to session storage. To do that, click on the "Files" tab *on the left*, followed by the "Upload to session storage" icon on top. You may need to wait for the session to load before the icon appears. Navigate to where the CSV file is stored on your local drive and upload it to Google Colab. You need to do this every time you get disconnected from Google Colab.

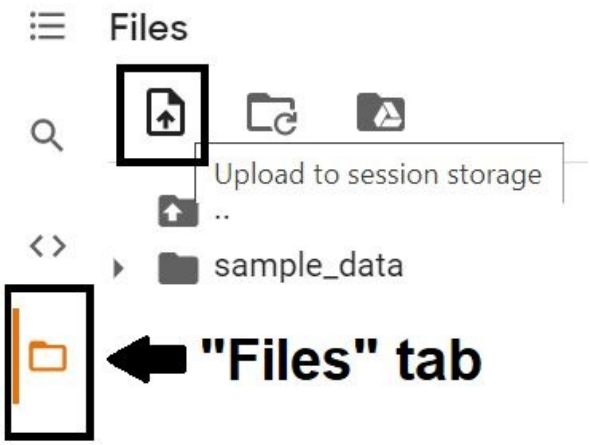

Figure 3.2: Uploading files to Google Colab

To read the *pandasDemo.csv* file, we use the **`read_csv()`** method in pandas. This method accepts the path of our file as an argument and

reads the file into a DataFrame:

```
classData = pd.read_csv('pandasDemo.csv')
```

The statement above reads *pandasDemo.csv* into a DataFrame called **`classData`**. We'll be working with this DataFrame for the rest of the chapter.

3.6 Exploring data in a DataFrame

When we first load data into a DataFrame, we typically want to have a quick look at the data. This can be done using the **`head()`** method, which shows the first five rows of a DataFrame. If we want more than five rows, we pass an integer to the method. For instance, to get the first six rows in **`classData`**, we write:

```
classData.head(6)
```

	ModuleID	Instructor	TA	Enrolment	Rating
0	CS101	Aaron	Vera	38.0	9.0
1	CS101	Beth	Carol	NaN	9.0
2	CS101	Calvin	Andy	28.0	7.0
3	CS102	Dan	Zidane	39.0	6.0
4	CS102	Aaron	Andy	32.0	NaN
5	CS102	Beth	Vera	NaN	7.0

Figure 3.3: Displaying the first six rows of a DataFrame

We can also generate some descriptive statistics for the *numerical* columns in our dataset using the **`describe()`** method:

```
classData.describe()
```

	Enrolment	Rating
count	9.000000	8.000000
mean	34.000000	7.875000
std	4.873397	1.125992
min	28.000000	6.000000
25%	30.000000	7.000000
50%	32.000000	8.000000
75%	39.000000	9.000000
max	40.000000	9.000000

Figure 3.4: Using the describe() method

3.7 Changing labels of row(s) and column(s)

Next, we can change the row and column labels of our DataFrame using the **`rename()`** method. To do that, we pass the **`index`** and **`columns`** parameters to the method:

```
classData = classData.rename(columns={'TA':'Assistant'}, index={0:'Row
Zero',1:'Row One'})
classData.head()
```

The code above changes the label of the **`TA`** column to **`'Assistant'`**, and the labels of rows 0 and 1 to **`'Row Zero'`** and **`'Row One'`**, respectively.

By default, the **`rename()`** method (and most of the other methods in pandas) does not change the DataFrame directly. Instead, it returns a new DataFrame. When this happens, we say that the change is not in place (i.e., the original DataFrame is not modified).

If we want **`classData`** to have the new labels, we need to assign the returned DataFrame back to **`classData`**, which we did in this example. If you run the code above, you'll get the following output:

	ModuleID	Instructor	Assistant	Enrolment	Rating
Row Zero	CS101	Aaron	Vera	38.0	9.0
Row One	CS101	Beth	Carol	NaN	9.0
2	CS101	Calvin	Andy	28.0	7.0
3	CS102	Dan	Zidane	39.0	6.0
4	CS102	Aaron	Andy	32.0	NaN

Figure 3.5: Changing the labels of a DataFrame

3.8 Selecting Data from a DataFrame

3.8.1 Selecting Columns

We can select one or more columns from our DataFrame and assign the result to a variable. To select one column, we use the column's label; this gives us a *pandas Series*:

```
X1 = classData['ModuleID']
print(type(X1))
```

The example above selects the **`ModuleID`** column as a Series and assigns it to **`X1`**. When we print the data type of **`X1`**, we get the following output:

```
<class 'pandas.core.series.Series'>
```

If we want to *select a column as a DataFrame or select more than one column*, we use a list of labels.

```
X2 = classData[['ModuleID']]
X3 = classData[['ModuleID', 'Instructor']]
print(type(X2))
print(type(X3))
X3.head()
```

Here, we use the list **`['ModuleID']`** to select **`ModuleID`** as a DataFrame and assign the result to **`X2`**. Next, we use the list **`['ModuleID', 'Instructor']`** to select **`ModuleID`** and **`Instructor`** as a DataFrame and assign the result to **`X3`**.

In both examples, note the use of two sets of square brackets. If you run the code above, you'll get the following output:

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
```

	ModuleID	Instructor
Row Zero	CS101	Aaron
Row One	CS101	Beth
2	CS101	Calvin
3	CS102	Dan
4	CS102	Aaron

Figure 3.6: Selecting columns as a DataFrame

3.8.2 Selecting Rows

Using Boolean Arrays

To select rows, we can use boolean arrays. Suppose we want to select all the rows in **`classData`** where **`ModuleID`** equals **`'CS101'`**, the example below shows how to do it:

```
arr1 = classData['ModuleID'] == 'CS101'
print(arr1)
```

Here, we first create a boolean condition (**`classData['ModuleID'] == 'CS101'`**) and assign it to a variable called **`arr1`**. Next, we print the value of **`arr1`**.

This gives us the following boolean array, which is a pandas Series:

```
Row Zero     True
Row One      True
2            True
3           False
4           False
5           False
6           False
7           False
8           False
9           False
10          False
```

```
Name: ModuleID, dtype: bool
```

The first three values in this Series are **`True`** because the corresponding rows in **`classData`** satisfy the boolean condition **`classData['ModuleID'] == 'CS101'`**.

To use the Series to select rows, we write:

```
classData[arr1]
```

This gives us a DataFrame that consists of the first three rows in **`classData`**:

	ModuleID	Instructor	Assistant	Enrolment	Rating
Row Zero	CS101	Aaron	Vera	38.0	9.0
Row One	CS101	Beth	Carol	NaN	9.0
2	CS101	Calvin	Andy	28.0	7.0

Figure 3.7: Using boolean arrays to select rows

If we want to use more than one condition to select rows, we can combine them using the **`&`** or **`|`** operators, which represent “and” and “or”, respectively:

```
arr1 = classData['ModuleID'] == 'CS101'
arr2 = classData['Instructor'] == 'Aaron'
classData[arr1 & arr2]
```

In the example above, we create two boolean conditions and assign them to **`arr1`** and **`arr2`**. We then combine the conditions using the **`&`** operator and use the resulting array to select rows from **`classData`**.

If you run this example, you’ll get all the rows in **`classData`** where **`ModuleID`** equals **`'CS101'`** *and* **`Instructor`** equals **`'Aaron'`**. In other words, you’ll get the first row as this is the only row that satisfies both conditions.

If we do not want to assign the boolean conditions above to variables, we can use the statement below:

```
classData[(classData['ModuleID'] == 'CS101') & (classData['Instructor']
== 'Aaron')]
```

If we do it this way, we need to enclose the conditions in parenthesis ().

Using iloc[] or loc[]

Besides using boolean arrays to select rows, we can use **`iloc[]`** or **`loc[]`**.

`iloc[]` is primarily for selecting by position (i.e., by row number starting from 0) while **`loc[]`** is for selecting by row label. Similar to selecting columns, we can select a row as a pandas Series or DataFrame.

`classData.iloc[0]` and **`classData.loc['Row Zero']`** select the first row as a pandas Series using the row number 0 and label **`'Row Zero'`**, respectively.

If we want to select a row as a DataFrame or select more than one row, we use a list.

`classData.iloc[[0]]` and **`classData.loc[['Row Zero']]`** select the first row as a DataFrame.

`classData.iloc[[0, 3, 4]]` and **`classData.loc[['Row Zero', 3, 4]]`** select the first, fourth and fifth rows as a DataFrame.

We can also select rows as a DataFrame using the slice notation. It is easier to slice using row numbers as these slices work the same way as the usual Python slices. For instance, **`classData.iloc[0:3]`** selects the first three rows in **`classData`** as a DataFrame.

3.8.3 Selecting Rows and Columns

Next, we can select rows and columns. To do that, we specify the row(s) that we want, followed by a comma and the column(s) that we want.

For instance, **`classData.iloc[0:5, 0:2]`** selects rows 0 to 4 and columns 0 to 1, as shown in the DataFrame below:

	ModuleID	Instructor
Row Zero	CS101	Aaron
Row One	CS101	Beth
2	CS101	Calvin
3	CS102	Dan
4	CS102	Aaron

Figure 3.8: Selecting columns and rows

`classData.iloc[:, 2]`, on the other hand, selects column 2 from all the rows in **`classData`**.

This is because default values for the start and end of a slice are 0 and the array's length, respectively. Therefore, the slice before the comma (`:`) selects all the rows in **`classData`**, while the index after it selects column 2.

Column 2 is selected as a Series because we used `2` instead of `[2]`. If we want to select it as a DataFrame instead, we write **`classData.iloc[:, [2]]`** or **`classData.iloc[:, 2:3]`**.

Last but not least, **`classData.iloc[:, :-1]`** selects all the columns except the last column from all the rows in **`classData`**. This is because the slice after the comma (`:-1`) selects columns from index 0 to -1, including column 0 but excluding column -1 (which is the last column). Therefore, we get a DataFrame with all the rows and columns, except for the last column.

3.9 Updating a DataFrame

Next, let's learn to update our DataFrame. We can update a DataFrame by adding a column to it or by updating existing columns.

The statement below adds a **`AverageGPA`** column to the **`classData`** DataFrame:

```
classData['AverageGPA'] = [44, 46, 47, 41, 45, 49, 40, 41, 45, 48, 42]
```

The next statement updates the column by assigning **`classData['AverageGPA']*0.1`** to it:

```
classData['AverageGPA'] = classData['AverageGPA']*0.1
```

If you display the first five rows of **`classData`** now, you'll get the following output:

	ModuleID	Instructor	Assistant	Enrolment	Rating	AverageGPA
Row Zero	CS101	Aaron	Vera	38.0	9.0	4.4
Row One	CS101	Beth	Carol	NaN	9.0	4.6
2	CS101	Calvin	Andy	28.0	7.0	4.7
3	CS102	Dan	Zidane	39.0	6.0	4.1
4	CS102	Aaron	Andy	32.0	NaN	4.5

Figure 3.9: Updating a DataFrame

3.10 Useful Methods in pandas

We've covered a lot in this chapter so far. Let's end the chapter by discussing some useful methods in pandas.

sort_values()

The first is the **`sort_values()`** method. This method sorts a DataFrame in ascending order by default. If you want to sort in descending order, you need to pass **`ascending = False`** to the method. Let's look at an example:

```
classData.sort_values(['Rating', 'AverageGPA'], ascending=False)
```

This sorts **`classData`** in descending order using the **`Rating`** column. If there are multiple rows with the same value for **`Rating`**, the method sorts the rows further using the **`AverageGPA`** column.

The **`sort_values()`** method does not modify the original DataFrame (i.e., the sorting is not in place). Instead, it returns a DataFrame with the sorted values.

isnull()

`isnull()` returns a DataFrame where missing values in the original DataFrame get mapped to **`True`**, while other values get mapped to **`False`**.

In pandas, missing values include both the **`None`** keyword (without quotes) and the NumPy **`NaN`** value (including the equivalent **`nan`** and **`NAN`** values). An empty string (**`''`**), the string **`'None'`**, and the number **`0`**, on the other hand, are not considered missing values.

Let's look at an example of the **`isnull()`** method:

```
import numpy as np
myData = pd.DataFrame([[None], [np.NaN], [''], ['Apple']], columns =
['A'])
print(myData.isnull())
```

Here, we create a DataFrame (**myData**) with one column (**A**) and four rows; the values for the first two rows are missing. Next, we use **myData** to call the **isnull()** method. If you run the code above, you'll get the following output:

```
       A
0   True
1   True
2  False
3  False
```

The values for the first two rows in **myData.isnull()** are **True** because the values for the first two rows in **myData** are missing.

We can find the total number of missing values in **myData** using **myData.isnull().sum()**.

Recall that **True** is equal to 1 in Python. Hence, when we sum **myData.isnull()**, we get the number of **True** values in **myData.isnull()**, which corresponds to the number of missing values in **myData**.

If you run the statement

```
myData.isnull().sum()
```

you'll get the following output:

```
A    2
dtype: int64
```

This tells us that there are 2 missing values in column **A** for **myData**. Referring back to our **classData** DataFrame, if we run the statement

```
classData.isnull().sum()
```

we'll get the following output:

```
ModuleID      0
Instructor    0
Assistant     0
Enrolment     2
Rating        3
AverageGPA    0
```

```
dtype: int64
```

This tells us that the **`Enrolment`** and **`Rating`** columns have 2 and 3 missing values, respectively.

dropna()

Most machine learning algorithms do not work well with missing values. Therefore, we need to handle these values. For instance, we can choose to delete them. To do that, we use the **`dropna()`** method.

By default, this method deletes rows with missing values. If we want to delete columns instead, we need to pass **`axis=1`** to the method. The first statement below drops all the rows with missing values, while the second statement drops all the columns with missing values:

```
classData_rows_deleted = classData.dropna()
classData_columns_deleted  = classData.dropna(axis=1)
```

In both cases, the **`dropna()`** method does not change the original DataFrame. Instead, it returns a new DataFrame. Therefore, we need to assign the resulting DataFrame to a variable, as shown in the examples above.

to_numpy()

Next, we have the **`to_numpy()`** method. As the name suggests, this method converts a pandas Series or DataFrame to a NumPy array.

For instance, the example below converts **`myData2`** to a NumPy array.

```
myData2 = pd.DataFrame([[1, 2], [3, 4]], columns = ['A', 'B'])
myArr = myData2.to_numpy()

print(type(myData2))
print(myData2)
print(type(myArr))
print(myArr)
```

If you run the code above, you'll get the following output:

```
<class 'pandas.core.frame.DataFrame'>
   A  B
0  1  2
```

```
1  3  4
<class 'numpy.ndarray'>
[[1 2]
[3 4]]
```

corr()

Last but not least, we have the **`corr()`** method. This method gives us the pairwise correlation coefficients of columns in a DataFrame.

Correlation is a statistical measure that indicates the extent to which two variables are related. If two variables move in the same direction, they have a positive correlation. In contrast, if they move in opposite directions, they have a negative correlation.

Correlation coefficients range from -1 to 1, with -1 indicating a perfect negative correlation, 0 indicating no correlation, and 1 indicating a perfect positive correlation.

A perfect positive correlation occurs when an increase in one variable coincides with an increase of a fixed amount in the other. For instance, if we have the following DataFrame:

```
myData3 = pd.DataFrame({'A':[1, 4, 7, 10], 'B':[1, 2, 3, 4], 'C':[2, 12,
1, 5]})
```

Columns **`A`** and **`B`** have a perfect positive correlation as **`A`** increases by 3 units whenever **`B`** increases by 1. (However, note that this correlation does not imply that **`A`** causes **`B`** or vice versa.) Some machine learning algorithms do not work well when features are strongly correlated with each other. We can use the **`corr()`** method to check if this occurs:

```
myData3.corr()
```

This gives us the table of correlation coefficients below:

	A	B	C
A	1.000000	1.000000	-0.051988
B	1.000000	1.000000	-0.051988
C	-0.051988	-0.051988	1.000000

Figure 3.10: Correlation coefficients between columns

This table is symmetrical about the diagonal. Hence, we can just focus on the coefficients in the triangle.

We can see that **`A`** and **`B`** are perfectly correlated (refer to the underlined value), while **`C`** is not correlated with either of them. As such, we can drop either column **`A`** or **`B`** from our dataset.

Chapter 4 - Matplotlib

In the last two chapters, we learned to work with data stored in NumPy arrays, pandas Series, and pandas DataFrames. In this chapter, let's learn to plot charts for our data.

There are many Python libraries for plotting charts. The popular ones include Matplotlib, Seaborn, and Plotly. This chapter focuses on the Matplotlib library. We'll also briefly discuss some plotting methods in the pandas library that rely on Matplotlib in the background.

4.1 What is Matplotlib?

Matplotlib is a free and open-source plotting library for the Python programming language. It is one of the most popular libraries for data visualization and provides us with two interfaces for plotting graphs - the object-oriented (OO) and pyplot interface.

The OO interface allows us to work directly with Matplotlib's objects (such as the **`Figure`** and **`Axes`** objects) and gives us greater control over the designs of our charts. The pyplot interface, on the other hand, is designed to emulate a popular plotting software called MATLAB and is easier to use.

The pyplot interface is more popular. Therefore, this chapter focuses on the pyplot interface. In the last part of the chapter, we'll briefly discuss the OO interface and learn about the added flexibility it offers.

4.2 Using matplotlib.pyplot

To use the pyplot interface, we need to import the **`matplotlib.pyplot`** module. Create a new notebook called *Chapter 4 - Matplotlib.ipynb* and run the following commands:

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

Here, we first import the NumPy and pandas libraries. Next, we import the **`matplotlib.pyplot`** module using the **`plt`** alias and add the line **`%matplotlib inline`** to specify the backend.

The backend is responsible for the *actual rendering* of the chart.

Matplotlib allows us to specify the structure of our chart (such as whether we want to plot a scatter plot or a bar chart, what data to use, what title it should have, and so on) but does not generate the chart itself.

To generate the chart, it relies on a user-specified backend. When we write **`%matplotlib inline`**, we specify that we want Matplotlib to use Jupyter's own backend to generate the chart. With this backend, the chart is plotted within the notebook itself.

4.2.1 Plotting a Scatter Plot

Now, let's learn to plot a scatter plot. A scatter plot helps show the relationship between two variables and can assist us in narrowing down the types of machine learning models to build.

To plot a scatter plot, we use the **`scatter()`** function in Matplotlib. Most functions in Matplotlib accept 1D array-like structures as input. In the examples below, we use Python lists to illustrate. Besides Python lists, we can use NumPy arrays or pandas Series.

```
sp_x = [2, 3, 4, 6, 2, 8, 6, 9, 12, 1, 7, 1]
sp_y = [10, 3, 4, 12, 5, 12, 4, 5, 6, 8, 9, 10]
```

```
plt.scatter(sp_x, sp_y)
plt.show()
```

Here, we declare and initialize two lists **`sp_x`** and **`sp_y`** for the x and y values of the scatter plot, respectively. Next, we pass the two lists to the **`scatter()`** function and call the **`show()`** function to display the chart.

Calling **`show()`** is optional in a Jupyter Notebook as Jupyter automatically displays the plot when we execute the code. Therefore, we'll omit this command in subsequent examples. If you are not using Jupyter Notebook and the chart does not display, you'll need to use the **`show()`** function.

If you run the code above, you'll get the following plot:

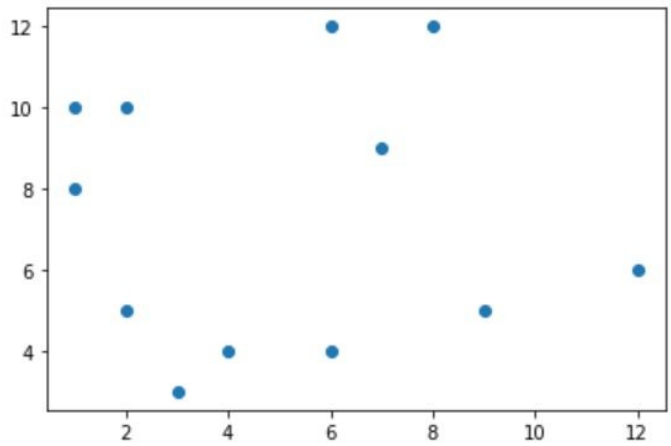

Figure 4.1: Scatter plot

4.2.2 Plotting a Bar Chart

Next, let's move on to bar charts. To plot a bar chart, we use categorical data.

```
b_x = ['0-3', '4-6', '7-9']
b_y = [20, 50, 30]
plt.bar(b_x, b_y)
```

In the code above, we declare two lists - **`b_x`** and **`b_y`**.

`b_x` consists of three categories (**`'0-3'`**, **`'4-6'`**, and **`'7-9'`**) and **`b_y`** consists of some measured values corresponding to the three categories.

For instance, if **`b_x`** represents age groups, **`b_y`** may represent the number of children in each group.

To plot a bar chart, we pass **`b_x`** and **`b_y`** to the **`bar()`** function for the x and y values, respectively. If you run the code above, you'll get the following chart:

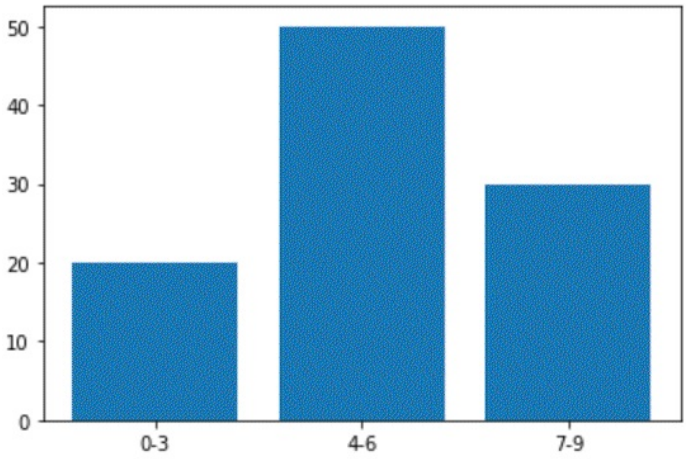

Figure 4.2: Bar chart

4.2.3 Plotting a Histogram

A histogram looks similar to a bar chart but is more suitable for non-categorical data. It is commonly used to show frequency distributions.

```
h_x = [7, 7, 7, 1, 1, 0, 0, 4, 5, 5, 6, 6, 8, 9, 9, 10]
plt.hist(h_x, bins=5)
```

In the example above, we declare a list **`h_x`** with values from 0 to 10. Next, we use the **`hist()`** function to plot a histogram, specifying the number of bins as 5. With **`bins = 5`**, the range of **`h_x`** (0 to 10) is divided into 5 equal-width bins.

The first bin is from 0 to 2 (including 0 but excluding 2), the second is from 2 (inclusive) to 4 (exclusive), and so on, while the last bin is from 8 (inclusive) to 10 (inclusive). The height of each bar represents the frequency (i.e., the number of elements within each interval).

The code above gives us the following histogram:

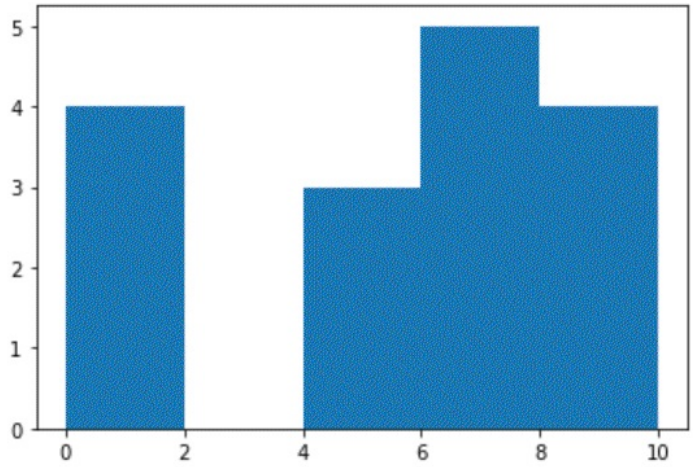

Figure 4.3: Histogram

Specifying the number of bins is optional when plotting a histogram in Matplotlib. In this example, if we do not specify the number of bins, the function plots one bar for each number from 0 to 10.

4.2.4 Plotting a Line Graph

To plot a line graph, we use the **`plot()`** function. Suppose we have the following lists:

```
l_x = [7, 1, 4, 8, 5, 2, 3]
l_y = [98, 2, 32, 128, 15, 28, 18]
```

If we pass these two lists to the **`plot()`** function:

```
plt.plot(l_x, l_y)
```

we'll get the following graph:

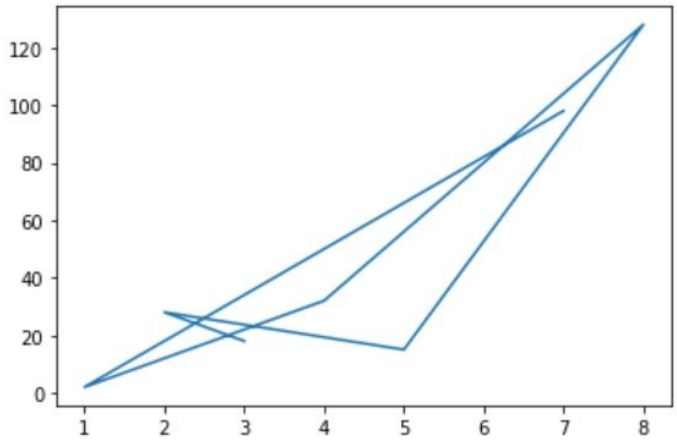

Figure 4.4: Line graph with unsorted points

The **`plot()`** function joins the points in the graph based on the order of the elements in the first list (which represent the x values). In the code above, the first list is **`[7, 1, 4, 8, 5, 2, 3]`**. Hence, the first point for the graph is at x = 7. This point is joined to the second point at x = 1, followed by the third point at x = 4 and so on, resulting in a weird looking graph.

To plot the graph correctly, we need to sort the points before passing them to the function. This can be done using the code below:

```
zipped = zip(l_x,l_y)
sorted_zip = sorted(zipped)
l_x, l_y = zip(*sorted_zip)
```

Here, we first pass **`l_x`** and **`l_y`** to a built-in Python function called **`zip()`**. This function pairs the corresponding elements in **`l_x`** and **`l_y`** as tuples and returns a zip object, which we assign to a variable called **`zipped`**.

`zipped` consists of the following tuples: **`(7, 98)`**, **`(1, 2)`**, **`(4, 32)`**, **`(8, 128)`**, **`(5, 15)`**, **`(2, 28)`**, and **`(3, 18)`**, which are not sorted.

We pass **`zipped`** to the Python **`sorted()`** function to sort the tuples and assign the resulting list - **`[(1, 2), (2, 28), (3, 18), (4, 32), (5, 15), (7, 98), (8, 128)]`** - to a variable called **`sorted_zip`**.

The tuples are now sorted, and we need to “unzip” them. To do that, we

use the **`zip()`** function again. However, this time we pass the sorted list to the **`zip()`** function using the **`*`** operator.

As a result, the **`zip()`** function returns two tuples, which we assign back to **`l_x`** and **`l_y`**. If you print the values of **`l_x`** and **`l_y`** now, you'll get the following output:

```
(1, 2, 3, 4, 5, 7, 8)
(2, 28, 18, 32, 15, 98, 128)
```

We can now pass **`l_x`** and **`l_y`** to the **`plot()`** function again:

```
plt.plot(l_x, l_y)
```

This gives us the following line graph:

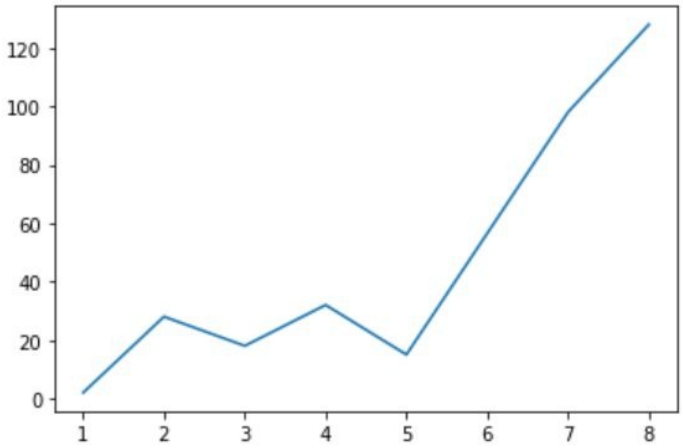

Figure 4.5: Line graph with sorted points

4.3 Using pandas

4.3.1 plot()

In the section above, we learned to plot charts using the pyplot interface. If our data is stored in a pandas Series or DataFrame, in addition to the pyplot interface, we can use the **`plot()`** method *in pandas* to plot our charts.

This method uses Matplotlib in the background by default and is very similar to the pyplot functions discussed above.

However, there are two major differences. Firstly, pyplot functions do not label the charts we plot, but the pandas **`plot()`** method does. Secondly, instead of having separate functions for different types of charts, the pandas **`plot()`** method uses the **`kind`** parameter.

Let's look at an example.

In the "Plotting a Scatter Plot" section above, we used **`plt.scatter(sp_x, sp_y)`** to plot a scatter plot for **`sp_x`** and **`sp_y`**.

To do the same in pandas, we use the code below:

```
sp_x = [2, 3, 4, 6, 2, 8, 6, 9, 12, 1, 7, 1]
sp_y = [10, 3, 4, 12, 5, 12, 4, 5, 6, 8, 9, 10]
df = pd.DataFrame({'A':sp_x, 'B':sp_y})
df.plot(kind='scatter', x='A', y='B')
```

Here, we first create a DataFrame **`df`** with two columns, **`A`** and **`B`**. Next, we use the DataFrame to call the pandas **`plot()`** method, passing **`kind='scatter'`** to specify the chart type, and **`x='A', y='B'`** to specify the columns for the x and y axes, respectively.

If you run this example, you'll get a scatter plot that is very similar to Figure 4.1 above. However, the **`plot()`** method labels the axes of a scatter plot using labels of the columns used to plot the chart. Therefore, this new scatter plot's x and y axes will be labeled "A" and "B",

respectively.

Next, let's look at how we can use the pandas **`plot()`** method to plot a bar chart and a line graph:

Plotting a Bar Chart

```
b_x = ['0-3', '4-6', '7-9']
b_y = [20, 50, 30]
df = pd.DataFrame({'A':b_x, 'B':b_y})
df.plot(kind='bar', x='A', y='B')
```

Plotting a Line Graph

```
l_x = [7, 1, 4, 8, 5, 2, 3]
l_y = [98, 2, 32, 128, 15, 28, 18]
df = pd.DataFrame({'A':l_x, 'B':l_y})
df = df.sort_values(['A'])
df.plot(kind='line', x='A', y='B')
```

These examples should be self-explanatory. We pass **`kind='bar'`** and **`kind='line'`** to the **`plot()`** method to plot a bar chart and a line graph, respectively. To specify the x and y values, we use the **`x`** and **`y`** parameters. For the line graph, we sort the DataFrame using the x values before calling the **`plot()`** method.

If you run the examples, you'll get a bar chart that looks very similar to Figure 4.2 and a line graph that looks very similar to Figure 4.5.

Finally, let's learn to plot a histogram using the pandas **`plot()`** method. For histograms, we do not specify the **`x`** and **`y`** parameters. Instead, we use the column (**`df['A']`**) to call the method:

```
h_x = [7, 7, 7, 1, 1, 0, 0, 4, 5, 5, 6, 6, 8, 9, 9, 10]
df = pd.DataFrame({'A':h_x})
df['A'].plot(kind='hist', bins=5)
```

This gives us a histogram that is very similar to Figure 4.3.

4.3.2 hist()

Besides the **`plot()`** method, pandas comes with another useful method - **`hist()`** - for plotting histograms. This method plots a histogram for

every numerical column in a DataFrame. Let's look at an example:

```
df = pd.DataFrame({'A':[2, 3, 1, 1, 4, 4], 'B':[3, 4, 4, 1, 2, 2]})
df.hist()
```

This example gives us the following output, which shows a histogram on the left for column **`A`** and another on the right for column **`B`**:

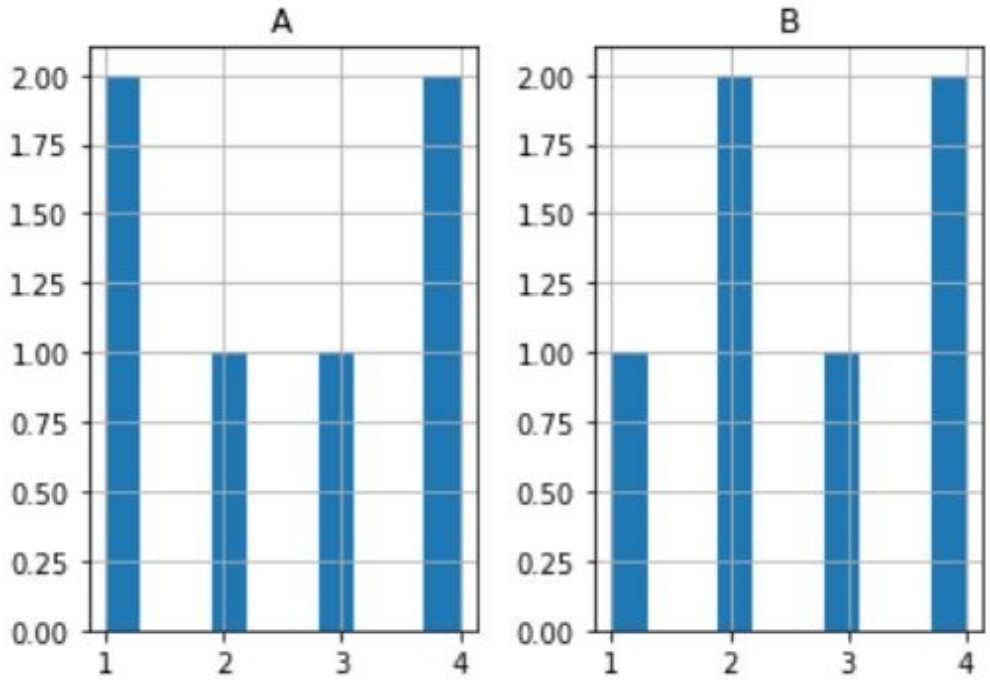

Figure 4.6: pandas hist() method

4.4 Designing our Charts

In the previous sections, we learned to plot basic charts without any customization. In this section, we'll learn to design our charts. For the sake of brevity, we'll only show how to do it using Matplotlib; pandas works similarly and has the same parameters in most cases.

To design our charts in Matplotlib, we need to pass additional parameters to the plotting functions.

To set the color of our chart, we pass the **`color`** parameter. The list of strings (e.g., "blue", "red", and "green") and other formats supported for this parameter can be found at https://matplotlib.org/stable/gallery/color/named_colors.html and https://matplotlib.org/stable/tutorials/colors/colors.html, respectively.

For scatter plots and line graphs, we can specify the marker using the **`marker`** parameter. Commonly used markers include **`'x'`** for crosses and **`'o'`** for circles. You can find the list of acceptable markers at https://matplotlib.org/stable/api/markers_api.html.

To set the size of the markers, we use the **`s`** parameter for scatter plots and the **`ms`** parameter for line graphs. In both cases, we assign numbers to these parameters.

Besides color, marker, and marker size, we can specify a label for our data using the **`label`** parameter. This parameter is useful if we want to plot multiple sets of data on the same chart. We'll see an example of this later.

Next, we can set the title, x and y-axis labels, and tick values on our axes using the **`title()`**, **`xlabel()`**, **`ylabel()`**, **`xticks()`**, and **`yticks()`** functions, respectively. The first three functions accept strings as input, while the last two accept lists. We can also add a legend and grid lines to our chart using the **`legend()`** and **`grid()`** functions, respectively.

Last but not least, we can specify the size of our chart. To do that, we

use the **`figure()`** function to create a **`Figure`** object (we'll discuss what a **`Figure`** object is in the next section) and pass the **`figsize`** parameter to set the width and height of the figure in inches.

Let's look at an example:

```
x1 = [1, 2, 4, 6, 8, 9, 10]
y1 = [3, 4, 6, 12, 4, 5, 7]
y2 = [5, 1, 2, 6, 8, 9, 1]

plt.figure(figsize=(10, 5))

plt.scatter(x1, y1, color='black', s=50, marker='x', label='With
Reward')

plt.scatter(x1, y2, color='darkgray', s=40, marker='o', label='Without
Reward')

plt.legend(loc='best')
plt.grid()

plt.xticks([0, 2, 4, 6, 8, 10])
plt.yticks(range(0, 13))

plt.xlabel('Age')
plt.ylabel('Number of Tries')

plt.title("Designing Our Charts")
```

Here, we first initialize three lists - **`x1`**, **`y1`**, and **`y2`**.

Next, we call the **`figure()`** function to create a new **`Figure`** object and specify the figure size as (10, 5). This means the figure will have a width of 10 inches and a height of 5 inches.

Next, we call the **`scatter()`** function twice to generate two scatter plots *on the same chart*. For the first scatter plot (**`y1`** vs. **`x1`**), we use **`'x'`** as the marker, with a marker size of 50. We also specify the color as **`'black'`** and label it as **`'With Reward'`**.

For the second scatter plot (**`y2`** vs. **`x1`**), we use **`'o'`** as the marker, with a marker size of 40. We specify the color as **`'darkgray'`** and label it as **`'Without Reward'`**.

After specifying the scatter plots, we call the **`legend()`** function to

display the legend. This function uses the labels of our scatter plots to display a legend on our chart. We pass **`loc='best'`** to specify the location of the legend.

Acceptable locations include **`'upper left'`**, **`'upper right'`**, **`'lower left'`**, **`'lower right'`**, **`'upper center'`**, **`'lower center'`**, **`'center left'`**, **`'center right'`**, and **`'best'`**.

`'best'` places the legend at the location with the least amount of overlap with other elements on the chart. After calling the **`legend()`** function, we call the **`grid()`** function to add grid lines to our chart.

Next, we call the **`xticks()`** and **`yticks()`** functions to specify the markings for the x and y axes, respectively, and the **`xlabel()`** and **`ylabel()`** functions to label the axes. Last but not least, we call the **`title()`** function to set the title of the chart. If you run the code above, you'll get the following chart:

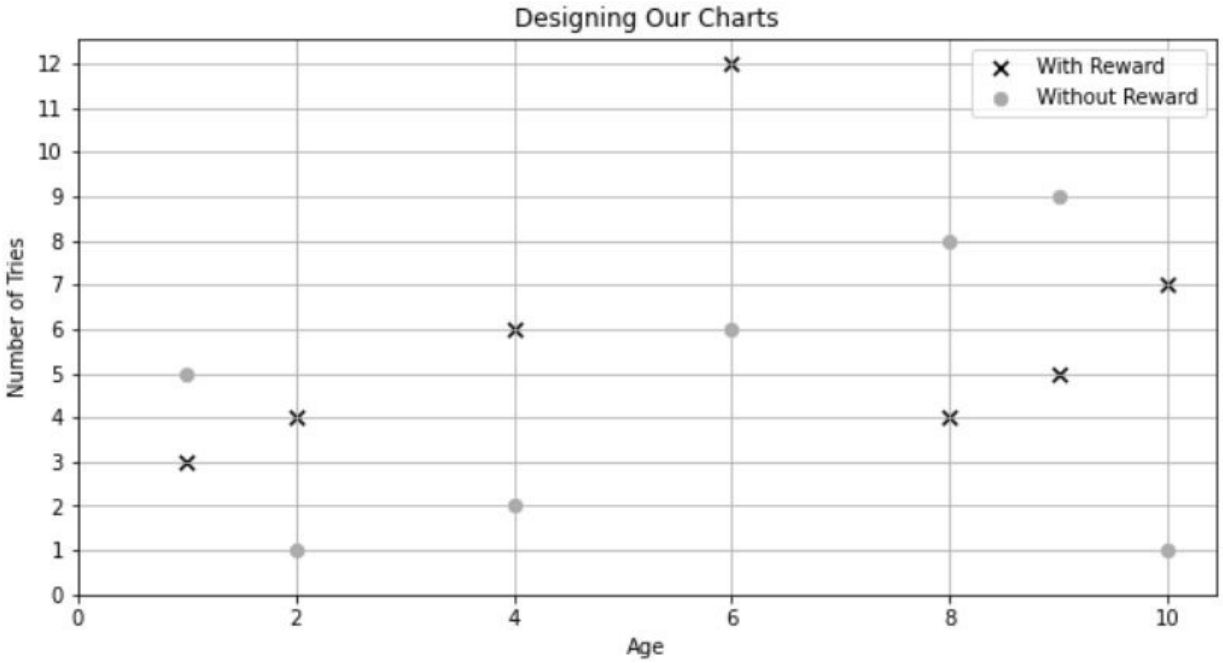

Figure 4.7: Designing our charts

Note that this chart has been scaled to fit the book's width and does not have the figure size specified.

4.5 The Object-Oriented Interface

So far, we've discussed how to plot charts in Matplotlib using the pyplot interface. While this interface is convenient and easy to use, it offers less control over our charts. For instance, when plotting multiple sets of data, it is harder to state whether we want to plot the data on one chart or on multiple separate charts.

In this section, we'll discuss the object-oriented interface, which offers more flexibility and control over the layout of our charts. To understand how this interface works, we need to first discuss the anatomy of a figure in Matplotlib.

There are two important objects in Matplotlib - the **`Figure`** and **`Axes`** objects.

A **`Figure`** object (also referred to as a figure) is the top-level container of a plot. Within each figure, we can have multiple **`Axes`** objects. You can think of an **`Axes`** object as an individual plot inside a **`Figure`** object.

As an illustration, the diagram below shows one **`Figure`** object (with gray background) and two **`Axes`** objects (with white backgrounds).

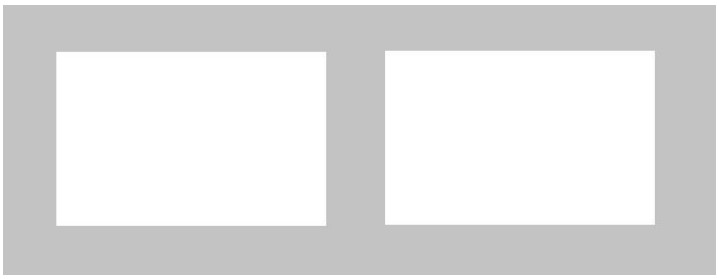

Figure 4.8: Figure and Axes objects

To work with **`Figure`** and **`Axes`** objects, we need to create them. There are many ways to do so, each with its pros and cons. An easy way is to use the **`subplots()`** function in pyplot. This function returns a **`Figure`** object *and* a set of **`Axes`** objects that we can manipulate using their indexes.

We typically pass the number of rows and columns of the subplot grid to the function. For instance, in the diagram above, the number of rows is 1, and the number of columns is 2. If we do not specify the number of rows and columns, the default values are 1. We can also pass the **`figsize`** parameter to the function to set the figure size.

Let's look at some examples:

```
fig, my_ax = plt.subplots()
a = [1, 2, 3, 4]
b = [7, 3, 1, 4]
my_ax.scatter(a, b)
```

In the example above, we call the **`subplots()`** function without specifying the number of rows and columns. Hence, the function uses the default values of 1 and gives us a figure with one row and one column. In other words, we get one **`Figure`** and one **`Axes`** object.

We assign the **`Figure`** object to a variable called **`fig`** and the **`Axes`** object to a variable called **`my_ax`**. Next, we use **`my_ax`** to call the **`scatter()`** function. This gives us the following output:

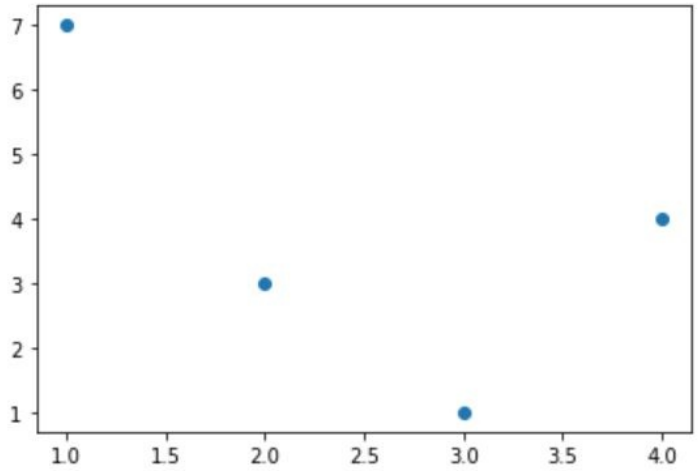

Figure 4.9: Using an Axes object to plot a scatter plot

If we wish to use the pandas **`plot()`** method to plot our chart instead, we need to pass the **`Axes`** object to the method using the **`ax`** parameter. An example is shown below:

```
fig, my_ax2 = plt.subplots()
df = pd.DataFrame({'A':[1, 2, 3, 4], 'B':[7, 3, 1, 4]})
```

```
df.plot(kind='scatter', x='A', y='B', ax=my_ax2)
```

This gives a scatter plot that looks very similar to Figure 4.9, except that its x and y axes are labeled “A” and “B”, respectively.

Next, let’s look at how we can plot two charts on a single figure:

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
df = pd.DataFrame({'A':[1, 2, 3, 4], 'B':[4, 2, 3, 1]})

axs[0].scatter(df['A'], df['B'])
axs[1].plot(df['A'], df['B'])

axs[0].set_title('Chart 1')
axs[0].set_xlabel('Age')
axs[0].set_ylabel('Number of Tries')
axs[0].set_xticks([1, 2, 3, 4])
axs[0].set_yticks([1, 2, 3, 4])
```

Here, we create a figure (**`fig`**) with 1 row, 2 columns, and a figure size of (10, 5).

When a figure has one row *or* one column, the **`Axes`** objects are stored as a 1D array. In the example above, we assign this array to a variable called **`axs`**.

To access the individual objects in the array, we use their indexes. **`axs[0]`** gives us the **`Axes`** object in the first column, while **`axs[1]`** gives the object in the second.

We use **`axs[0]`** to plot a scatter plot and **`axs[1]`** to plot a line graph. Next, we set the title, x and y-axis labels, and tick values for **`axs[0]`**.

To do that in the OO interface, we cannot use the same functions as the pyplot interface. Instead, we need to use the **`set_title()`**, **`set_xlabel()`**, **`set_ylabel()`**, **`set_xticks()`**, and **`set_yticks()`** methods, respectively.

If you run the code above, you’ll get the following output:

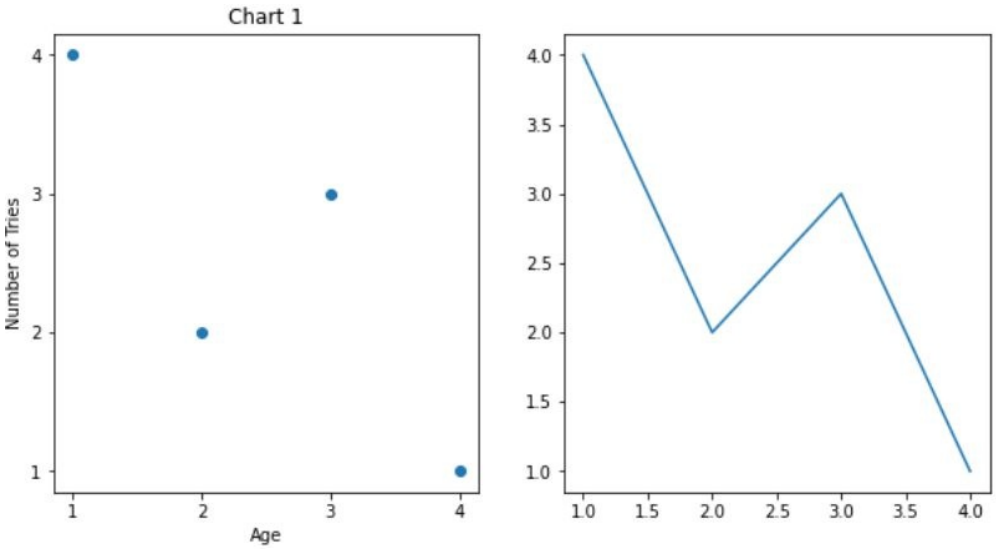

Figure 4.10: Plotting two charts on a figure

Last but not least, let's look at a figure with more than 1 row (or 1 column). Suppose we create a figure with 2 rows and 3 columns:

```
fig, axs = plt.subplots(2, 3)
```

The **`subplots()`** function gives us a 2D array of **`Axes`** objects. The diagram below shows the indexes for the objects:

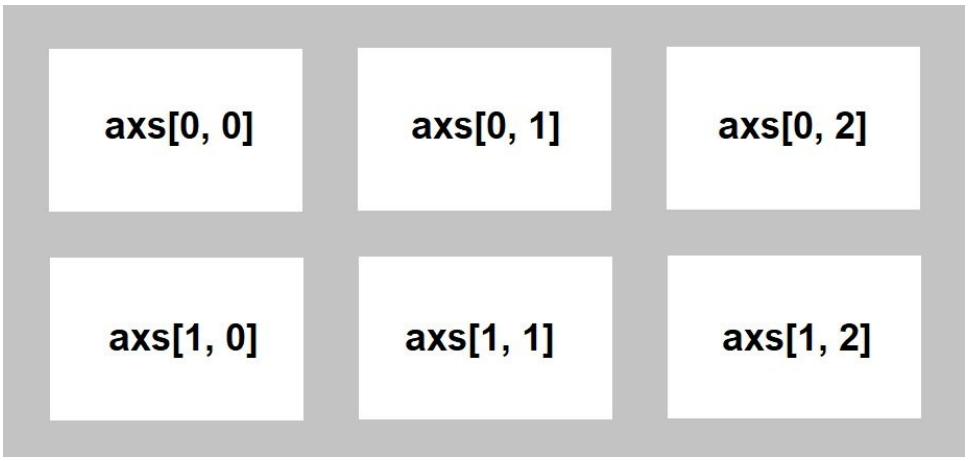

Figure 4.11: Indexes of Axes objects in a figure with two rows and three columns

We'll have a chance to create a figure with two rows and three columns for our project later.

We've covered several important concepts for the Matplotlib library in this

chapter. However, the library offers many more functionalities and variations that we are unable to cover. If you are interested in finding out more, you can find its official documentation at https://matplotlib.org/stable/api/index.html.

Chapter 5 - Scikit-Learn

Congratulations on making it to Chapter 5! We are now almost ready to build our own machine learning models. However, before we do that, we need to discuss one more library - the Scikit-Learn library.

The Scikit-Learn library (also known as the sklearn library) is a free and hugely popular machine learning library for the Python programming language.

The first section of this chapter gives an overview of the library, while subsequent sections show how to use different modules in the library.

5.1 Overview of the Scikit-Learn Library

5.1.1 Organizational Structure

Scikit-learn is a massive Python library that comes with an extensive assortment of classes and functions for performing various machine learning tasks. These classes and functions are grouped into numerous modules that are intuitively named.

For instance, classes for building decision tree-based models are found in the **`sklearn.tree`** module, while classes for data preprocessing are in the **`sklearn.preprocessing`** module.

The classes themselves are also named appropriately. For example, the class for building a linear regression model is called **`LinearRegression`**, while that for scaling a dataset using standardization is called **`StandardScaler`**.

If you are interested in learning about a particular class, you can visit sklearn's online documentation at https://scikit-learn.org/stable/modules/classes.html. Just scroll through the page and click on the link for the relevant class. You'll be directed to a page that lists the class's *parameters*, *attributes*, and *methods*.

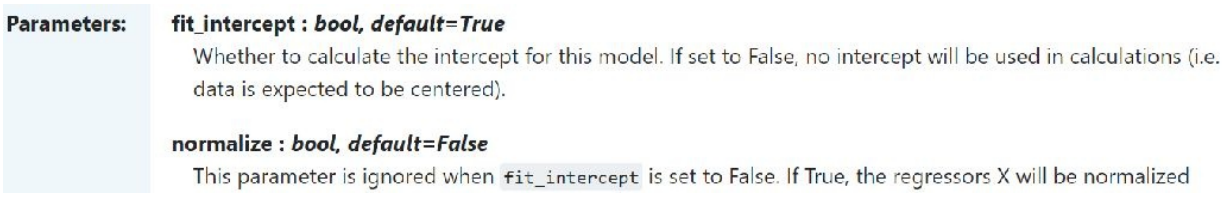

Figure 5.1: Class parameters

The "Parameters" section lists the parameters that you can pass to a class constructor when instantiating an object.

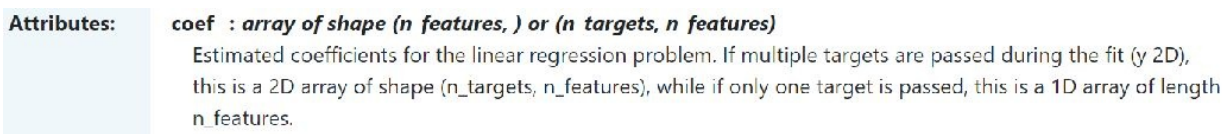

Figure 5.2: Class attributes

The “Attributes” section lists the class attributes, which are for storing *model* parameters.

In machine learning, parameters can refer to the parameters of a method or parameters of a model.

Parameters of a method refer to the list of variables in a method’s declaration. For instance, the “Parameters” section discussed above gives us the parameters of a class constructor (which is a special type of method).

On the other hand, parameters of a model refer to the values learned (i.e., calculated) by an algorithm during the training process. These parameters are stored in the class attributes.

Methods

`fit`(X, y[, sample_weight])	Fit linear model.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model.
`score`(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

Figure 5.3: Class methods

If you scroll further down the page, you’ll see the “Methods” section. This section lists the methods available in the class. You can click on a method name to get a more detailed description of the method.

5.1.2 Estimators, Transformers, and Predictors

sklearn follows an object-oriented approach and is built around three main types of objects: estimators, transformers, and predictors. All estimators have the **`fit()`** method, transformers have the **`transform()`** method, and predictors have the **`predict()`** method.

If you look at sklearn’s documentation, you’ll see that almost all classes implement the **`fit()`** method and either the **`transform()`** or **`predict()`** method.

The implementation of these methods is different for different classes.

For instance, the implementation of the **`fit()`** method in the **`LinearRegression`** class is entirely different from that in the **`StandardScaler`** class. This is because the two methods perform very different tasks, even though they share the same name. We'll see many examples of this later.

Before that, let's first discuss what estimators, transformers, and predictors are.

Estimators and the fit() method

An estimator refers to any object that learns (i.e., estimates or calculates some parameters) from input data; this learning happens inside the **`fit()`** method.

An example is the **`fit()`** method in the **`StandardScaler`** class. This class is for scaling the features in our dataset. Its **`fit()`** method estimates the means and variances of the features in the input dataset and stores the results in the **`mean_`** and **`var_`** attributes, respectively.

Transformers and the transform() method

Some estimators also transform a dataset. These estimators are known as transformers and perform the transformation inside a method called **`transform()`**. This method accepts an array as input and uses the parameters calculated by the **`fit()`** method of the same object to return a transformed dataset.

As an illustration, let's consider the **`transform()`** method in the **`StandardScaler`** class.

The objective of scaling a dataset using the **`StandardScaler`** class is to achieve a mean of 0 and a variance of 1. This process is known as standardization and is necessary for some machine learning algorithms. The example below shows how to do it:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
X = pd.DataFrame({'A':[1, 2, 3, 2], 'B':[11, 1, 8, 3]})
scaler = StandardScaler()
```

```
scaler.fit(X)
print(scaler.mean_)
print(scaler.var_)
```

Here, we first import the **`StandardScaler`** class from the **`sklearn.preprocessing`** module. Next, we create a DataFrame **`X`** with two columns, **`A`** and **`B`**.

To standardize this dataset, we instantiate a **`StandardScaler`** object and use its **`fit()`** method to calculate the means and variances of columns **`A`** and **`B`**, by passing **`X`** to the method.

After calling the **`fit()`** method, the results are stored in the **`mean_`** and **`var_`** attributes of the **`StandardScaler`** object (**`scaler`**). If we run the code above, we'll get the following output:

```
[2.   5.75]
[ 0.5    15.6875]
```

2 and 5.75 are the means of columns **`A`** and **`B`**, respectively, while 0.5 and 15.6875 are the variances. If we want to standardize a value manually, we use the formula:

$$x\_standardized = (x - mean) / (standard\ deviation)$$

(Note: Standard deviation is the square root of variance.)

This calculation can be tedious if we have a lot of values to transform. Fortunately, we do not need to do the math ourselves; we can use the **`transform()`** method in the **`StandardScaler`** class to do it for us.

The **`transform()`** method scales a dataset using the means and variances calculated by the **`fit()`** method and returns a transformed dataset as a *NumPy array*. In the statement below, we use the **`transform()`** method to scale **`X`** and assign the transformed array to a variable called **`X_scaled`**:

```
X_scaled = scaler.transform(X)
```

If you print the value of **`X_scaled`** now, you'll get the following output:

```
[[-1.41421356  1.32550825]
 [ 0.         -1.19926937]
 [ 1.41421356  0.56807496]
 [ 0.         -0.69431384]]
```

The first column in this array gives us the standardized values of column **`A`**, while the second gives us the standardized values of column **`B`**.

Both columns have a mean of 0 and a variance of 1. We can verify that using the **`mean()`** and **`var()`** methods in NumPy:

```
print(X_scaled[:, 0].mean())
print(X_scaled[:, 1].mean())
print(X_scaled[:, 0].var())
print(X_scaled[:, 1].var())
```

This gives us the following output:

```
0.0
0.0
0.9999999999999998
1.0
```

The variance for the first column is not exactly 1 due to the way our computers represent floating-point numbers. Other than that, the **`transform()`** method has successfully standardized the two columns in our DataFrame.

The fit_transform() method

In the example above, we called the **`fit()`** and **`transform()`** methods separately. Alternatively, we can call the **`fit_transform()`** method. Calling this method is equivalent to calling **`fit()`** and then **`transform()`**, but **`fit_transform()`** is often optimized and runs faster than calling the other two methods separately.

Predictors and the predict() method

Finally, some estimators are able to make predictions when given a dataset. These estimators are known as predictors and perform the prediction using a method called **`predict()`**.

This method makes predictions based on the parameters calculated by

the **`fit()`** method. For instance, the **`predict()`** method in the **`LinearRegression`** class uses the coefficients calculated by the **`fit()`** method of the same object to make predictions.

We'll look at this method when we discuss machine learning algorithms in subsequent chapters.

5.2 Data Preprocessing with Scikit-Learn

Next, let's move on to discuss data preprocessing with sklearn. As mentioned in Chapter 1, most of the time, we need to process the data we receive before feeding them to our machine learning algorithms.

Data preprocessing typically involves the following tasks: handling missing data, encoding text and categorical data, and feature scaling.

The following section uses the dataset below to illustrate how to perform data preprocessing with sklearn.

	Color	Years	Strength	Height	Weight	Dangerous
0	Green	2.3	210.0	170.0	20 to 30 kg	Yes
1	Red	4.1	100.0	NaN	10 to 20 kg	No
2	Blue	1.4	NaN	412.0	0 to 10 kg	No
3	Green	NaN	313.0	123.0	10 to 20 kg	Yes
4	NaN	5.2	512.0	372.0	0 to 10 kg	Yes

Figure 5.4: Dataset for demonstrating data preprocessing

This dataset has no specific meaning and is created for the sole purpose of demonstrating how data preprocessing works. The **`Color`**, **`Years`**, **`Strength`**, **`Height`**, and **`Weight`** columns are the features of the dataset, while the **`Dangerous`** column is the target.

Here, we choose to use a DataFrame to explain the concepts as it is easier to visualize modifications with a DataFrame. In subsequent chapters, we'll show how to work with NumPy arrays, which are typically faster than DataFrames.

The examples below assume you've loaded the dataset (stored in the *datapreprocessing.csv* file) into a DataFrame called **`df`** and imported the NumPy and pandas libraries using the statements below:

```
import pandas as pd
import numpy as np
df = pd.read_csv('datapreprocessing.csv')
```

5.2.1 Handling Missing Data

Most machine learning algorithms mandate a complete dataset as missing data can introduce biases and affect the model's accuracy. Missing values in CSV files are typically denoted as **`np.nan`** when we load them using the **`read_csv()`** method.

A simple workaround to handle these missing values is to discard their rows or columns using the **`dropna()`** method in pandas (refer to Chapter 3, Section 3.10).

This approach works well if you have a large dataset and only a small percentage of missing values. If that is not the case, a better approach is to replace the missing data with statistical measures. This process of replacing missing values in an incomplete dataset is known as data imputation.

To perform data imputation in sklearn, we use the **`SimpleImputer`** class in the **`sklearn.impute`** module. Let's look at an example using **`df`**, which has missing values for the **`Color`**, **`Years`**, **`Strength`**, and **`Height`** columns.

The code below replaces the missing values for the **`Years`**, **`Strength`**, and **`Height`** columns:

```
# Importing the SimpleImputer class
from sklearn.impute import SimpleImputer

# Instantiating a SimpleImputer object
imp = SimpleImputer(missing_values=np.nan, strategy='mean')

# Calling the fit() method to calculate the means
imp.fit(df[['Years', 'Strength', 'Height']])

# transforming the data
df[['Years', 'Strength', 'Height']] = imp.transform(df[['Years',
'Strength', 'Height']])
```

Here, we first import the **`SimpleImputer`** class. Next, we instantiate a **`SimpleImputer`** object called **`imp`**, passing two parameters - **`missing_values`** and **`strategy`** - to the constructor.

`missing_values` tells the imputer what to treat as missing values (**`np.nan`** in our example, which is the default), while **`strategy`** tells the imputer what strategy to use to replace the missing values. The commonly used strategies are **`'mean'`**, **`'median'`** and **`'most_frequent'`**, which replace the missing values with the mean, median, and mode of the column, respectively.

After instantiating the **`SimpleImputer`** object, we use it to call the **`fit()`** method, passing **`df[['Years', 'Strength', 'Height']]`** to the method. This method expects a 2-dimensional array and calculates the means of the columns in the array; it stores the results in the **`statistics_`** attribute.

If we print the value of **`statistics_`** now:

```
print(imp.statistics_)
```

we'll get **`[3.25  283.75  269.25]`** as the output.

3.25 is the mean of the first feature (**`Years`**), while 283.75 and 269.25 are the means of the second and third features (**`Strength`** and **`Height`**), respectively.

After calculating the means, we pass **`df[['Years', 'Strength', 'Height']]`** to the **`transform()`** method to get a new dataset with the missing values replaced. This method uses the means stored in **`statistics_`** to do the replacement and returns a transformed dataset, which we assign back to **`df[['Years', 'Strength', 'Height']]`**.

If you print the DataFrame now, you'll get the following output:

	Color	Years	Strength	Height	Weight	Dangerous
0	Green	2.30	210.00	170.00	20 to 30 kg	Yes
1	Red	4.10	100.00	269.25	10 to 20 kg	No
2	Blue	1.40	283.75	412.00	0 to 10 kg	No
3	Green	3.25	313.00	123.00	10 to 20 kg	Yes
4	NaN	5.20	512.00	372.00	0 to 10 kg	Yes

Figure 5.5: Missing values replaced

All the missing values for the three columns have been replaced. Next, let's fill in the missing value for the **`Color`** column; we'll use the **`'most_frequent'`** strategy for this column.

Note that both the **`fit()`** and **`transform()`** methods in the **`SimpleImputer`** class require a 2D array as input. Hence, in the code below, we pass **`df[['Color']]`**, instead of **`df['Color']`** to the methods.

You can tell the shape required by a method from its documentation. If the documentation shows the shape of the parameter as (n_samples, n_features) or (n_samples, n_targets), it requires a 2D array. On the other hand, if it shows (n_samples,), a 1D array is expected.

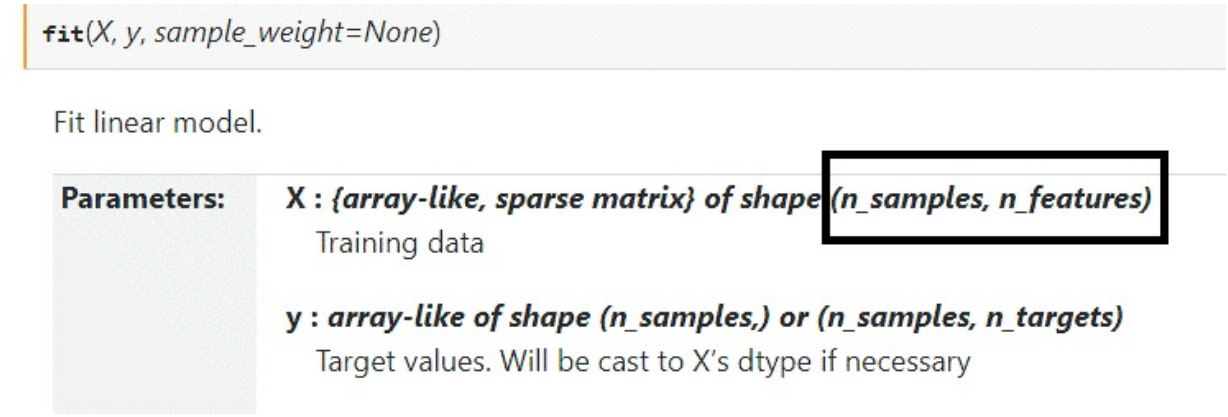

Figure 5.6: Shape of input data required

Let's fill in the missing value for the **`Color`** column now:

```
# Updating the 'strategy' parameter of the SimpleImputer object
imp.set_params(strategy='most_frequent')

# Calling the fit() method to get the mode
imp.fit(df[['Color']])

# Transforming the column
df[['Color']] = imp.transform(df[['Color']])
```

Here, as we are using a different strategy for the **`Color`** column, we need to update the **`strategy`** parameter for **`imp`**. We do that using the **`set_params()`** method in the **`SimpleImputer`** class.

Next, we use **`imp`** to call the **`fit()`** method, passing **`df[['Color']]`** to the method. If we print the value of **`statistics_`** now, we'll get **`['Green']`** as the output.

After calling the **`fit()`** method, we call the **`transform()`** method to transform **`df[['Color']]`** and assign the result back to the **`Color`** column.

If you run the code above, the missing value for the **`Color`** column will be replaced with **`'Green'`**.

In the examples above, I called the **`fit()`** and **`transform()`** methods separately to demonstrate what the two methods do. From this point forward, I'll use the **`fit_transform()`** method directly for the sake of brevity.

In addition, in this section, we use the entire dataset to demonstrate how to do data preprocessing. In an actual machine learning project, you would normally split your dataset into training and test subsets. In that case, you should call the **`fit_transform()`** method on the training set and only call the **`transform()`** method on the test set.

You should treat the test set as if it does not exist during the training stage, as this dataset is supposed to mimic new data collected *after* the model has been developed. Therefore, we should not use it to calculate any parameters for the model, including parameters for data-preprocessing. In other words, we should not pass the test set to the

`fit()` or **`fit_transform()`** method. Else, parameters will be calculated using information from the test set, which is incorrect. We'll have a chance to work with training and test sets in subsequent chapters.

5.2.2 Encoding Categorical Data

Next, let's move on to discuss data encoding.

Frequently, when working on a machine learning project, the incoming dataset might include categorical values that are non-numerical. For example, the **`Color`**, **`Weight`**, and **`Dangerous`** columns in **`df`** have non-numerical values.

Only a small subset of machine learning algorithms can handle non-numerical categorical data. Therefore, we need to convert these categorical text data into numerical data before passing them to our algorithms.

There are a few ways to do that.

One way is to convert each categorical text to a number. For instance, to encode the **`Weight`** column, we can assign a running sequence of numbers to the categories (such as 0 for '0 to 10 kg', 1 for '10 to 20 kg', and so on).

To do that, we can use the **`OrdinalEncoder`** or **`LabelEncoder`** class.

The two classes are very similar, except that **`OrdinalEncoder`** is designed to work with the features of a dataset, while **`LabelEncoder`** is designed to work with the labels. Therefore, the **`fit()`** method of the **`LabelEncoder`** class expects a 1D array, while that of the **`OrdinalEncoder`** class expects a 2D array.

The code below demonstrates how to encode the labels in **`df`** (stored in the **`Dangerous`** column) using the **`LabelEncoder`** class:

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['Dangerous'] = le.fit_transform(df['Dangerous'])
```

Here, we first import the **`LabelEncoder`** class and instantiate a **`LabelEncoder`** object. Next, we use it to call the **`fit_transform()`** method, passing a 1D array (**`df['Dangerous']`**) to the method. This method returns a NumPy array, which we assign back to the **`Dangerous`** column.

Next, let's encode the **`Weight`** column (which is a feature) using the **`OrdinalEncoder`** class:

```
from sklearn.preprocessing import OrdinalEncoder
oe = OrdinalEncoder(dtype=np.int)
df[['Weight']] = oe.fit_transform(df[['Weight']])
```

The **`OrdinalEncoder`** class allows us to specify the data type to use when encoding categories. The default is floating-point numbers (**`np.float64`**). If we want to encode categories as integers instead, we need to pass **`dtype=np.int`** to the constructor, as shown in this example.

You can see the result of the encoding steps above in Figure 5.7 later.

Encoding categories as a running sequence of numbers is a simplistic approach to categorical encoding and can be done easily using the **`LabelEncoder`** and **`OrdinalEncoder`** classes.

However, one problem with such a simple approach is that it suggests a relationship between the categories that may not exist in reality.

For instance, our **`df`** DataFrame has a **`Color`** column with three values – "Green", "Red", and "Blue". The **`OrdinalEncoder`** class encodes categories in alphabetical order. Hence, "Blue" will be encoded as 0, "Green" as 1, and "Red" as 2.

A machine learning algorithm may misinterpret these numbers as suggesting that "Blue" is smaller than "Red" and draw invalid conclusions from the data. Therefore, encoding non-ordinal features (i.e., features where the order does not matter) using the **`OrdinalEncoder`** class is not ideal.

A common approach to encoding non-ordinal features is to use one hot

encoding, where each category value is converted to a new column and assigned a value of 0 or 1. To do one hot encoding in sklearn, we use the **`OneHotEncoder`** class in the **`sklearn.preprocessing`** module.

If you read the documentation for this class, you'll see a constructor parameter called **`sparse`**, which is **`True`** by default. This means the methods in the **`OneHotEncoder`** class return a sparse matrix by default.

A sparse matrix is a matrix that consists of very few non-zero elements. For instance,

[[0, 0, 0, 4, 0],
[0, 1, 0, 0, 0]]

is a sparse matrix as it only has two non-zero elements. Storing this matrix as a regular 2D array is a waste of space since most elements are just 0s. Therefore, sklearn compresses the matrix into a special data structure that only stores the non-zero values.

Most methods in sklearn accept this sparse structure as input. However, if you want to work with a regular 2D array instead, you can pass **`sparse = False`** to the **`OneHotEncoder()`** constructor when initializing an object. When you do that, the **`fit_transform()`** method returns a regular NumPy array.

Let's look at an example:

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(dtype=np.int, sparse = False, drop='first')
color_encoded = ohe.fit_transform(df[['Color']])
df2 = pd.DataFrame(color_encoded, columns = ohe.get_feature_names())
df = pd.concat((df, df2), axis = 1)
df
```

Here, we first import the **`OneHotEncoder`** class from the **`sklearn.preprocessing`** module.

Next, we initialize a **`OneHotEncoder`** object called **`ohe`**. We've explained the **`dtype`** and **`sparse`** parameters previously. However, notice that we also pass **`drop='first'`** to the constructor; we'll explain this parameter later.

After instantiating **ohe**, we use it to call the **fit_transform()** method, passing **df[['Color']]** as input to the method.

Next. we store the resulting array in a variable called **color_encoded** and convert this array to a DataFrame (**df2**) on the next line.

To name the columns in **df2**, we use the **get_feature_names()** method in the **OneHotEncoder** class. This method returns the names of the columns returned by the **fit_transform()** method.

Finally, we add **df2** to our original DataFrame using the **concat()** method in pandas. This method works like the **concatenate()** function in NumPy.

If you run the code now, you'll get the following DataFrame:

	Color	Years	Strength	Height	Weight	Dangerous	x0_Green	x0_Red
0	Green	2.30	210.00	170.00	2	1	1	0
1	Red	4.10	100.00	269.25	1	0	0	1
2	Blue	1.40	283.75	412.00	0	0	0	0
3	Green	3.25	313.00	123.00	1	1	1	0
4	Green	5.20	512.00	372.00	0	1	1	0

Figure 5.7: Categorical columns encoded

Notice that this DataFrame has two additional columns - **x0_Green** and **x0_Red**. Both columns result from the one hot encoding step.

x0_Green equals 1 when **Color** equals 'Green'. The same applies to **x0_Red**, which equals 1 when **Color** equals 'Red'. However, we do not have a **x0_Blue** column. This is due to the **drop='first'** parameter mentioned earlier.

The **x0_Blue** column (which corresponds to the first category - "Blue") is dropped because we can use **x0_Green** equals 0 and **x0_Red** equals 0 to indicate that the color is "Blue". (This happens on the third row.)

Therefore, **`x0_Blue`** is redundant. Dropping this redundant column is necessary for some machine learning algorithms (such as linear regression) as it can affect the model's parameters.

5.2.3 Feature Scaling

Now that we've encoded all the categorical columns, let's move on to feature scaling.

Feature scaling is a technique for calibrating the range of features in a dataset. This ensures that features do not have vastly different scales and prevents features with large values from dominating the results of our algorithms.

For instance, some machine learning algorithms use the Euclidean distance between instances to estimate parameters. If one feature has a range of 0 to 10,000 while another has a range of 0 to 5, the feature with the larger values will dominate the distance calculation.

Suppose we have two instances, A and B.

If the values for the first feature of A and B are 8080 and 8000, respectively (denoted as p and q in the formula below), and the values for the second feature are 2 and 1 (denoted as r and s), the Euclidean distance between the two instances is given by the calculation below:

Euclidean Distance

$$= \sqrt{(p-q)^2 + (r-s)^2}$$

$$= \sqrt{(8080-8000)^2 + (2-1)^2}$$

$$= \sqrt{(80)^2 + (1)^2}$$

Even though the values for the second feature differ by 100% (2 vs. 1), this difference does not contribute much to the distance. In contrast, the values for the first feature (with a difference of only 1% - 8000 vs. 8080) contribute much more to it. This bias occurs because the first feature has

larger values.

To prevent such biases from occurring, we can scale both features. There are two possible options for doing so: normalization and standardization.

Normalization scales a feature to values between 0 and 1 inclusive using the formula

$$x\_normalized = (x - x\_min) / (x\_max - x\_min),$$

where x_max and x_min are the maximum and minimum values of the feature.

Standardization, on the other hand, scales a feature such that the mean becomes 0 and variance 1, as previously discussed when we introduced the **`transform()`** method.

The decision to normalize or standardize our dataset depends on the specifics of the problem. Normalization tends to be affected by outliers (i.e., extreme values) as its computation uses the maximum and minimum values of the features. Standardization, on the other hand, does not bound values to a specific range. This can be a problem for some machine learning algorithms that expect a specific input range. For instance, most neural networks expect an input range of 0 to 1.

In general, there is no hard and fast rule for when you should normalize or standardize your data. You can always train all three datasets (raw, normalized, and standardized) and compare the performance of each dataset.

We've seen how to perform standardization using the **`StandardScaler`** class previously. To perform normalization, we use the **`MinMaxScaler`** class:

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
df[['Years', 'Strength', 'Height']] = mms.fit_transform(df[['Years',
'Strength', 'Height']])
df
```

The code above should require no further explanation. If you run the code, you'll get the following output:

	Color	Years	Strength	Height	Weight	Dangerous	x0_Green	x0_Red
0	Green	0.236842	0.266990	0.162630	2	1	1	0
1	Red	0.710526	0.000000	0.506055	1	0	0	1
2	Blue	0.000000	0.445995	1.000000	0	0	0	0
3	Green	0.486842	0.516990	0.000000	1	1	1	0
4	Green	1.000000	1.000000	0.861592	0	1	1	0

Figure 5.8: Features scaled to a range of 0 to 1

We have successfully scaled **`Years`**, **`Strength`**, and **`Height`** to a range of 0 to 1.

5.3 Pipeline and ColumnTransformer

In the previous sections, we learned to do data preprocessing using different classes in sklearn. We first used the **`SimpleImputer`** class to replace missing values in the dataset before using the **`LabelEncoder`**, **`OrdinalEncoder`**, and **`OneHotEncoder`** classes to encode textual data. Finally, we used the **`MinMaxScaler`** class to scale the **`Years`**, **`Strength`**, and **`Height`** columns.

This process is quite a hassle as we need to call the **`fit_transform()`** methods of the different classes separately. Fortunately, there is an easier way to do it; we can use pipelines and column transformers.

5.3.1 Pipeline

A pipeline allows us to chain different machine learning tasks by combining multiple estimators into a composite estimator. For instance, we can use a pipeline to chain a **`SimpleImputer`** estimator with a **`MinMaxScaler`** estimator. (Recall that any object with a **`fit()`** method is an estimator.)

All but the last estimator in the pipeline must have the **`transform()`** method (i.e., they must also be transformers), while the last estimator only needs to have the **`fit()`** method.

Let's look at an example.

The code below may appear a bit jumbled up if you are reading on a very small screen (e.g., on your mobile phone) or using a large font. If that's the case, try switching to landscape mode or using a smaller font size.

```
data = pd.DataFrame([[1], [4], [np.NaN], [8], [11]], columns=['A'])

from sklearn.pipeline import Pipeline

pl = Pipeline([
   ('imp', SimpleImputer(strategy="mean")),
   ('scaler', MinMaxScaler())
])
```

```
print(pl.fit_transform(data))
```

Here, we first create a DataFrame with one column (**`A`**) and five rows; the third row has a missing value.

Next, we import the **`Pipeline`** class from **`sklearn.pipeline`** and instantiate a **`Pipeline`** object.

The **`Pipeline()`** constructor accepts a list of (name, estimator) tuples as input. These tuples represent the tasks in the pipeline and are arranged according to the order of the tasks.

“name” is a user-chosen string for referencing the task, while “estimator” is the estimator needed to perform the task.

In the example above, the tuples are **`('imp', SimpleImputer(strategy="mean"))`** and **`('scaler', MinMaxScaler())`**.

`'imp'` and **`SimpleImputer(strategy="mean")`** are the name and estimator for the first task, while **`'scaler'`** and **`MinMaxScaler()`** are the name and estimator for the second.

After instantiating the **`Pipeline`** object, we assign it to a variable **`pl`**.

A **`Pipeline`** object has the same methods as its last estimator. For instance, if the last estimator is a predictor, the object has the **`fit()`** and **`predict()`** methods.

If we call a pipeline method that *includes fitting* (e.g., the **`fit()`**, **`fit_predict()`**, or **`fit_transform()`** method), sklearn sequentially calls the **`fit_transform()`** methods of all but the last estimator in the pipeline.

On the other hand, if we call a method that *does not include fitting* (e.g., **`predict()`** or **`transform()`**), it calls the **`transform()`** methods of those estimators.

In both cases, sklearn passes the output of each method call as input to the next. When it reaches the last estimator, it calls the corresponding

method of that estimator.

For instance, if we call the **`predict()`** method of a pipeline with four estimators, sklearn calls the **`transform()`** methods of the first three estimators before calling the **`predict()`** method of the last.

On the other hand, if we call the **`fit_predict()`** method of the same pipeline, it calls the **`fit_transform()`** methods of the first three estimators before calling the **`fit_predict()`** method of the last.

In our example, we call the **`fit_transform()`** method of the pipeline **`pl`**.

As the **`fit_transform()`** method includes fitting, sklearn calls the **`fit_transform()`** method of the first estimator in the pipeline and passes the result of this estimator to the **`fit_transform()`** method of the second (which is also the last).

As a result, we get the following array:

```
[[0. ]
[0.3]
[0.5]
[0.7]
[1. ]]
```

The first estimator (i.e., the **`SimpleImputer`** estimator) replaces the missing value in **`data`** with the mean before passing the resulting dataset to the last estimator (i.e., the **`MinMaxScaler`** estimator), which scales it to a range of 0 to 1.

Instead of having to call the **`fit_transform()`** method twice ourselves, the pipeline does it for us in the background. Therefore, we only need to call the **`fit_transform()`** method once to achieve the same result.

We'll have a chance to work with pipelines in subsequent chapters.

5.3.2 ColumnTransformer

Next, let's move on to column transformers.

A column transformer is similar to a pipeline, except that it is only for data transformation. Therefore, all estimators in a column transformer must have the **`transform()`** method. In addition, a column transformer does not pass the output of each method call as input to the next. Instead, it concatenates the results of the method calls and returns the transformed dataset.

Let's look at an example:

```
from sklearn.compose import ColumnTransformer

data = pd.DataFrame([[1], [4], [np.NaN], [8], [11]], columns=['A'])

ct = ColumnTransformer([
  ('imp', SimpleImputer(strategy="mean"), ['A']),
   ('scaler', MinMaxScaler(), ['A'])
])
```

Here, we first import the **`ColumnTransformer`** class from the **`sklearn.compose`** module. Next, we create the **`data`** DataFrame using the same values as the previous example.

After that, we instantiate a **`ColumnTransformer`** object and assign it to **`ct`**.

The **`ColumnTransformer()`** constructor accepts a list of transformers as input, with each transformer represented as a (name, transformer, columns) tuple. "name" is a user-chosen string for referencing the transformer, "transformer" is a transformer object, and "columns" is a list of columns.

In the example above, the first tuple is **`('imp', SimpleImputer(strategy="mean"), ['A'])`**.

This tuple is very similar to the first tuple we passed to the **`Pipeline()`** constructor previously, except that there is an additional element (**`['A']`**) in the current tuple. This additional element allows us to specify the column to which we want to apply the **`'imp'`** transformation.

Let's use the column transformer to transform **`data`** now:

```
print(ct.fit_transform(data))
```

The **`fit_transform()`** method in the **`ColumnTransformer`** class calls the **`fit_transform()`** method of all the transformers in **`ct`** *independently*. Each transformer gives a NumPy array, and all arrays are concatenated in the final result. The code above gives us the following output:

```
[[ 1.   0. ]
 [ 4.   0.3]
 [ 6.   Nan]
 [ 8.   0.7]
 [11.   1. ]]
```

As you can see, this result is different from the result returned by the **`pl`** pipeline in the previous section. While **`pl`** passed the result of one method call as input to the next and returned an array with one column, **`ct`** performs the two method calls independently and returns an array with two columns.

Pipelines and column transformers are both helpful in simplifying our workflow. Which one to use depends on the specifics of the task we want to perform.

One feature of a column transformer is that it allows us to specify the column(s) that we want a transformer to work on. Therefore, we can use it to apply different transformations to different columns. Let's use a dataset with three columns to demonstrate this.

```
data = pd.DataFrame({'A': [1, 2, 3, 4, 5], 'B':['Apple', 'Orange',
'Apple', 'Banana', 'Apple'], 'C':[11, 12, 13, 14, 15]})

ct2 = ColumnTransformer([
  ('encode', OrdinalEncoder(), ['B']),
  ('normalize', MinMaxScaler(), ['A'])
], remainder='passthrough')

print(ct2.fit_transform(data))
```

Here, we create a DataFrame with three columns **`A`**, **`B`**, and **`C`**.

Next, we pass a list of tuples to the **`ColumnTransformer()`** constructor to specify the transformation for column **`B`**, followed by that for column **`A`**.

Notice that we also pass **`remainder='passthrough'`** to the constructor.

By default, only columns specified in the list of tuples are transformed and returned in the resulting array; the remaining columns are dropped. By specifying **`remainder='passthrough'`**, the remaining columns are not dropped but are returned with the transformed columns.

If you run the code above, you'll get the following array:

```
[[ 0.    0.     11.  ]
 [ 2.    0.25   12.  ]
 [ 0.    0.5    13.  ]
 [ 1.    0.75   14.  ]
 [ 0.    1.     15.  ]]
```

The order of the columns in this array follows the order of the columns in the transformers list. Columns that are not transformed are added to the right of the transformed columns.

The first column above is the result of the "encode" transformation for column **`B`**, while the second is the result of the "normalize" transformation for column **`A`**. The last column is from column **`C`**, which is added to the result because we specify **`remainder='passthrough'`** when instantiating our **`ColumnTransformer`** object. We'll have a chance to use column transformers in our project later.

5.4 Model Evaluation with Scikit-Learn

Now, let's move on to model evaluation. An essential part of any machine learning project is evaluating how well our model performs.

For supervised machine learning, evaluation typically involves comparing the results predicted by our model with the actual labels in the dataset.

The following section discusses some of the frequently used metrics for evaluating classification and regression models. Classification models predict labels with discrete values (e.g., "Dog", "Cat", and "Bird") while regression models predict labels with continuous values (e.g., 1.32, 52.1, and 21.1).

5.4.1 Classification metrics

Let's start with classification. The commonly used metrics for classification are accuracy, precision, and recall.

Accuracy

Accuracy refers to the fraction of instances predicted correctly by our model. It ranges from 0 to 1, with 1 indicating 100% accuracy. Suppose we want to classify 50 dogs and 50 cats. If we classify 90 of the animals correctly, the accuracy is 0.9.

To calculate accuracy, we use the **`accuracy_score()`** function in the **`sklearn.metrics`** module. This function requires us to pass the true labels first, followed by the predicted labels. Let's look at an example.

```
from sklearn.metrics import accuracy_score

true = ['Cat', 'Cat', 'Dog', 'Dog', 'Cat', 'Dog']
pred = ['Cat', 'Cat', 'Cat', 'Dog', 'Cat', 'Cat']

score = accuracy_score(true, pred)
print(score)
```

Here, we first import the **`accuracy_score()`** function from the **`sklearn.metrics`** module. Next, we declare two lists - **`true`** and **`pred`**

- for storing the true and predicted labels, respectively.

We then pass the two lists to the **`accuracy_score()`** function and assign the result to **`score`**. If you run the code above, you'll get

```
0.6666666666666666
```

as the output. We get this result because four out of the six labels in the **`true`** list are predicted correctly.

Precision and Recall

Classification accuracy is very easy to calculate and interpret. However, one issue with this score is that it can be misleading if we have imbalanced data.

Suppose our dataset consists of 95 dogs and 5 cats. A model that blindly classifies all animals as dogs has an accuracy of 0.95. However, this model did not achieve such accuracy by detecting any meaningful pattern in the dataset. As such, it is unlikely to perform well on a new dataset, especially one that contains more cats.

Besides looking at classification accuracy, we can look at precision and recall. To understand precision and recall, we need to discuss the confusion matrix. Referring to the two lists in the previous example:

```
true = ['Cat', 'Cat', 'Dog', 'Dog', 'Cat', 'Dog']
pred = ['Cat', 'Cat', 'Cat', 'Dog', 'Cat', 'Cat']
```

the confusion matrix is given below:

		Predicted Values: Dog	Predicted Values: Cat
True Values	Dog	1 (TP)	2 (FN)
	Cat	0 (FP)	3 (TN)

There are four terms to understand when interpreting the confusion matrix: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). We define these terms based on the *predicted* values.

In our example above, suppose we consider `'Dog'` as positive and `'Cat'` as negative.

If an instance is predicted as “Positive” (refer to the column highlighted in grey above), it can either be a true positive (TP) or a false positive (FP) outcome. TP is an outcome where an instance is correctly predicted as positive, while FP is an outcome where it is incorrectly predicted as positive.

The number of TP and FP outcomes in our example are 1 and 0, respectively. In other words, out of the 1 instance predicted as “Positive” (i.e., as `'Dog'`), 1 of them is predicted correctly, and 0 of them predicted wrongly.

Next, we have FN and TN outcomes.

If an instance is predicted as “Negative”, it can either be a false negative (FN) or a true negative (TN) outcome. FN is an outcome where it is incorrectly predicted as negative, while TN is an outcome where it is correctly predicted as negative.

In our example above, the number of FN and TN outcomes are 2 and 3, respectively. In other words, out of the 5 instances predicted as “Negative” (i.e., as `'Cat'`), 2 of them are predicted wrongly (i.e., they are actually dogs), and 3 of them are predicted correctly.

False positives and false negatives are both undesirable outcomes, which value to minimize depends on our model’s objective.

If we are developing a model to predict the presence of a malignant tumor (with “Negative” indicating absence of tumor), we would want to minimize FN outcomes as we do not want to deprive a patient of treatment because of a misdiagnosis. On the other hand, if we are trying to classify investment opportunities, we would like to minimize FP

outcomes as we do not want to invest in something that ends up being unprofitable.

TP, FP, and FN are used in the calculations of precision and recall.

Precision measures the percentage accuracy of a classifier *when it predicts that an instance is positive*. It is given by the formula

Precision = TP / (TP + FP)

In our example, the classifier only predicts one instance as positive (i.e., predicts that it is a dog) and that instance is indeed positive. Therefore, precision equals 1. Using the precision formula above,

Precision = TP / (TP + FP) = 1 / (1 + 0) = 1

Next, let's look at recall. Recall measures the percentage accuracy of our classifier *when the true value of an instance is positive*. It is given by the formula

Recall = TP / (TP + FN)

In our example, we have three positive instances in the **`true`** array. Out of these three instances, the classifier predicts one instance correctly. Therefore, recall is 1/3. Using the recall formula above,

Recall = TP / (TP + FN) = 1 / (1 + 2) = 0.333333

To calculate recall and precision in sklearn, we use the **`precision_score()`** and **`recall_score()`** functions in the **`sklearn.metrics`** module:

```
from sklearn.metrics import precision_score, recall_score

true = ['Cat', 'Cat', 'Dog', 'Dog', 'Cat', 'Dog']
pred = ['Cat', 'Cat', 'Cat', 'Dog', 'Cat', 'Cat']

precision = precision_score(true, pred, pos_label = 'Dog')
recall = recall_score(true, pred, pos_label = 'Dog')

print(precision)
print(recall)
```

Both functions require us to pass the true labels, followed by the

predicted labels. We also pass **`pos_label = 'Dog'`** to indicate that we want to consider **`'Dog'`** as positive.

If you run the code above, you'll get the following values:

```
1.0
0.3333333333333333
```

which match the results of our calculations above.

5.4.2 Regression metrics

Now, let's discuss metrics for evaluating regression models. The commonly used metrics are Root Mean Squared Error and R-squared.

Root Mean Squared Error

Root Mean Squared Error (RMSE) is the square root of Mean Squared Error (MSE).

MSE is determined by calculating the mean of the squared differences between the true and predicted values. Suppose we have two lists - **`true`** and **`pred`** - that store the true and predicted values of 4 instances:

```
pred = [2.1, 1.4, 5.6, 7.9]
true = [2.5, 1.6, 5.1, 6.8]
```

To calculate MSE, we do the following:

Calculate the differences between the true and predicted values

differences = [-0.4, -0.2, 0.5, 1.1]

Square the differences

squares = [0.16, 0.04, 0.25, 1.21]

Calculate the mean of the squares

mean = (0.16 + 0.04 + 0.25 + 1.21)/4 = 0.415

This gives us the MSE. To get RMSE, we take the square root of MSE.

RMSE = Square root of 0.415 = 0.644

There is no simple answer for what a good RMSE is. A RMSE of $200 when predicting housing price is excellent (probably too good to be true) as it suggests a typical prediction error of only $200. The same RMSE when predicting the cost of a cab ride, on the other hand, is abysmal. All things being equal, the lower the RMSE, the better.

One advantage of using RMSE as an evaluation metric is that larger errors are amplified compared to smaller errors (due to the squaring of the errors). This forces our model to focus on reducing large errors. In addition, RMSE uses the same unit as the target variable. For instance, if the target variable is measured in feet (e.g., height), the unit for the RMSE will also be feet. This makes it easier for us to interpret the RMSE.

R-squared

Besides RMSE, another popular regression metric is R-squared, also known as the R^2 score. This score measures how much of the variance in the target variable can be explained by our model. Variance is a measure of how far the true values are from their mean.

The formula for calculating the R^2 score is:

$$R^2\ score = 1 - \frac{SSD\ between\ predicted\ and\ true\ values}{SSD\ between\ true\ values\ and\ mean}$$

$$where\ SSD = Sum\ of\ Squared\ Differences$$

Let's use an example to illustrate:

```
pred = [2.1, 1.4, 5.6, 7.9]
true = [2.5, 1.6, 5.1, 6.8]
```

Here, the true values are 2.5, 1.6, 5.1, and 6.8, with a mean of 4. To calculate the R^2 score, we do the following:

Find the sum of the squared differences between the true values and their mean (which equals 4)

differences between true values and mean = [-1.5, -2.4, 1.1, 2.8]
squared differences = [2.25, 5.76, 1.21, 7.84]
sum of square differences = 2.25 + 5.76 + 1.21 + 7.84 = 17.06

Find the sum of the squared differences between the true and predicted values

differences between true values and predicted values = [-0.4, -0.2, 0.5, 1.1]
squared differences = [0.16, 0.04, 0.25, 1.21]
sum of squared differences = 0.16 + 0.04 + 0.25 + 1.21 = 1.66

Calculate the R^2 score

$R^2 = 1 - (1.66/17.06) = 0.902696$

If you find the calculations above daunting, do not worry. What is important is to understand that, in general, the higher the R^2 score, the better. The best possible R^2 score is 1, which happens when the sum of squared differences between the true and predicted values is zero. It is also possible for R^2 to be negative. This occurs when the model performs worse than the mean of the data.

The calculations above can be done using the **`sklearn.metrics`** module. This module comes with two functions - **`mean_squared_error()`** and **`r2_score()`** - for calculating the RMSE and R^2 score, respectively. Both functions require us to pass the true values first, followed by the predicted values:

```
from sklearn.metrics import r2_score, mean_squared_error
pred = [2.1, 1.4, 5.6, 7.9]
true = [2.5, 1.6, 5.1, 6.8]
RMSE = mean_squared_error(true, pred, squared=False)
r2 = r2_score(true, pred)
print(RMSE)
print(r2)
```

Here, we first use the **`mean_squared_error()`** function to calculate the RMSE. By default, this function gives us the MSE. If you want the RMSE, you need to pass **`squared=False`** to the function.

Next, we use the **`r2_score()`** function to calculate the R^2 score. If you run the code above, you'll get the following values:

```
0.6442049363362565
0.902696365767878
```

which are the same as the ones we calculated.

5.5 Model Selection with Scikit-Learn

The section above discussed various metrics for model evaluation. When working on a machine learning project, we commonly build more than one machine learning model and use the metrics above to select the best-performing model.

There are several approaches to model selection.

5.5.1 Train Test Split

One approach is to split our dataset into training and test subsets and train different models using the training set. We first evaluate the models on the training set and select the best-performing model. This model is then further evaluated on the test set to determine if it generalizes well to data not used in its training.

To split our dataset into training and test subsets, we can use the **`train_test_split()`** function in the **`sklearn.model_selection`** module.

This function accepts one or more arrays (such as lists, NumPy arrays, or pandas DataFrames) as input and splits the array(s) into training and test subsets. After splitting, it returns two or more arrays containing the train-test split of the input array(s).

Let's use a simple dataset to illustrate how the function works. Suppose we pass the following arrays to the function:

```
X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [23, 11, 31, 45, 12, 65, 43, 69, 13, 12]
```

The function randomly splits the two arrays and returns four arrays. If we do a 80-20 split, the function may return the following arrays:

Training Subset

```
X = [1, 2, 4, 5, 6, 7, 9, 10]
y = [23, 11, 45, 12, 65, 43, 13, 12]
```

Test Subset

```
X = [3, 8]
y = [31, 69]
```

We typically use the training subset to train different models and evaluate them. The best model is then selected and evaluated on the test subset. We'll demonstrate how to use the **`train_test_split()`** function in subsequent chapters when we work with actual datasets.

5.5.2 k-Fold Cross-Validation

Besides using train-test split for model selection, we can do k-fold cross-validation.

k-fold cross-validation involves splitting our dataset into k-subsets, known as k folds. Out of the k folds, k-1 of them are used for training, while the remaining fold is used for validation. The algorithm is trained and tested k times, each time using a new fold as the validation set. Finally, the result of the cross-validation process is the average of the results obtained on each run.

Let's use a simple dataset to illustrate. Suppose we have the following dataset:

```
X = [1, 2, 3, 4, 5, 6, 7, 8, 9]
y = [23, 11, 31, 45, 12, 65, 43, 69, 13]
```

and decide to do a 3-fold cross-validation, we can split **`X`** and **`y`** into the following folds

Fold 1

```
X = [1, 5, 9], y = [23, 12, 13]
```

Fold 2

```
X = [2, 6, 8], y = [11, 65, 69]
```

Fold 3

```
X = [3, 4, 7], y = [31, 45, 43]
```

To do cross-validation, we train a model on folds 1 and 2 and evaluate it on fold 3. Next, we repeat the process twice by training the model on folds 2 and 3, evaluating it on fold 1, and training it on folds 1 and 3, and evaluating it on fold 2.

Each evaluation gives us a score, say 0.97, 0.93 and 0.99. We take the mean of these scores (0.97 + 0.93 + 0.99)/3 = 0.963 as the final score.

If we have a reasonably large dataset, we can combine k-fold cross-validation with a train-test split. To do that, we first split our dataset into training and test subsets. Next, we *split the training subset further* into k-folds and perform cross-validation on these folds to select the best model. The selected model is then evaluated on the test set to determine if it generalizes well to unseen data.

When we perform k-fold cross-validation on the training set, the model is trained k times, giving us a more robust estimate of the model's performance.

There are many ways to do cross-validation in sklearn. The easiest is to use the **`cross_val_score()`** function in the **`sklearn.model_selection`** module. We'll show how to use this function when we discuss machine learning algorithms in subsequent chapters.

Chapter 6 - Regression

Now that we are familiar with the main libraries for machine learning, we are ready to move on to algorithms. This chapter starts with one of the most fundamental branches of machine learning - regression.

6.1 What is Regression?

Regression is a form of analysis that attempts to determine the relationship between a dependent variable (known as the target variable) and a series of other variables (known as independent variables, predictor variables, or features). After determining the relationship, we can use it to make predictions for the target variable when given new feature values.

The target variable in regression can take on continuous values (such as 1.234, 31.23, and 9.131). This is in contrast to classification, where the target variable can only take on discrete values (such as 1, 2, 3, or ‘Male’ and ‘Female’).

There are many machine learning algorithms for regression. Commonly used ones include linear regression, polynomial regression, and support vector regression. This chapter discusses two of the most basic but popular regression algorithms - linear and polynomial regression.

We’ll first discuss the theory behind them before showing the steps involved in building a regression model in Python.

6.2 Linear Regression

Linear regression attempts to find a *linear relationship* between the target and predictor variables. We can express this relationship as

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n$$

where y is the target variable, and $x_1, x_2, \ldots, x_n$ are the predictor variables or features.

Linear regression that only involves one feature is known as simple linear regression, while that with multiple features is known as multiple linear regression.

$a_0, a_1, a_2, \ldots, a_n$ are the parameters (also known as coefficients or weights) of the model. The main objective of the linear regression algorithm is to determine the best values for these parameters.

To understand how the algorithm works, let's consider an example.

For ease of understanding, we'll use simple linear regression to demonstrate. Graphically, a simple linear relationship between two variables is represented as a straight line and can be easily visualized.

The idea behind simple linear regression is straightforward. The algorithm tries to draw a line that best fits the given data. This line is known as the regression line and *represents the predicted values* generated by the algorithm.

As an illustration, let's consider three data points - (0.5, 7), (1.5, 4), and (2.5, 10), represented by three dots in the chart below. We want to draw a line that best fits these three points. Here, we show two out of an infinite number of possible lines:

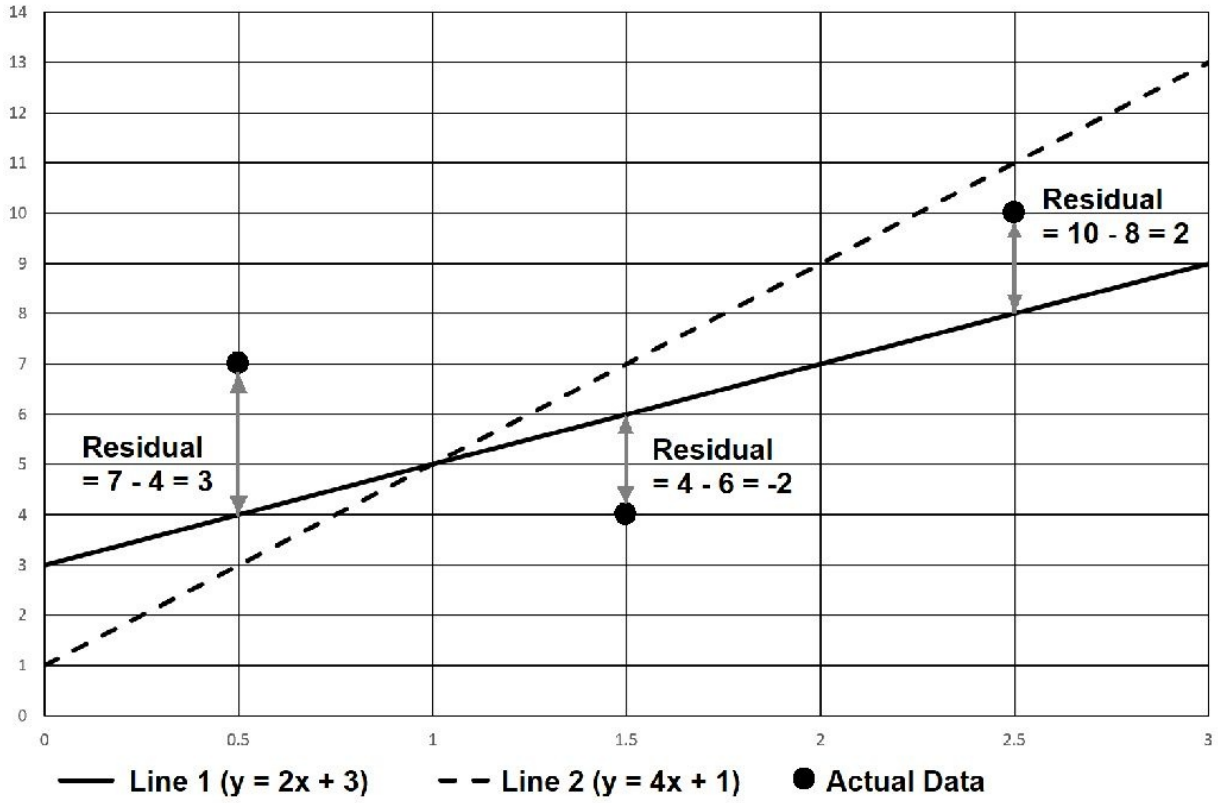

Figure 6.1: Comparing two regression lines

How do we choose the best line?

To do that, we need to minimize what is known as the cost function. A commonly used cost function for linear regression is the Mean Squared Error (MSE) function discussed in the previous chapter.

A data point's error refers to the distance between its predicted and actual value. This error is technically known as the *residual*. The objective of the linear regression algorithm is to find the set of parameters that results in the smallest MSE.

In the figure above, the parameters of the solid line (given by the equation y = 2x + 3) are 2 and 3.

2 represents the slope of the line, while 3 represents the vertical intercept. With this set of parameters, the MSE is given by:

$MSE = [(7- 4)^2 + (4 - 6)^2 + (10 - 8)^2] / 3 = 5.67$

For the dotted line (y = 4x + 1), the parameters are 4 and 1 and the MSE

is given by:

$MSE = [(7 - 3)^2 + (4 - 7)^2 + (10 - 11)^2] / 3 = 8.67$

Based on a comparison of the two MSEs, the solid line is a better fit as it has a smaller MSE.

Theoretically, the linear regression algorithm needs to consider all possible values of the parameters and select the set of parameters that results in a line with the smallest MSE. However, it is impossible to do so in practice as the number of possible values is infinite.

Hence, the algorithm uses mathematical techniques to find the set of optimal parameters instead. These techniques include solving a matrix equation known as the normal equation or using a technique called gradient descent to try different sets of parameters iteratively.

Thankfully, we do not need to perform these computations ourselves; we can use the **`LinearRegression`** class in sklearn to do it.

6.3 Linear Regression with Scikit-Learn

In this section, we'll learn to build a linear regression model using the **`LinearRegression`** class, based on data stored in a file called *housing.csv*.

Let's get started.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
%matplotlib inline
```

Here, we first import the NumPy, pandas, and Matplotlib libraries.

Next, we import the **`LinearRegression`** class from the **`sklearn.linear_model`** module; this class is needed to train our model. We also import the **`train_test_split()`** function for splitting our dataset, and the **`mean_squared_error()`** and **`r2_score()`** functions for evaluating the model.

Step 1: Loading the Data

After importing the modules, let's load the data in *housing.csv* into a DataFrame:

```
housing = pd.read_csv('housing.csv')
```

Step 2: Examine the data

Now, let's do some basic data exploration.

```
housing.head()
```

This gives us the following output:

	Floor Area (sqft)	Value ($1000)
0	665.0	161.0
1	442.0	83.0
2	302.0	53.0
3	336.0	57.0
4	673.0	152.0

Figure 6.2: The housing dataset

This dataset consists of two columns - **`Floor Area (sqft)`** and **`Value ($1000)`**. The first column gives the floor area of a house in sqft, while the second gives the price of the house in multiples of $1000. We want to use floor area to predict the price of a house.

First, let's check if there are any missing values in this dataset:

```
housing.isnull().sum()
```

This gives us the following output, which indicates that there are no missing values:

```
Floor Area (sqft)    0
Value ($1000)        0
dtype: int64
```

Step 3: Split the Dataset

The third step is to split our data into training and test subsets. To do that, we use the **`train_test_split()`** function.

```
X = housing[['Floor Area (sqft)']]
y = housing['Value ($1000)']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Here, we first select the feature and target variable in our dataset and assign them to **`X`** and **`y`**, respectively. Most sklearn methods expect features to be passed as a 2D array. Therefore, we select the feature as

a DataFrame using two sets of square brackets.

Next, we pass **`X`** and **`y`** to the **`train_test_split()`** function. To specify the amount of data to use for testing, we use the **`test_size`** parameter. For this example, we specify **`test_size`** as 0.2, which means we want to use 20% of our data for the test set.

Finally, we specify the **`random_state`** parameter. This parameter controls the shuffling applied to the data before the split. The number 42 has no special meaning; you can choose another integer if you desire. However, if you want to produce the same output shown in this chapter, you need to use the same number 42. Else, your data will be shuffled differently, resulting in a different split and different outputs.

The **`train_test_split()`** function returns four arrays in the following order:

- the training subset of the features
- the test subset of the features
- the training subset of the target variable, and
- the test subset of the target variable

We assign these four arrays to **`X_train`**, **`X_test`**, **`y_train`**, and **`y_test`**, respectively.

As **`X`** is a DataFrame, **`X_train`** and **`X_test`** are DataFrames. Similarly, as **`y`** is a Series, **`y_train`** and **`y_test`** are Series.

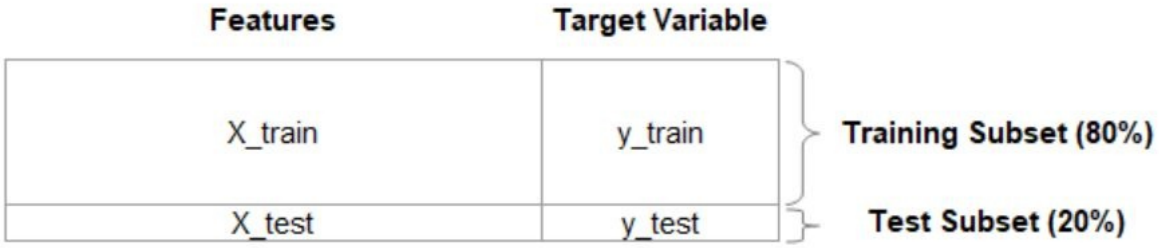

Figure 6.3: Training and test subsets

Step 4: Visualizing the Data

Now, let's do a simple scatter plot of the training subset to explore the

relationship between housing price and floor area:

```
plt.scatter(X_train['Floor Area (sqft)'], y_train, s=10)
plt.title('Housing Price vs Floor Area')
plt.xlabel('Floor Area (sqft)')
plt.ylabel('Price in $1000s')
```

This gives us the following chart.

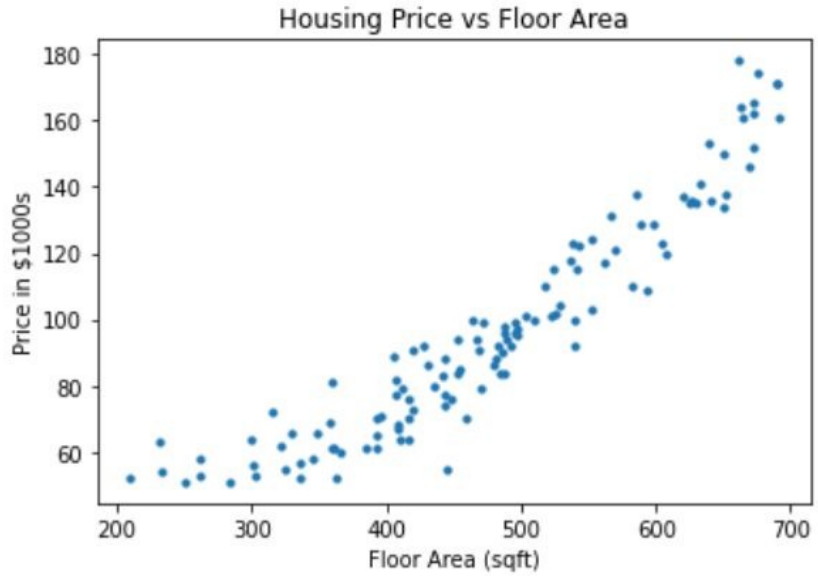

Figure 6.4: Scatter plot of the training set

Step 5: Data Preprocessing

There is no need to perform data preprocessing for this example as our dataset does not contain any missing value or textual data. In addition, we do not need to do feature scaling as there is only one feature in the dataset.

Step 6: Train the Model

We are now ready to train our model.

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

Here, we initialize a **`LinearRegression`** object **`lr`** and use it to call the **`fit()`** method in the **`LinearRegression`** class, passing **`X_train`** and **`y_train`** to the method.

The **`fit()`** method calculates the parameters of our model by minimizing the cost function discussed previously. It does not return any value. Instead, it updates the **`coef_`** and **`intercept_`** attributes in the **`LinearRegression`** class.

We can print these attributes using the statements below:

```
print(lr.intercept_)
print(lr.coef_)
```

This gives us the following output:

```
-29.6514648875731
[0.2640654]
```

You may get slightly different results due to differences in how our systems handle floating-point numbers.

A simple linear relationship can be represented by the equation

$y = a_0 + a_1 x_1$

`intercept_` and **`coef_`** give the values of a_0 and a_1, respectively.

Based on the output above, the model determines that we can express the relationship between house price and floor area as

house price = -29.7 + 0.264*(floor area)

* Coefficients are rounded off to 3 significant figures in the equation above.

Using this relationship, we can predict the prices of houses based on their floor areas. For instance, if we want to predict the prices of two houses with floor areas of 250sqft and 300sqft each, we use the calculations below:

house price = -29.7 + 0.264*250 = 36.3

house price = -29.7 + 0.264*300 = 49.5

Alternatively, we can use the **`predict()`** method in the

`LinearRegression` class:

```
predicted_price = lr.predict([[250], [300]])
print(predicted_price)
```

To use the **`predict()`** method, we need to pass a 2D array to the method. Here, we pass **`[[250], [300]]`** to the method.

`[250]` represents the feature (i.e., the floor area) of the first house, while **`[300]`** represents the second.

The code above gives the following output:

```
[36.3648842  49.56815401]
```

These values are very close to the ones we calculated ourselves, with slight differences due to rounding errors. We can plot the regression line using the code below:

```
plt.scatter(X_train['Floor Area (sqft)'], y_train, s=10)

y_pred = lr.predict(X_train)
plt.plot(X_train['Floor Area (sqft)'], y_pred)

plt.title('Housing Price vs Floor Area')
plt.xlabel('Floor Area (sqft)')
plt.ylabel('Price in $1000s')
```

Here, we first plot a scatter plot for the training subset using the **`scatter()`** function.

Next, we get the predicted values for **`X_train`** using the **`predict()`** method and store the results in a variable called **`y_pred`**. We then pass **`X_train['Floor Area (sqft)']`** and **`y_pred`** to the **`plot()`** function to plot the regression line. If you run the code above, you'll get the following output:

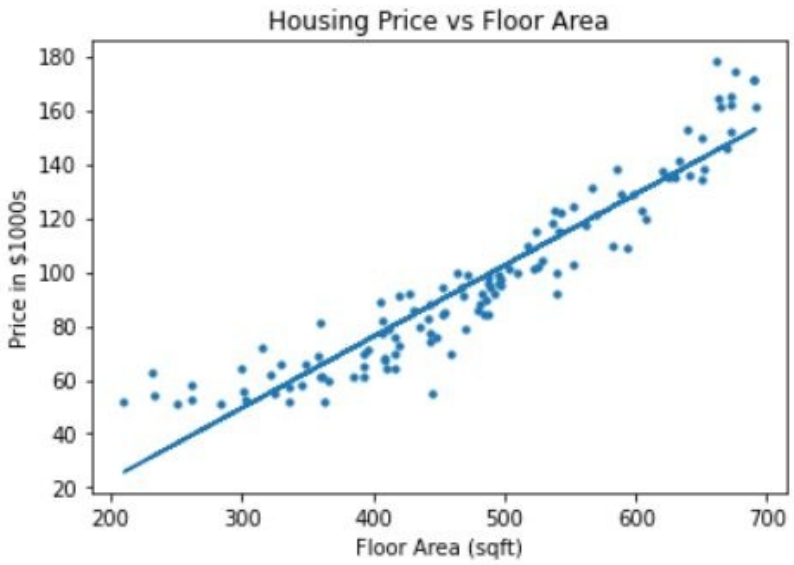

Figure 6.5: Plotting the regression line

Step 7: Evaluate the Model

After we get the parameters of our model, we need to evaluate the model's performance:

```
RMSE = mean_squared_error(y_train, y_pred, squared=False)
r2 = r2_score(y_train, y_pred)
print(RMSE)
print(r2)
```

Here, we evaluate the model on the training subset by passing the true y values (**`y_train`**) and the predicted y values (**`y_pred`**, which we calculated in Step 6) to the **`mean_squared_error()`** and **`r2_score()`** functions. This gives us the following output:

```
11.426788012892116
0.8827389714759885
```

These scores are reasonably good, but not excellent. We can further evaluate the model on the test set to see how well it generalizes to the test set.

To do that, we pass **`X_test`** to the **`predict()`** method.

Note that we should *NOT* pass **`X_test`** to the **`fit()`** method. If we do that, **`lr`** will calculate the model parameters again using the test set,

which is not what we want. The code below shows how to evaluate the model on the test set:

```
y_pred_test = lr.predict(X_test)
RMSE = mean_squared_error(y_test, y_pred_test, squared=False)
r2 = r2_score(y_test, y_pred_test)
print(RMSE)
print(r2)
```

This gives us the following output:

```
10.413298895214577
0.9229207556091985
```

which shows that our model generalizes well to unseen data. In fact, the model performs slightly better on the test set than the training set.

6.4 Polynomial Regression

Although the linear regression model scores above are decent, you may wonder if it is possible to improve them.

If you refer to the chart in Step 6 above, you'll see that the regression line fails to capture the relationship in our dataset fully. This is especially evident for data points in the bottom left corner of the chart. These points seem to lie more on a quadratic curve than a straight line.

If we want to test the hypothesis that a quadratic curve fits our data better, we can use polynomial regression.

Polynomial regression models the relationship between the features and target variable as an n^{th} degree polynomial. In the case of a quadratic relationship between a single feature x and a target variable y, the model takes the form:

$$y = a_0 + a_1 x + a_2 x^2$$

The objective of the polynomial regression algorithm is to find the set of coefficients (i.e., a_0, a_1, and a_2) that results in the smallest MSE. Let's learn to do that in sklearn.

6.5 Polynomial Regression with Scikit-Learn

Building a polynomial regression model with sklearn is similar to building a regular linear regression model. In fact, both use the **`LinearRegression`** class. This is because we can transform a polynomial equation into a linear equation.

With reference to the quadratic equation above, if we replace x^2 with w (i.e., let $w = x^2$), we get the equation below:

$$y = a_0 + a_1x + a_2w$$

This is a linear equation between one target variable y and *two* predictor variables x and w.

In other words, a quadratic regression model corresponds to a *multiple linear regression* model with two features - x and w (where $w = x^2$).

To build a model for the relationship above, we need to create the additional feature w (i.e., x^2). This is not too difficult to do.

However, if we have more features or we want to use higher degrees, the examples below show the *additional* features we need:

- One feature x with degree 3, we need x^2 and x^3.
- Two features v and x with degree 2, we need v^2, vx, and x^2.
- Two features v and x with degree 3, we need v^2, x^2, v^3, x^3, vx, v^2x, and vx^2.

The more features we have, the harder it is to figure out the features we need. Fortunately, sklearn provides a class called **`PolynomialFeatures`** that can create these features for us. Let's learn to do that now.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2)
```

```
X_train_transformed = poly_features.fit_transform(X_train)
```

Here, we first import the **`PolynomialFeatures`** class and instantiate an object called **`poly_features`**. We pass **`degree=2`** to the constructor to indicate that we want to generate features up to degree 2.

Next, we pass **`X_train`** to the **`poly_features.fit_transform()`** method to get the features we need. As **`X_train`** consists of one feature and we specify **`degree=2`**, the **`fit_transform()`** method gives us three output features.

If the original feature is x, the method gives us x^0, x^1, and x^2.

x^0 is the constant 1 as any number raised to the power of 0 is 1.

x^1 is the original feature, and x^2 is the feature raised to the power of 2.

Here, we assign the output features to a variable called **`X_train_transformed`**. If you print the first element in **`X_train_transformed`** now:

```
print(X_train_transformed[0])
```

you'll get the following output:

```
[1.00000e+00 6.91000e+02 4.77481e+05]
```

These numbers are presented in scientific notation.

1.00000e+00 = 1 is the value for (floor area)^0,
6.91000e+02 = 691 is the value for (floor area)^1, and
4.77481e+05 = 477481 is the value for (floor area)^2.

To train our model, we instantiate a **`LinearRegression`** object **`pr`** and pass **`X_train_transformed`** (which stores the three features discussed above) and **`y_train`** to its **`fit()`** method:

```
pr = LinearRegression()
pr.fit(X_train_transformed, y_train)
print(pr.intercept_)
print(pr.coef_)
```

If you run the code above, you'll get the following output:

```
70.91161404547069
[ 0.         -0.18658297  0.00047343]
```

Based on these parameters, the model estimates that the relationship between house price and floor area can be expressed as:

house price = 70.9 + 0*(floor area)^0 - 0.187*(floor area)^1 + 0.000473*(floor area)^2

As before, the coefficients above are rounded off to 3 significant figures.

We can predict the price of a house with a floor area of 250 sqft using the calculation below:

house price = 70.9 + 0*(250)^0 - 0.187*(250)^1 + 0.000473*(250)^2 = 53.7125

Alternatively, we can use the model's **`predict()`** method. To do that, we need to pass 250 (as a 2D array) to **`poly_features.transform()`** to generate higher powers of 250 first, before passing the returned values to **`pr.predict()`**:

```
num_transformed = poly_features.transform([[250]])
predicted_price = pr.predict(num_transformed)
print(predicted_price)
```

If you run the code above, you'll get **`[53.85545111]`** as the output. The discrepancy between this value and our calculated value of 53.7125 is due to rounding errors.

To visualize how well our model fits the data, let's plot the regression curve.

```
# Plotting the scatter plot
plt.scatter(X_train['Floor Area (sqft)'], y_train, s=10)

# Plotting the curve
y_pred = pr.predict(X_train_transformed)
sorted_zip = sorted(zip(X_train['Floor Area (sqft)'], y_pred))
X_train_sorted, y_pred_sorted = zip(*sorted_zip)
plt.plot(X_train_sorted, y_pred_sorted)
```

```
# Labelling the Chart
plt.title('Housing Price vs Floor Area')
plt.xlabel('Floor Area (sqft)')
plt.ylabel('Price in $1000s')
```

Here, we first pass **`X_train['Floor Area (sqft)']`** and **`y_train`** to the **`scatter()`** function to plot a scatter plot for the training set.

Next, we pass **`X_train_transformed`** to the **`predict()`** method to get the predicted values and assign them to **`y_pred`**.

Now, we are almost ready to plot the regression curve using **`X_train['Floor Area (sqft)']`** and **`y_pred`**. However, recall that the **`plot()`** function in Matplotlib requires points to be sorted based on their x values. This poses a problem as **`X_train['Floor Area (sqft)']`** is not sorted.

To get the correct curve, we need to zip **`X_train['Floor Area (sqft)']`** and **`y_pred`** into a zip object, sort this object, and unzip the resulting list into two tuples (refer to Chapter 4, Section 4.2.4).

After sorting, we assign the resulting tuples to **`X_train_sorted`** and **`y_pred_sorted`** and pass them to the **`plot()`** function. If you run the code above, you'll get the following output:

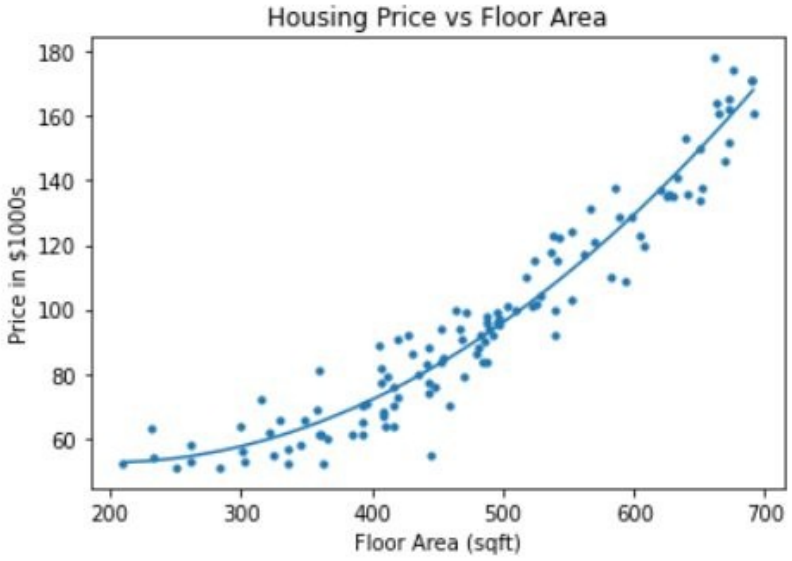

Figure 6.6: Plotting the regression curve

This curve definitely looks like a better fit than the previous straight line. Let's evaluate our polynomial regression model now:

```
RMSE = mean_squared_error(y_train, y_pred, squared=False)
r2 = r2_score(y_train, y_pred)
print(RMSE)
print(r2)
```

Here, we evaluate the model on the training subset. This gives us the following output:

```
8.522835594847725
0.9347660644643525
```

Next, let's evaluate it on the test subset:

```
X_test_transformed = poly_features.transform(X_test)
y_pred_test = pr.predict(X_test_transformed)
RMSE = mean_squared_error(y_test, y_pred_test, squared=False)
r2 = r2_score(y_test, y_pred_test)
print(RMSE)
print(r2)
```

This gives us the following output:

```
6.425083465019807
0.9706560666441122
```

As expected, the polynomial regression model performs better than the linear regression model. For instance, the RMSE for the polynomial regression model evaluated on the test subset is 6.43 (rounded off), lower than the corresponding value for the linear regression model (10.4).

Similarly, the R^2 score is also better (0.971 vs. 0.923).

6.6 Pipeline

In the previous section, we learned to train a polynomial regression model.

To do that, we need to do two things. First, we need to create the additional features using the **`PolynomialFeatures`** class. After creating the features, we pass them to the linear regression algorithm to train our model.

The following section shows how we can combine the two steps using a pipeline.

To do that, we use the code below:

```
from sklearn.pipeline import Pipeline

# Creating the pipeline
pipeline = Pipeline([('poly', PolynomialFeatures(degree=2)), ('model',
LinearRegression())])

# Training the Model
pipeline.fit(X_train, y_train)
```

Here, we first import the **`Pipeline`** class from the **`sklearn.pipeline`** module.

Next, we instantiate a **`Pipeline`** object and pass two tuples to its constructor. The first tuple is for the **`PolynomialFeatures`** estimator, while the second is for the **`LinearRegression`** estimator.

After instantiating the **`Pipeline`** object (**`pipeline`**), we train our model by calling the object's **`fit()`** method.

When we do that, sklearn calls the **`fit_transform()`** method of the **`PolynomialFeatures`** estimator and passes the output to the **`fit()`** method of the **`LinearRegression`** estimator.

Once that is done, the training is complete and our model is ready. We can use the model to make predictions using the **`predict()`** method,

similar to what we did previously. In addition, we can evaluate it on the training and test subsets:

```
# Using the Model to make predictions
print(pipeline.predict([[250]]))

# Evaluating the model on the train set
y_pred = pipeline.predict(X_train)
print(mean_squared_error(y_train, y_pred, squared=False))
print(r2_score(y_train, y_pred))

# Evaluating the model on the test set
y_pred_test = pipeline.predict(X_test)
print(mean_squared_error(y_test, y_pred_test, squared=False))
print(r2_score(y_test, y_pred_test))
```

If you run the code above, you'll get the following output:

```
[53.85545111]
8.522835594847725
0.9347660644643525
6.425083465019807
0.9706560666441122
```

These values are identical to what we got previously without using a pipeline. However, notice how much more convenient it is to use a pipeline to train and evaluate our model.

In particular, we do not need to store the output features returned by the **`PolynomialFeatures`** class. This helps to streamline our workflow substantially.

6.7 Cross-Validation

Besides simplifying our workflow, another advantage of using a pipeline is it helps to prevent data leakage when we do cross-validation. Data leakage occurs when information from outside the training set (such as the test set) is used to train the model.

When we discussed data preprocessing previously, we mentioned that we should only call the **`transform()`** method on the test set and not the **`fit()`** or **`fit_transform()`** method. This ensures that data from the test set is not used to train the model, which is the correct approach.

The same applies when we do cross-validation.

If we need to do any data preprocessing on our dataset, we should call the **`fit()`** or **`fit_transform()`** method on the k-1 training folds and only call the **`transform()`** or **`predict()`** method on the validation fold.

Doing so manually is tedious. Fortunately, we can use a pipeline.

If we use a pipeline for cross-validation, sklearn applies the **`fit_transform()`** and **`fit()`** methods on the training folds. For the validation fold, it only applies the **`transform()`** and **`predict()`** methods.

The code below shows how we can use a pipeline for cross-validation:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(pipeline, X_train, y_train, cv=3,
scoring="neg_root_mean_squared_error")
neg_rmse = scores.mean()
-neg_rmse
```

Here, we first import the **`cross_val_score()`** function. Next, we call the function by passing five arguments to it.

The first argument is the pipeline (**`pipeline`**), followed by the array for the features (**`X_train`**), the array for the target variable (**`y_train`**), the number of folds (**`cv=3`**), and the scoring criterion.

The scoring criterion is passed as a string, using predefined values listed at https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter.

In the example above, we use the **`"neg_root_mean_squared_error"`** criterion, which corresponds to the negative value of RMSE.

`cross_val_score()` expects the scoring criterion to be based on a metric where greater values are better. Therefore, instead of using RMSE as the criterion (where smaller values are better), we use negative RMSE (where greater values are better).

The **`cross_val_score()`** function gives us three scores as we specify the number of folds to be three. We assign these scores to a variable called **`scores`** and use the **`mean()`** method to find their mean. We then assign the mean to **`neg_rmse`** and print the value of **`-neg_rmse`**.

If you run the code above, you'll get the following output:

```
8.615541039554307
```

We can also use the **`cross_val_score()`** function to get the R^2 score:

```
scores = cross_val_score(pipeline, X_train, y_train, cv=3, scoring="r2")
r2 = scores.mean()
r2
```

Here, we use **`"r2"`** as the scoring criterion. This gives us the following output:

```
0.9291077625814611
```

Both scores indicate that our model is reasonably good, which matches the conclusion we got earlier when we evaluated it using the **`mean_squared_error()`** and **`r2_score()`** functions.

Chapter 7 – Classification

In the previous chapter, we learned to use regression to make predictions where the target variable can take on continuous values. In this chapter, let's move on to discuss another important application of machine learning - classification.

7.1 What is Classification?

Classification is very similar to regression, except that instead of predicting continuous values, classification is concerned with the prediction of discrete values. For instance, we can use classification algorithms to predict whether an email is spam or non-spam. These discrete values (such as “Spam” and “Non-Spam”, “Cat”, “Dog”, and “Rabbit” etc.) are known as classes.

There are many machine learning algorithms for classification. Commonly used ones include decision tree, random forest, support vector machine (SVM), and logistic regression.

In this chapter, we’ll discuss three algorithms - decision tree, random forest, and SVM.

7.2 Decision Tree

Decision trees are some of the most ubiquitous algorithms in machine learning. The basic concept of a decision tree is to make classification predictions by tracing through a series of if-else rules in an “upside-down tree”, similar to the tree shown below:

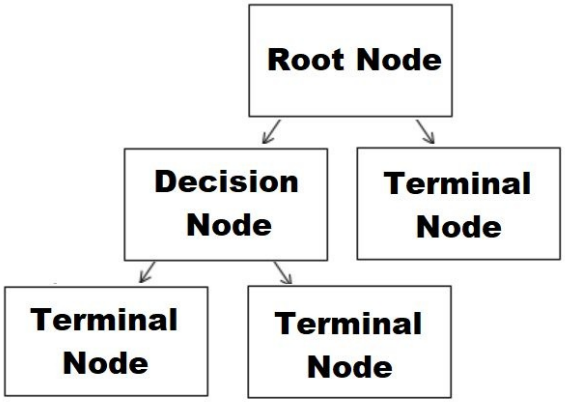

Figure 7.1: Decision tree

Training a decision tree consists of iteratively splitting the dataset into two subsets based on the rules in the root and decision nodes. Contrary to traditional programming where these rules are determined manually, a decision tree determines its rules autonomously by computing the level of impurity in each tree node.

A node’s impurity level is calculated using either the Gini impurity or entropy score, both of which lead to similar trees in most cases. An impurity of 0 occurs when all instances in a node belong to the same class. When that happens, we say that the node is “pure”.

The basic idea behind a decision tree is to construct a tree by selecting rules with the smallest weighted impurities. Let’s use an example to illustrate. Suppose we have the dataset below and we want to predict whether it is safe to eat an unknown animal given its hairiness, size, and weight:

	Hairiness	Size	Weight	Class
A	Not Hairy	Large	3.8	Safe
B	Not Hairy	Large	1.2	Safe
C	Not Hairy	Small	3.5	Safe
D	Not Hairy	Small	3.2	Dangerous
E	Hairy	Large	5.4	Dangerous
F	Hairy	Small	3.6	Dangerous

sklearn requires us to encode non-numerical categorical features before applying the decision tree algorithm. To start constructing the tree, the decision tree algorithm considers different split points of all the features in the dataset and chooses the feature and split point that give us the best result.

One way to choose the split points is to sort the values of each feature and use the mean of two adjacent values. For instance, for the **`Weight`** feature, if we sort the values, we get 1.2, 3.2, 3.5, 3.6, 3.8, and 5.4. Therefore, we can split the dataset using the values 2.2 (i.e., the mean of 1.2 and 3.2), 3.35, 3.55, 3.7, and 4.6.

A possible rule can then be:

If Weight <= 2.2, classify the instance into Subset 1. Else, classify it into Subset 2.

For the **`Hairiness`** feature, we can encode “Not Hairy” as 0 and “Hairy” as 1. A possible rule is:

If Hairiness <= 0.5, classify the instance into Subset 1. Else, classify it

into Subset 2.

Let's compute the weighted impurity for the *Hairiness <= 0.5* rule above. The table below shows the breakdown of the classification:

		Classification	
		Subset 1	Subset 2
True Class Label	Safe	A, B, C	-
	Dangerous	D	E, F

We will use the Gini impurity measure to illustrate. Gini impurity is given by the formula

$$Gini\ impurity = 1 - sum\ of\ P(k)^2\ for\ all\ the\ classes,$$

where $P(k)$ stands for the proportion of instances that belong to class k in a particular subset.

Don't worry if this sounds confusing, an example will make it clearer.

In our example, there are two classes (**`Safe`** and **`Dangerous`**) and two subsets (Subset 1 and Subset 2). Let's calculate the Gini impurity for the first subset (i.e., Subset 1).

There are four instances in this subset. Out of these four instances, three belong to the **`Safe`** class and one to the **`Dangerous`** class. Therefore,

P(Safe) = 3/4 and P(Dangerous) = 1/4

To get the Gini impurity for this subset, we sum the squares of these two proportions and subtract the sum from 1.

In other words, the Gini impurity for Subset 1
$= 1 - [(3/4)^2 + (1/4)^2]$
$= 0.375$

Next, let's compute the Gini impurity for Subset 2. Out of the two instances in this subset, all belong to the **`Dangerous`** class.

Therefore, Gini impurity for Subset 2
$= 1 - [(2/2)^2 + (0/2)^2]$
$= 0$

As expected, we get an impurity of 0 for this subset as all instances in the subset belong to the same class.

After we calculate the Gini impurities, we can calculate the *weighted impurity* of the *Hairiness <= 0.5* rule by multiplying the Gini impurity of each subset with its relative size. In our example, there are four (out of six) instances in Subset 1 and two in Subset 2.

Therefore, weighted impurity
$= (4/6)*(0.375) + (2/6)*(0)$
$= 0.25$

To construct a decision tree, the decision tree algorithm computes the weighted impurities of all possible rules and chooses the rule with the lowest weighted impurity. Once it finds the rule, it splits the dataset into two subsets using it.

In our example, the *Hairiness <= 0.5* rule gives us the smallest weighted impurity. Suppose we encode the class **`Safe`** as **`0`** and **`Dangerous`** as **`1`**, after the first split, we have the following tree:

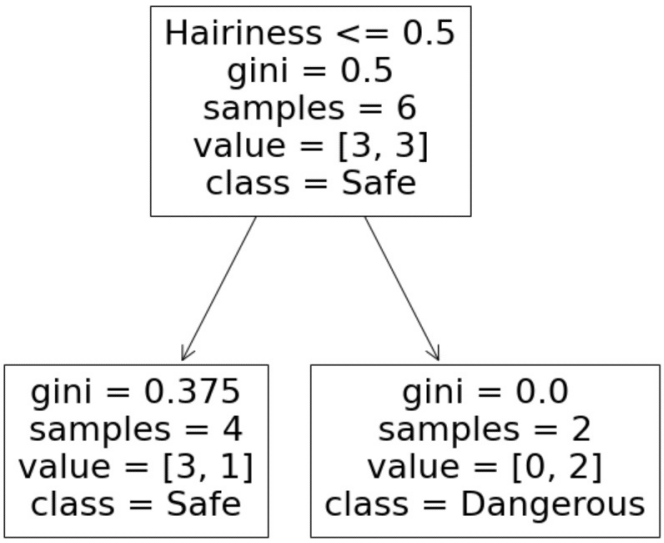

Figure 7.2: Decision tree after 1 split

The subset on the left satisfies the rule in the node above. In our example, this subset satisfies the *Hairiness <= 0.5* rule; it has 4 samples (i.e., 4 instances) and a Gini impurity of 0.375.

The subset on the right does not satisfy the *Hairiness <= 0.5* rule. Instead, it satisfies the *Hairiness > 0.5* rule. This subset has 2 samples and a Gini impurity of 0.

`value` gives us the number of instances in each class within a subset, with the classes arranged in ascending numerical order.

For instance, for the left subset, **`value=[3, 1]`** tells us that 3 instances in this subset have a class label of **`0`** (i.e., **`Safe`**) and 1 has a label of **`1`** (i.e., **`Dangerous`**). Since there are more instances with a label of **`Safe`**, the predicted label for this subset is **`Safe`**.

As the left subset is not pure yet, we can split it further. The algorithm applies the same logic described above to split the subset iteratively until it reaches the maximum depth (specified by us), or it cannot find a split that further reduces impurity. When that happens, the algorithm terminates and the tree is complete.

7.3 Random Forest

The decision tree algorithm is an intuitive algorithm that is very easy to interpret. However, a common problem with decision trees is that they tend to be unstable. A slight change in the dataset can significantly change a tree's structure. As such, decision trees are relatively inaccurate when making predictions.

One way to resolve this issue is to use an ensemble of decision trees to make multiple predictions and return the final prediction based on an aggregation function. This approach is called Random Forest and is a robust algorithm that gives better performance than a single decision tree.

For the sake of illustration, let's suppose we have a training set with five instances A, B, C, D, and E. We can randomly select 4 instances from this training set to train a decision tree. Next, we select another 4 instances to train a second decision tree, followed by another 4 to train a third decision tree, and so on.

Instances can be selected with or without replacement. Selecting with replacement means an instance may be chosen more than once for the same decision tree.

For instance, to train the first decision tree, we may select instances A, B, E, and A. Here, instance A is chosen more than once to train the tree. This selection process is called bagging (short for bootstrap aggregating) and is the default process used in sklearn.

Selecting without replacement, on the other hand, means an instance cannot be chosen more than once for the same decision tree. This process is known as pasting.

After we select the instances and train the trees, we can use them to predict the class of a new instance. The most frequently predicted class is typically chosen to be the final prediction. For example, suppose we train four trees and the predictions are class 1, 1, 2, and 1; the final prediction is class 1.

One significant difference between the random forest and decision tree algorithm is that the former does not consider all the features in our dataset when splitting a node. Instead, it searches for the best feature among a random subset of features. For instance, if our dataset has 16 features, it may randomly consider 4 features to choose the best split. This randomness leads to greater diversity in the trees, which generally results in a better-performing model.

7.4 Decision Tree and Random Forest with Scikit-Learn

Now that we understand how the decision tree and random forest algorithms work, let's build their models in sklearn.

In the example that follows, the goal is to predict whether a person has diabetes, given his glucose level and Body Mass Index (BMI). The dataset is stored in a file called *diabetes.csv*, and we are interested in using **`Glucose`** and **`BMI`** to predict **`Outcome`**.

Let's first import pandas and read the data file:

```
import pandas as pd
df = pd.read_csv('diabetes.csv')
```

Next, let's examine the data:

```
df.head()
```

	Glucose	BMI	Outcome
0	141	33.02	0
1	83	30.20	0
2	128	27.16	0
3	112	35.16	1
4	71	27.22	0

Figure 7.3: The diabetes dataset

```
df.isnull().sum()
```

```
Glucose    0
BMI        0
Outcome    0
```

```
dtype: int64
```

Now, we need to split the dataset into training and test sets:

```
X = df.iloc[:,:-1].to_numpy()
y = df.iloc[:,-1].to_numpy()
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)
```

Here, we first use **`iloc`** to select all but the last column in **`df`** and use **`to_numpy()`** to convert the resulting DataFame to a NumPy array; we then assign the array to **`X`**. Next, we use **`iloc`** again to select the last column, convert it to a NumPy array, and assign the array to **`y`**. (Refer to Chapter 3 Section 3.8.3 if you need help with the code above.)

Most methods in sklearn accept both pandas DataFrames and NumPy arrays as input. In this example, we show how to use NumPy arrays, which are typically faster than DataFrames.

After selecting our features and target variable, we pass **`X`** and **`y`** to the **`train_test_split()`** method to split our dataset into training and test subsets. Now, we can train our decision tree using the **`fit()`** method:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0)
clf.fit(X_train, y_train)
```

First, we need to import the **`DecisionTreeClassifier`** class from the **`sklearn.tree`** module. Next, we instantiate a **`DecisionTreeClassifier`** object, passing **`random_state = 0`** to the constructor to control the randomness of the estimator.

At times, several splits can have the same weighted impurity. When that happens, the estimator randomly selects one split. When we pass **`random_state = 0`** to the constructor, we ensure that the same split is selected if we run the code more than once. In other words, you will get the same tree if you run the code yourself.

We can plot the resulting tree using a predefined function called **`plot_tree()`**, found in the **`sklearn.tree`** module:

```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.tree import plot_tree
plt.figure(figsize=(20,10))
plot_tree(clf, feature_names=['Glucose', 'BMI'], class_names=['No',
'Yes'])
plt.show()
```

To plot the tree, we pass the tree classifier (**`clf`**) and two parameters - **`feature_names`** and **`class_names`** - to the function.

`feature_names` stores the names of the features in the dataset and is passed as a list of strings, sorted based on the order of the columns in the dataset.

`class_names` is also passed as a list of strings. These names correspond to the classes in the dataset, sorted in ascending numerical order.

Our dataset has two classes - **`0`** and **`1`**, with **`0`** indicating the absence of diabetes and **`1`** indicating otherwise. Therefore, we use the names **`'No'`** and **`'Yes'`** for the two classes, respectively, and pass **`class_names= ['No', 'Yes']`** to the **`plot_tree()`** function.

If you run the code above, you'll get the following output. Do not worry if you can't read the tree, we will not be using it:

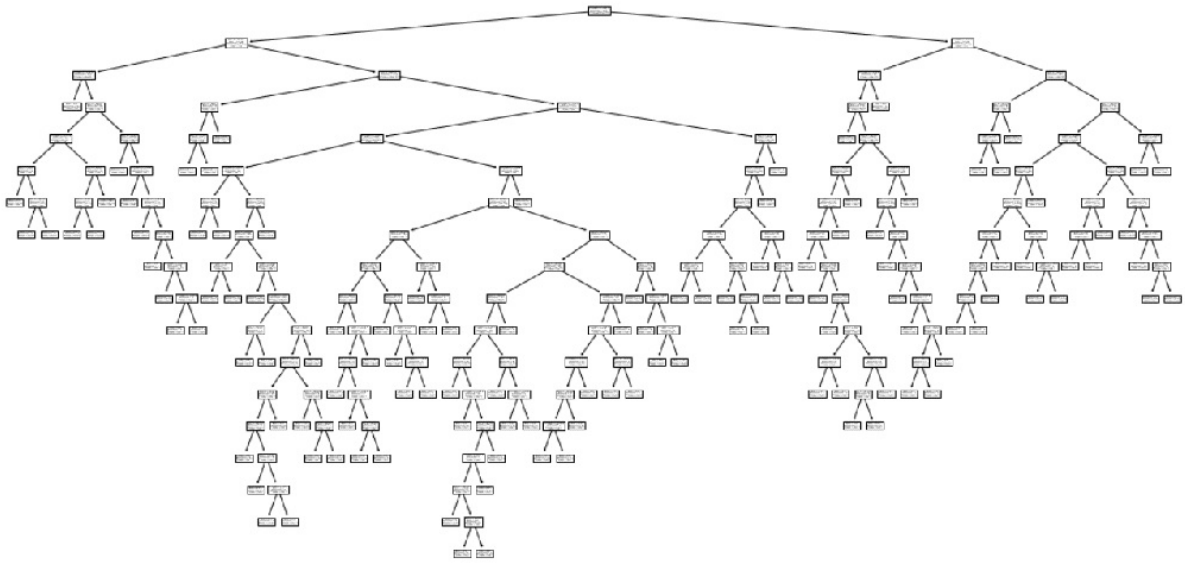

Figure 7.4: Full decision tree for the diabetes dataset

This tree is massive with many levels and is almost certain to overfit. Overfitting occurs when the model fits our training dataset so well that it picks up noise or random fluctuations in the training data. Such a model is unlikely to generalize well to new data.

To prevent overfitting, we can "prune" the tree by limiting its depth. We do that by specifying the **`max_depth`** hyperparameter when instantiating the tree object. A hyperparameter refers to a parameter whose value is specified by us when we train our models. This contrasts with other parameters (such as the coefficients of a linear regression model) whose values are derived by the model during the learning process.

If we have already instantiated the object, we can use the **`set_params()`** method to update the hyperparameter.

```
clf.set_params(max_depth = 3)
```

Let's train and plot the tree again:

```
clf.fit(X_train, y_train)
plt.figure(figsize=(20,10))
plot_tree(clf, feature_names=['Glucose', 'BMI'], class_names=['No',
'Yes'])
plt.show()
```

This gives us the following output:

(You can download this image and all the other images in this book at https://www.learncodingfast.com/machine-learning.)

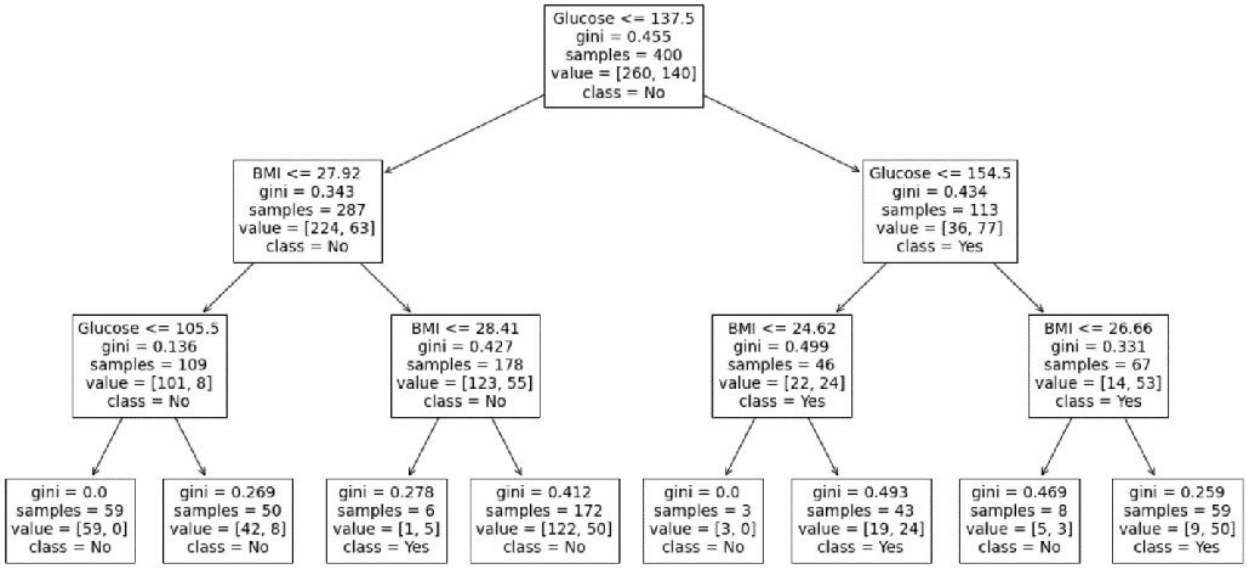

Figure 7.5: Decision tree for the diabetes dataset with three levels

We can now use the **`predict()`** method to predict the classes of new instances:

```
clf.predict([[90, 20], [200, 30]])
```

Here, we pass two instances - **`[90, 20]`** and **`[200, 30]`** - to the **`predict()`** method. The first instance (Glucose = 90, BMI = 20) satisfies the left-most branch of the tree, while the second (Glucose = 200, BMI = 30) satisfies the right-most.

If you run the code above, you'll get **`array([0, 1])`** as the output.

To see how well our decision tree performs, let's do a cross-validation on the training set:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(clf, X_train, y_train, cv=5,
scoring="accuracy")
accuracy = scores.mean()
accuracy
```

Here, we use **`"accuracy"`** as the scoring criterion. This gives us **`0.7125`** as the output, which is acceptable but not great. Let's see if we can improve the accuracy using a random forest model.

```
from sklearn.ensemble import RandomForestClassifier
clf_rf= RandomForestClassifier(n_estimators = 200, max_depth=3,
random_state=0)
```

The code above should be self-explanatory by now. We first import the **`RandomForestClassifier`** class from the **`sklearn.ensemble`** module and instantiate a new **`RandomForestClassifier`** object.

The **`max_depth`** and **`random_state`** parameters are the same as those of a decision tree. The only additional parameter - **`n_estimators`** - specifies the number of trees to use for the forest. The default is 100.

Let's train and evaluate the random forest model now:

```
clf_rf.fit(X_train, y_train)
scores = cross_val_score(clf_rf, X_train, y_train, cv=5,
scoring="accuracy")
accuracy = scores.mean()
accuracy
```

This gives us **`0.725`** as the output, which is slightly better than the accuracy for the decision tree model.

In the next section, we'll see if we can improve this score using another classification algorithm - the Support Vector Machine algorithm.

7.5 Support Vector Machine

Support Vector Machine (SVM) is one of the most powerful and versatile machine learning algorithms for both classification and regression problems.

In this section, we'll focus on using it for classification. We'll first discuss how it works for binary classification before discussing how it handles multi-class classification.

7.5.1 Binary Classification

Binary classification refers to the task of classifying instances into two classes.

To perform binary classification, the SVM algorithm tries to find a boundary (also known as a hyperplane) that separates the instances into two classes, such that the boundary is as far away from the closest training instances as possible.

As an example, let's consider the figure below where we need to separate the rectangles and circles. Two possible hyperplanes - the solid and dotted lines - are shown.

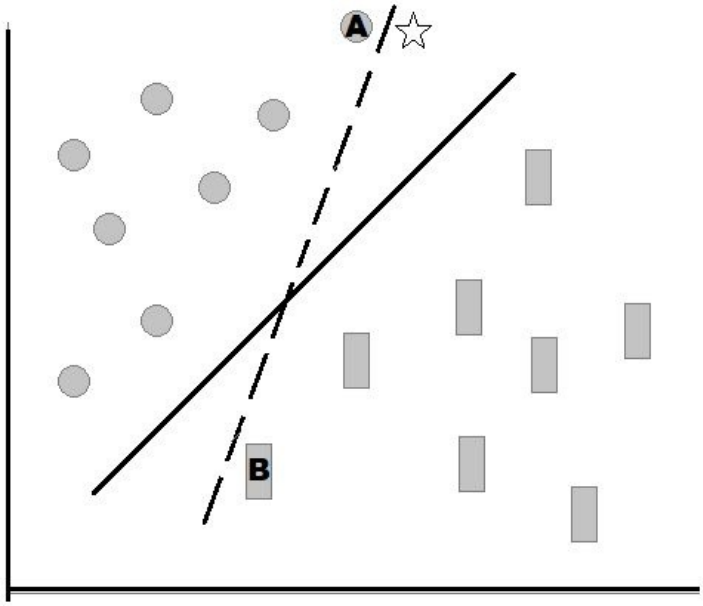

Figure 7.6: Comparing two hyperplanes

Both hyperplanes separate the circles and rectangles correctly. However, the dotted line is very close to circle A and rectangle B and is not likely to generalize well to new data. For instance, if we add an unknown shape represented by the star above, the dotted line will classify it as a rectangle.

Visually, we would agree that the unknown shape is more likely to be a circle since it is closer to the circles. The solid line will classify it as a circle, which is likely the correct classification.

The SVM algorithm works by finding a hyperplane that maximizes the distance between the plane and the closest data points. These data points are known as support vectors, and the distance between the two classes' support vectors is known as the margin.

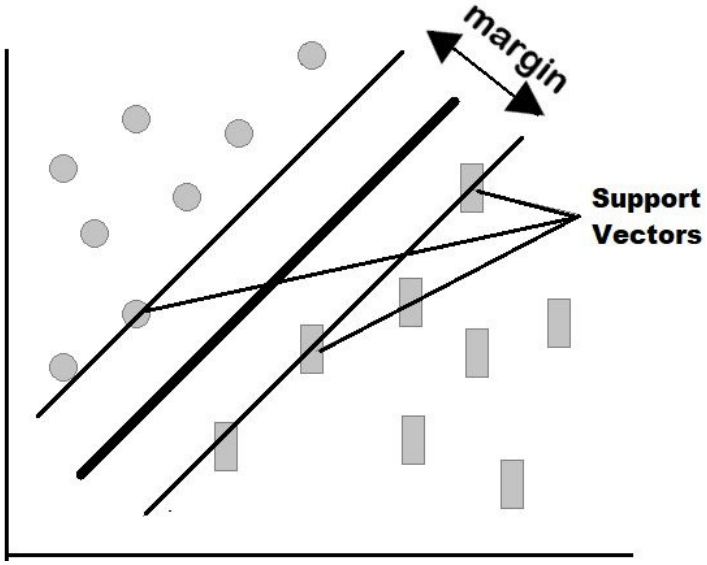

Figure 7.7: Margin and support vectors

Non-Linear Kernel

In the example above, our data was linearly separable (i.e., the points can be separated by a linear line). If the data was more convoluted and a straight line could not separate the two classes well, we need to use a non-linear SVM model.

Recall that to solve a similar problem of non-linearity in regression, we

went with polynomial regression instead of linear regression. The idea is the same here. To separate data that is not linearly separable, we can map the data to a higher-dimensional space.

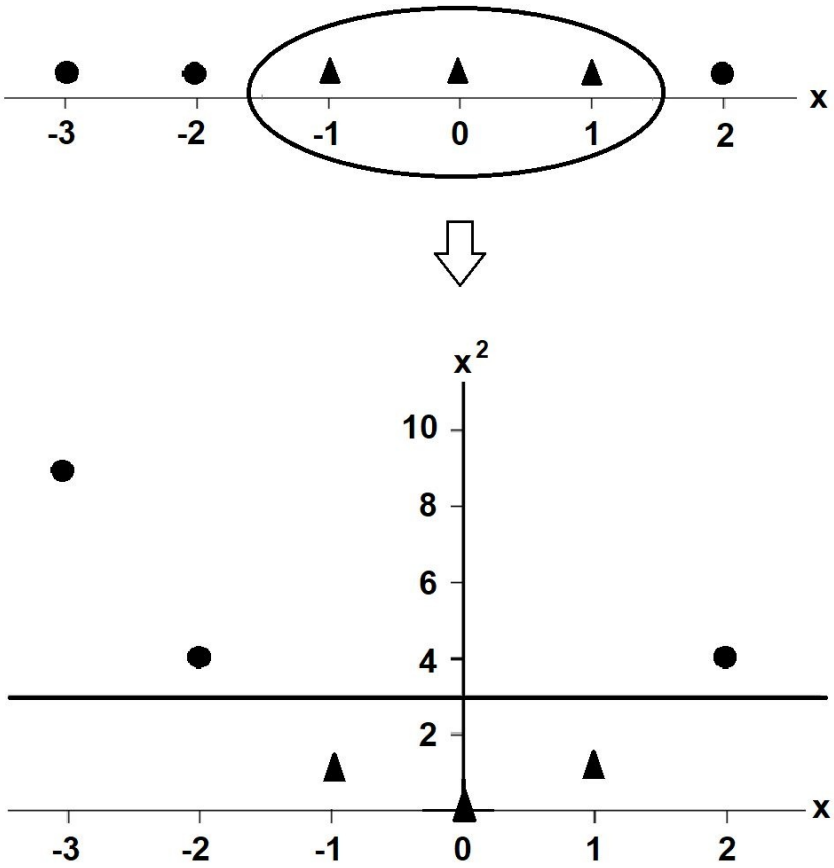

Figure 7.8: SVM for non-linearly separable data

In the figure above, the dataset on top (represented by x) is not separable by a linear line. However, we can map it to a 2D space by adding a new dimension x^2 to it. For instance, the first data point $x = -3$ can be mapped to $x = -3$ and $x^2 = 9$. When we do that, the dataset becomes linearly separable.

To map our dataset to a higher dimension, we used the **`PolynomialFeatures`** class in Chapter 6 for polynomial regression. Unfortunately, this approach tends to be computationally expensive and infeasible for the SVM algorithm.

For SVM, we need to use what is known as a kernel trick. Without going

into the mathematics behind it, what we need to know is that a kernel trick provides us with an efficient and less expensive way to transform data into higher dimensions.

To use the kernel trick, we pass a **`kernel`** hyperparameter to the SVM model when instantiating it. The kernels available in sklearn are **`'linear'`**, **`'poly'`**, **`'rbf'`**, **`'sigmoid'`**, and **`'precomputed'`**; the default is **`'rbf'`**.

We'll demonstrate how to use this hyperparameter when we train a SVM model later.

7.5.2 Multi-class Classification

Before we do that, let's discuss how we can use SVM for multi-class classification.

The SVM algorithm is fundamentally a binary classifier. In other words, it is only capable of classifying instances into two classes. This contrasts with algorithms like decision tree and random forest, which can handle multi-class classification natively.

If we want to use a binary classifier to classify multiple classes, we can use one of two heuristics methods. Both methods involve splitting the multi-class problem into multiple binary classification problems.

One-vs-Rest (OvR)

The first is the One-vs-Rest method. Suppose we need to classify our dataset into three classes - 1,2 and 3, the OvR method splits the problem into the following binary classification problems:

- Class-1 vs. Not-Class-1 (i.e., Class-2 and Class-3 are combined into Not-Class-1)
- Class-2 vs. Not-Class-2
- Class-3 vs. Not-Class-3

A binary classifier is trained on each problem, and a decision score is generated for each classifier. The output of the classifier with the highest

score is selected. For instance, suppose the first classifier predicts Class-1 with a score of 0.15, the second predicts Class-2 with a score of 0.21, and the third predicts Class-3 with a score of 0.98, the class with the highest score (i.e., Class-3) is selected.

One-vs-One (OvO)

Next, we have the One-vs-One method. This method involves training a binary classifier for each pair of classes and selecting the class with the most votes. With reference to the example above, the classifiers would classify:

- Class-1 vs. Class-2
- Class-1 vs. Class-3
- Class-2 vs. Class-3

If the first classifier predicts that an instance belongs to Class-2, the second predicts it belongs to Class-1, and the third predicts Class-2, our final prediction would be Class-2 (majority wins). In the event of a tie, the OvO method uses a decision function based on the confidence levels of the underlying binary classifiers to make a prediction.

sklearn provides two classes - **`OneVsOneClassifier`** and **`OneVsRestClassifier`** - for the OvO and OvR methods, respectively.

In most cases, there is no need to use these classes explicitly. When sklearn detects that we are trying to predict multiple classes using a binary classification algorithm, it automatically runs one of the two methods for us, depending on the algorithm.

For instance, when we use SVM to classify multiple classes, sklearn uses the OvO method; this allows the SVM algorithm to work seamlessly with multi-class classification problems.

7.6 SVM with Scikit-Learn

Now that we understand the intuition behind the SVM algorithm, let's train a SVM model in Scikit-Learn.

To do that, we use the **`SVC`** class (which stands for "Support Vector Classification") in the **`sklearn.svm`** module. For this example, we'll pass **`kernel='rbf'`** (the default kernel) and **`random_state = 0`** to the **`SVC()`** constructor.

SVM requires the input features of our dataset to lie on the same scale. Therefore, we'll use a pipeline to scale our features before passing them to the **`SVC`** estimator. The code below shows how it can be done:

```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline

scaler = ('scaler', StandardScaler())
clf_svc = ('SVM', SVC(kernel='rbf', random_state=0))
pipeline = Pipeline([scaler, clf_svc])
```

After creating the pipeline, we can use its **`fit()`** method to train our model:

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

Now, let's use the model to make some predictions:

```
pipeline.predict([[90, 20], [200, 30]])
```

If you run the code above, you'll get **`array([0, 1])`** as the output. Next, let's evaluate the model on the training subset using cross-validation:

```
scores = cross_val_score(pipeline, X_train, y_train, cv=5,
scoring="accuracy")
accuracy = scores.mean()
accuracy
```

This gives us **`0.745`** as the output, which is slightly better than both the random forest and decision tree models. Since the SVM model performs

the best, let's select this model and test how well it performs on the test set:

```
from sklearn.metrics import accuracy_score
y_pred = pipeline.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
accuracy
```

Here, we get `0.74` as the output, which is very close to the accuracy we got on the training set. This score indicates that our SVM model generalizes well to the test set.

Chapter 8 - Clustering

In the last two chapters, we dealt with datasets in which the label of each instance is provided.

For instance, the *housing.csv* dataset includes the price of each house, and the *diabetes.csv* dataset includes the diabetic status of each person.

We mentioned earlier that learning from datasets with labels is known as supervised learning. In this chapter, let’s move on to unsupervised learning.

In unsupervised learning, we only have access to the features of an instance and not its label. Therefore, the goal of unsupervised learning is not to predict the labels. Instead, the goal is to discover hidden patterns in the dataset.

8.1 What is Clustering?

One widely used application of unsupervised learning is clustering. In clustering, the goal is to aggregate instances into a few cohesive groups.

Doing so can help us solve a diverse array of problems. For instance, we can use clustering to segment customers into different groups based on their shopping preferences. We can also use it to detect outliers (by detecting instances that are not similar to any known clusters), reduce the number of features in our training set, or compress images.

8.2 K-Means Clustering

One of the most commonly used algorithms for clustering is k-means clustering. The intuition behind this algorithm is quite simple.

To perform k-means clustering, we must first decide on the number of clusters (k) we want. Next, we choose the centroids of these clusters randomly and perform the following steps iteratively:

First, we assign each training instance to a cluster based on its proximity to the cluster's centroid. Next, we compute the mean of all the data points assigned to a cluster and use that to update the centroid. After updating the centroids of all the clusters, we reassign the points and repeat the updating and assigning steps until all the centroids stop changing.

Let's use a small dataset with five points - (2,2), (3,2), (3,1), (1,1), and (0,0) - to illustrate.

If we want to cluster these points into two clusters, we need to select two centroids randomly. Suppose we choose (1.5, 1.5) and (3, 1.5) as the centroids and name the clusters C1 and C2, respectively.

Now, we need to assign each data point to the cluster with the nearest centroid. To determine how close a point is to a centroid, we commonly use the Euclidean distance formula mentioned in Chapter 5. The table below shows the distances of each point from the two centroids.

Centroid of C1 = (1.5, 1.5)
Centroid of C2 = (3, 1.5)

Data Point	Distance from centroid of C1	Distance from centroid of C2	Assigned to
(2,2)	0.707	1.12	C1
(3,2)	1.58	0.5	C2
(1,1)	0.707	2.06	C1
(3,1)	1.58	0.5	C2
(0,0)	2.12	3.35	C1

Based on the distances shown above, we'll assign (2, 2), (1,1), and (0, 0) to C1, and (3, 2) and (3, 1) to C2. Now, we need to calculate the mean of the data points in each cluster and use that to update the centroids.

New Centroid of C1

$$= \left(\frac{2+1+0}{3}, \frac{2+1+0}{3}\right) = (1, 1)$$

New Centroid of C2

$$= \left(\frac{3+3}{2}, \frac{2+1}{2}\right) = (3, 1.5)$$

The centroid of C1 changes to (1, 1), while that of C2 stays the same. Let's calculate the distances of our data points from the new centroids and reassign them.

Data Point	Distance from new centroid of C1	Distance from new centroid of C2	Assigned to
(2,2)	1.41	1.12	C2
(3,2)	2.24	0.5	C2
(1,1)	0	2.06	C1
(3,1)	2	0.5	C2
(0,0)	1.41	3.35	C1

Most of the points remain in the same cluster except for (2, 2). We need to recalculate the mean of the data points in the two clusters and update the centroids.

We repeat the process of updating the centroids and reassigning data points until the clusters no longer change. When that happens, we say that the algorithm has converged and the algorithm ends.

8.2.1 Centroid initialization methods

k-means clustering is a simple and easy-to-understand algorithm. However, when we use k-means clustering, there are two things we need to do.

First, we need to choose the initial centroids. Choosing the initial centroids is known as centroid initialization, and poor initialization can lead to suboptimal solutions.

For instance, the chart below shows two identical sets of data points clustered differently due to different initial centroids (A, B, and C vs. X, Y, and Z). While both solutions give us three clusters, the solution on the right is better as the points within each cluster are closer to their centroids.

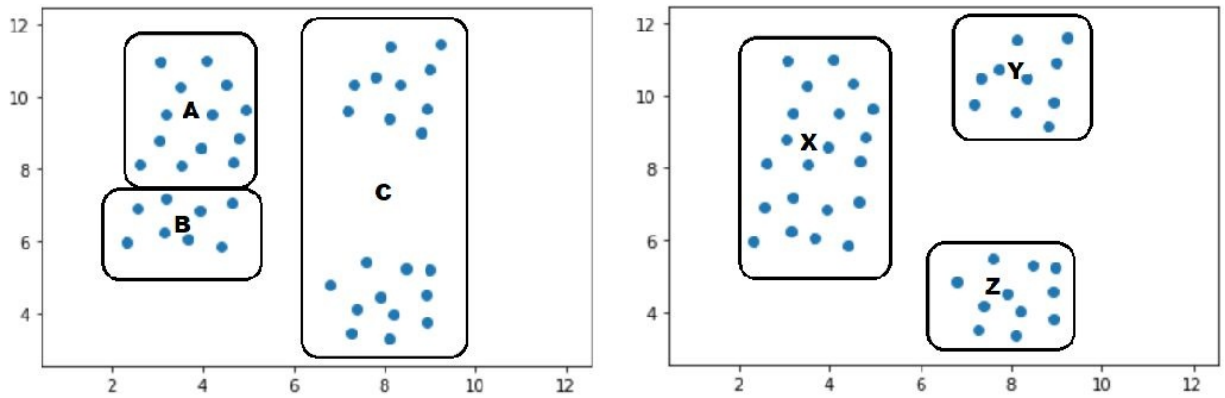

Figure 8.1: Centroid initialization

There are two ways to select the initial centroids.

The first way is to run the entire k-means algorithm multiple times with different randomly selected initial centroids. For each run, we compute the sum of the squared distances between each instance and the centroid of its final cluster. This sum is known as the inertia of the model. We then compare the inertias of all the runs and choose the run with the smallest inertia.

Another way is to select the initial centroids using a technique known as k-means++. This technique uses additional computation to determine centroids that are distant from one another and has been shown to result in faster convergence and better results. sklearn uses k-means++ by default.

8.2.2 Determining the Number of Clusters

Besides choosing the initial centroids, another task that we need to do is

decide on the number of clusters.

One way to do this is to visually determine the number of clusters by plotting the dataset on a scatter plot. However, this only works if we have a simple dataset that can be easily plotted.

A better approach is to use the elbow method. This method involves running the k-means algorithm multiple times with different numbers of clusters and calculating the inertia of each run. The goal is to find an optimal number of clusters where the decrease in inertia becomes less significant with added clusters.

In the next section, we'll demonstrate how to use the elbow method to determine the optimal number of clusters. We'll also learn to train a k-means model using sklearn.

8.3 K Means Clustering with Scikit-Learn

The dataset that we'll be using to train our model is stored in a file called *clustering.csv*. Let's load the dataset and explore it:

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

df = pd.read_csv('clustering.csv')

print(df.isnull().sum())
df.head()
```

This gives us the following output:

```
X1    0
X2    0
dtype: int64
```

	X1	X2
0	-4.581081	-8.286462
1	-0.753894	1.864580
2	9.019485	0.356573
3	-3.293414	6.087256
4	7.042307	1.157019

Figure 8.2: The clustering dataset

This dataset consists of two features - **`X1`** and **`X2`**, and our goal is to use both features to cluster the instances.

As the dataset is not labeled, we do not need to split it into training and test subsets because there are no true labels to evaluate against our predicted values. We also do not need to do any data preprocessing as

there are no missing values or textual data in the dataset. In addition, the range of **`X1`** and **`X2`** do not differ much, as shown in the output below:

```
df.describe()
```

	X1	X2
count	500.000000	500.000000
mean	-1.560198	1.453822
std	5.077718	6.713320
min	-11.043486	-11.727868
25%	-5.728146	-5.127837
50%	-2.371867	1.758263
75%	3.220747	7.714326
max	10.797640	16.719749

Figure 8.3: Statistics for the clustering dataset

Now, let's do a scatter plot of the dataset:

```
plt.scatter(df['X1'], df['X2'])
plt.xlabel('X1')
plt.ylabel('X2')
```

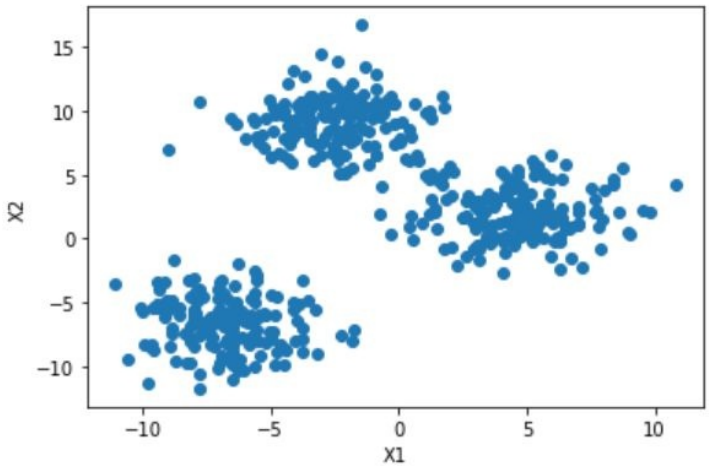

Figure 8.4: Scatter plot for the clustering dataset

Visually, we can see that there seem to be three clusters in our dataset. Let's use the elbow method to determine if 3 is indeed the optimal number of clusters:

```
from sklearn.cluster import KMeans
inertias = []

for k in range(2, 20):
    kmeans = KMeans(init='k-means++', random_state=0, n_clusters=k)
    kmeans.fit(df)
    inertias.append(kmeans.inertia_)

plt.figure(figsize=(12, 4))
plt.grid()
plt.plot(range(2, 20), inertias)
plt.xticks(range(0, 20))
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.show()
```

Here, we first import the **`KMeans`** class from the **`sklearn.cluster`** module. Next, we declare an empty list called **`inertias`**.

After declaring the list, we use a **`for`** loop to run the k-means algorithm 18 times (from k = 2 to k = 19). We instantiate a **`KMeans`** object for each run and pass **`n_clusters=k`** to the constructor to specify the number of clusters and **`random_state = 0`** for reproducibility of results.

We also pass **`init = 'k-means++'`** to specify the centroid initialization method. This specification is optional as k-means++ is the default initialization method used by sklearn. However, if we want to use the random initialization method instead, we need to pass **`init = 'random'`** to the constructor.

After instantiating the **`KMeans`** object, we use it to train our model and append the resulting inertia (stored in the **`inertia_`** attribute) to the **`inertias`** list.

After training all the models, the **`for`** loop ends. Outside the loop, we use the **`plot()`** method to plot the inertias on the y-axis against the number of clusters on the x-axis.

If you run the code above, you'll get the following elbow plot.

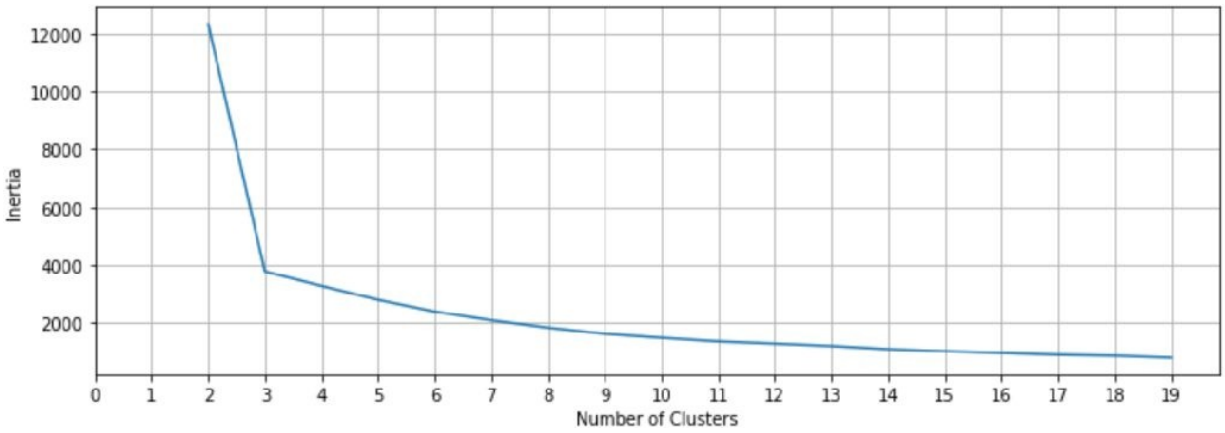

Figure 8.5: Elbow plot

This plot shows clearly that the inertia drops as the number of clusters increases. In fact, if we have 500 clusters, the inertia will be zero. This is because there are 500 data points in our dataset. If each point is its own cluster, we will not have any inertia.

However, having 500 clusters is useless and defeats the purpose of clustering in the first place. So instead of aiming for an inertia of zero, we strive to find an optimal number of clusters such that the drop in inertia goes from being quite significant to being not as substantial.

This happens when we have three clusters. From 2 to 3 clusters, the inertia dropped from above 12000 to below 4000. From 3 to 4 clusters, the drop is much less substantial. Therefore, 3 is a reasonable number of clusters to form.

After getting the number of clusters, we can rerun our k-means algorithm to determine the clusters' centroids (stored in the **`cluster_centers_`** attribute):

```
kmeans = KMeans(random_state=0, n_clusters=3)
kmeans.fit(df)
print(kmeans.cluster_centers_)
```

This gives us the following output:

```
[[-2.56005145  9.07257054]
[-6.78277285 -6.63304128]
 [ 4.58815239  1.91636363]]
```

sklearn automatically assigns an index to each cluster. As there are three clusters in our example, the indexes are 0, 1, and 2. To get the cluster of an instance, we use the **`predict()`** method:

```
# Getting the cluster for the first row in df
print(kmeans.predict(df.loc[[0]]))

# Getting the cluster for a new point (-2, 10)
print(kmeans.predict([[-2, 10]]))
```

Here, we use **`df.loc[[0]]`** to select the first row in **`df`** and pass this array to the **`predict()`** method. We also pass a new point (-2, 10) as a 2D array to the method. If you run the code above, you'll get the following output:

```
[1]
[0]
```

The first row in **`df`** is assigned to cluster 1, while the point (-2, 10) is assigned to cluster 0.

If you want to get the clusters of all the points in **`df`**, you can use the **`labels_`** attribute:

```
df['Cluster'] = kmeans.labels_
df.head()
```

Here, we add a column called **`Cluster`** to our **`df`** DataFrame and assign **`kmeans.labels_`** to it. If you print the first five rows in **`df`** now, you'll get the following output:

	X1	X2	Cluster
0	-4.581081	-8.286462	1
1	-0.753894	1.864580	2
2	9.019485	0.356573	2
3	-3.293414	6.087256	0
4	7.042307	1.157019	2

Figure 8.6: Clusters assigned to the instances

We can plot the three clusters (0, 1, and 2) on the same scatter plot using the code below:

```
plt.figure(figsize=(16, 10))
markers = ['x', '.', '+']

for i in range(0, 3):
    cond = df['Cluster'] == i
    df2 = df[cond]
    plt.scatter(df2['X1'], df2['X2'], label = 'Cluster '+str(i), marker
= markers[i])

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,
1], color = 'black', s = 50, marker = "s")

plt.xlabel('X1')
plt.ylabel('X2')
plt.legend()
```

Here, we first specify the size of our chart and declare a list called **`markers`** to store the markers for our clusters. Next, we use a **`for`** loop (that runs 3 times) to plot the clusters.

The first time the loop runs, **`i`** equals **`0`**. The condition **`df['Cluster'] == i`** becomes **`df['Cluster'] == 0`**.

This condition selects rows in **`df`** with a **`Cluster`** value of **`0`**. We assign the selected rows to a new DataFrame called **`df2`** and use the **`scatter()`** function to plot a scatter plot for these rows, passing **`label`**

`= 'Cluster 0'` and **`marker = 'x'`** to the function.

This plots all the data points in **`df`** with a **`Cluster`** value of **`0`**, using **`'x'`** as the marker.

After plotting the points for the first cluster, the **`for`** loop runs two more times to plot the other two clusters, changing the label and marker for each plot.

After plotting all three clusters, the **`for`** loop ends. Outside the **`for`** loop, we use the **`cluster_centers_`** attribute to plot the centroids of our clusters.

`cluster_centers_` stores the centroids as a NumPy array.

`cluster_centers_[:, 0]` gives us the first column of this array (i.e., the x-values for the centroids), while **`cluster_centers_[:, 1]`** gives us the second (i.e., the y-values).

If you run the code above, you'll get the following chart:

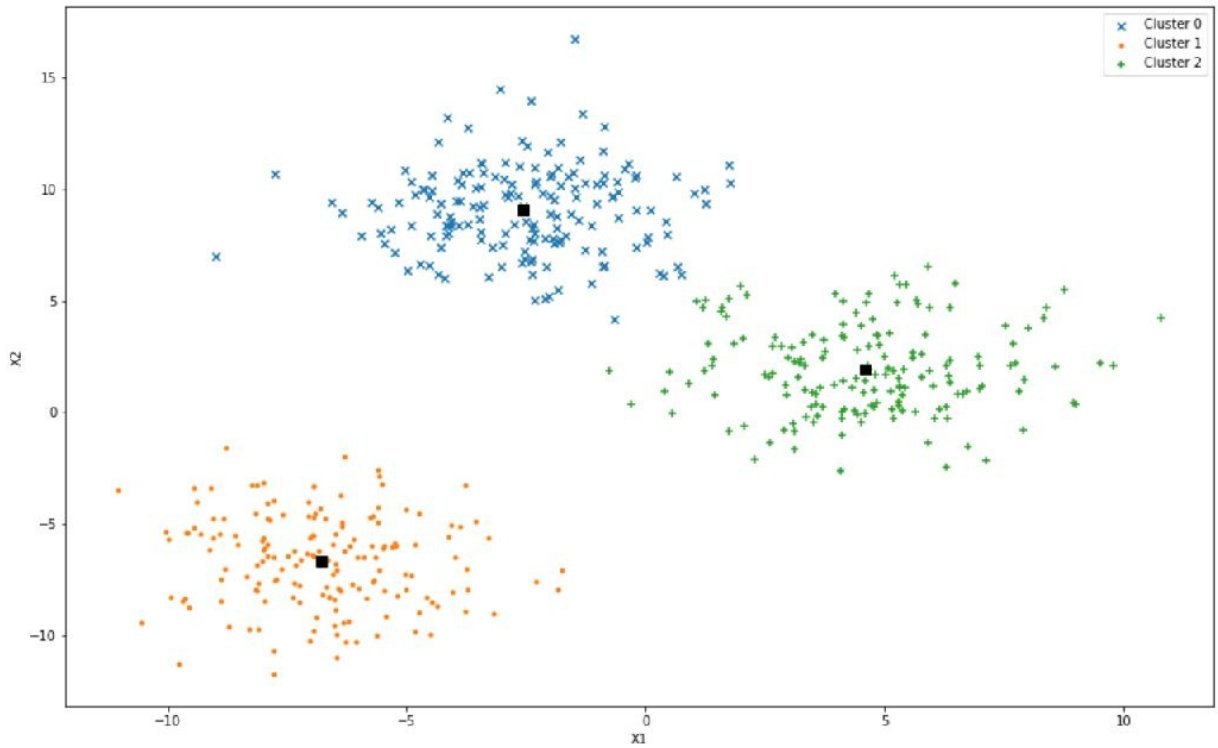

Figure 8.7: Scatter plot showing the different clusters and

centroids

Clusters 0, 1, and 2 are plotted using `'x'`, `'.'`, and `'+'`, respectively, while the centroids are plotted using squares.

Chapter 9 - Advanced Topics in Machine Learning

We've covered a lot in the previous chapters.

Before we proceed to the projects in the following three chapters, let's discuss a few advanced topics in machine learning. Specifically, we'll be discussing dimensionality reduction, overfitting and underfitting, and hyperparameter tuning in this chapter.

We'll start with dimensionality reduction.

9.1 Dimensionality Reduction

Dimensionality reduction is the process of reducing the number of features (dimensions) in our dataset.

For instance, if we have a dataset with 1000 features, a dimensionality reduction algorithm might reduce the feature set to 20 features. This reduction is possible because in most datasets, not all features are useful. For instance, some features may be correlated with others and offer little new information.

One of the most commonly used dimensionality reduction algorithms is Principal Component Analysis (PCA).

PCA reduces the number of features by identifying axes known as *components*.

The component that preserves the most variance (i.e., dispersion of points) in the data is identified first. Next, PCA identifies a second component that is orthogonal (i.e., perpendicular) to the first, and a third component that is orthogonal to the second (in a 3D space), and so on.

It repeats the process until the number of components matches the number of features. Finally, it projects the data onto the first n components, where n is a user-specified value that is typically smaller than the original number of features in the dataset.

To understand how PCA works, let's consider a dataset with two features - **`age`** and **`height`**. The data points in this dataset are represented as dots in the chart on the left of Figure 9.1 below:

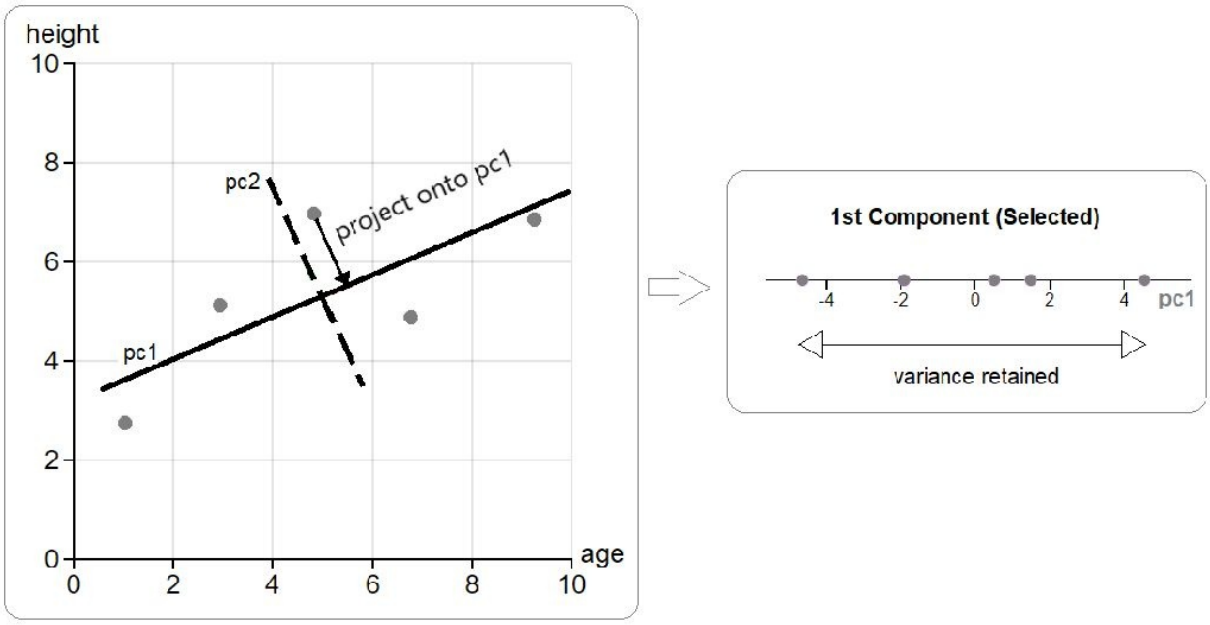

Figure 9.1: PCA components and projection

PCA requires the features in our dataset to have similar scales, as the range of features can affect how it determines the components. For instance, if one feature has a range of 0.5 to 2, while another has a range of 30 to 200, the feature with the wider range is likely to have more variance. As a result, the components determined by PCA might correspond more closely with that feature, which is not desirable.

In our example, the two features have similar scales. Therefore, we can proceed to use PCA to identify two components in the dataset.

As shown in Figure 9.1 above, the first and second components identified by the algorithm are **`pc1`** (the solid line) and **`pc2`** (the dotted line), respectively. These components are perpendicular to each other and **`pc1`** is selected as the first component because the dispersion of data points along this line is the greatest.

After identifying the components, PCA selects the most useful component(s) based on the number of components or percentage of variance we wish to retain.

For example, suppose we want to retain one component, it selects **`pc1`** and projects the points onto it. We can then train our model using **`pc1`**

instead of **`age`** and **`height`**, effectively reducing the number of dimensions from two to one.

One issue with using PCA to reduce the dimensionality of our dataset is that the components obtained by the algorithm (e.g., **`pc1`**) are not as readable and interpretable as the original features (**`age`** and **`height`**).

Nonetheless, the advantage of PCA, and dimensionality reduction in general, is that it can help us speed up the training process. In addition, it makes it easier for us to visualize the data, especially in cases where we reduce a large number of dimensions to 2 or 3.

Therefore, PCA can be a very useful tool in machine learning. We'll learn how to do it using sklearn at the end of this chapter.

9.2 Overfitting and Underfitting

Next, let's move on to discuss the concept of over and underfitting.

In the previous chapters, we discussed different algorithms for regression, classification, and clustering. Some algorithms are more complex than others.

For instance, the polynomial regression algorithm discussed in Chapter 6 uses higher degree features and is more complex than the simple linear regression algorithm. With added complexity, there is a likelihood that the resulting model will fit our training dataset better. However, there is also a risk of overfitting.

A model is said to be overfitting if it starts learning all the nitty-gritty details of the data, including the noise in the data. In this case, the model doesn't understand the data - it memorizes it. An overfitting model has poor generalization performance and is untenable for real-world situations.

To explain overfitting in simple terms, assume we have trained a machine learning algorithm to identify an apple. Apart from identifying robust features such as the color, shape, size, and texture of the apple, an overfitting model may also learn the number of speckles on the skin. As such, the model is unlikely to classify correctly when presented with an apple with a different number of speckles.

In contrast to overfitting, we have underfitting. As you might guess, underfitting occurs when our model is too simple to capture the actual trend of the data. To explain the concept of underfitting, we can again take our apple example. An underfitting model may learn that an apple is usually round but will not appreciate any other features of the apple. If we use an underfitting model, there are good chances that the model would identify something radically different (e.g., a ball) as an apple because the shapes are similar.

9.2.1 Variance and Bias

In general, underfitting models tend to have very few model parameters and are likely to exhibit high bias. High bias occurs when a large part of a model's error can be attributed to incorrect simplifying assumptions about the dataset.

For instance, in Chapter 6, we assumed a linear relationship between floor area and house price initially, when the underlying relationship was more quadratic in nature. Such incorrect assumptions can lead to a high-bias model which underfits the training data and is likely to perform poorly on both the training and test sets.

Overfitting models, on the other hand, have a large number of model parameters and are likely to exhibit low bias but high variance. High variance occurs when slight changes in the training data lead to big differences in the parameters of the model.

Overfitting models tend to memorize the training data rather than learn the dominant patterns, resulting in dramatically different parameters when trained with a different dataset. As a result, these models perform well on the training set but poorly on the test set.

The challenge is to find a good balance between bias and variance, such that the resulting model exhibits sufficient complexity to reduce the error to a minimum while retaining low bias and variance.

9.2.2 Overcoming Underfitting and Overfitting

For underfitting models, we can reduce the bias by using a more complex model. For instance, for regression, we can use polynomial regression instead of linear regression. For decision trees, we can increase the depth of our trees.

For overfitting models, we can reduce the variance by selecting a model with fewer parameters (such as using a degree of 3 instead of 10 for polynomial regression), increasing the size of our training data, or reducing the noise in our data (by removing outliers and errors).

In addition, we can perform what is known as regularization. Regularization is a technique that involves constraining a model to make

it simpler.

For instance, the SVM model covered in Chapter 7 can be regularized using the `C` and `gamma` hyperparameters.

For basic intuition, you can think of `C` as the cost of misclassification. A high `C` value penalizes misclassification in the training set and typically results in a more complex model.

`gamma` controls the range of influence of a training instance and is only relevant for non-linear kernels. A high `gamma` value corresponds to a smaller range of control and results in a complex model.

Figure 9.2 below shows the classification boundaries of SVM models with different `C` and `gamma` values.

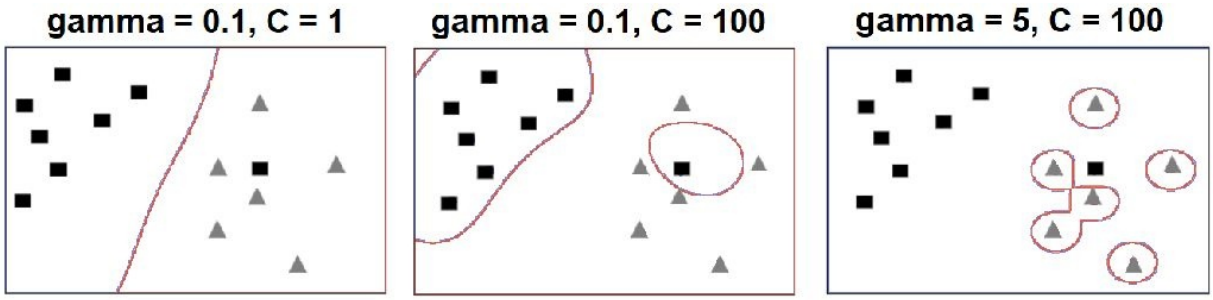

Figure 9.2: Effects of C and gamma

The first chart on the left uses a low `C` value that tolerates the misclassification of the rectangle (among the triangles) well and results in a simpler model.

In contrast, the last chart uses high `C` and `gamma` values that classify all the points correctly but result in a complex model that has overfitted the data and is unlikely to generalize well to new data.

To prevent such overfitting from happening, we can adjust the values of `C` and `gamma`. To do that, we perform what is known as hyperparameter tuning.

9.3 Hyperparameter Tuning

Hyperparameter tuning refers to the process of testing different hyperparameter values to find the best ones to use. There are two ways to do so.

One way is to fiddle with different hyperparameter values and evaluate their performances manually. This approach is time-consuming and cumbersome.

A better approach is to use the **`GridSearchCV`** class in sklearn to automate the process. To do that, we specify the hyperparameter values to test using a dictionary or a list of dictionaries. For each combination of hyperparameter values specified, the **`GridSearchCV`** class trains a model and evaluates it using cross-validation (hence the "CV" suffix in its name).

For instance, to test different values for the **`C`** and **`gamma`** hyperparameters, we can use the list below:

```
[{'C': [1, 2, 3], 'kernel': ['linear']},
{'C': [0.1, 0.2], 'gamma': [10, 20], 'kernel': ['rbf']}]
```

This list consists of two dictionaries. For the first dictionary, the possible combinations are

```
C = 1, kernel = 'linear'
C = 2, kernel = 'linear'
C = 3, kernel = 'linear'
```

For the second dictionary, the possible combinations are

```
C = 0.1, gamma = 10, kernel = 'rbf'
C = 0.1, gamma = 20, kernel = 'rbf'
C = 0.2, gamma = 10, kernel = 'rbf'
C = 0.2, gamma = 20, kernel = 'rbf'
```

This gives us a total of 7 combinations. If we use 5-fold cross-validation to evaluate these combinations, there will be a total of 7x5 = 35 rounds of training. The **`GridSearchCV`** class performs these rounds of training and returns the set of hyperparameter values with the best performance.

In this example, the number of training rounds is relatively small as we only have seven combinations to test. If we have more combinations and the number of training rounds becomes unmanageable (e.g., thousands of training rounds), it is more feasible to do a randomized search using the **`RandomizedSearchCV`** class.

As the name suggests, the **`RandomizedSearchCV`** class does not try all combinations of the hyperparameter values we specify. Instead, it randomly selects a fixed number of combinations and evaluates them. Other than that, the **`RandomizedSearchCV`** class is very similar to the **`GridSearchCV`** class.

In the next section, we'll demonstrate how to perform PCA on a dataset with 30 features, train a SVM model to classify the instances, and use the **`GridSearchCV`** class to find the best set of hyperparameter values.

9.4 Dimensionality Reduction and Hyperparameter Tuning with Scikit-Learn

The dataset that we'll be using is the breast cancer dataset available in sklearn. This is a well-known dataset in the machine learning community and can be downloaded using the **`load_breast_cancer()`** function in sklearn.

The dataset consists of two classes – **`malignant`** and **`benign`**, and each instance has 30 features.

Our goal is to reduce the dimensionality of the dataset and train a SVM model to classify the instances. Let's do that now.

```
import numpy as np
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

Here, we first import the necessary modules, including the **`sklearn.datasets`** module needed to download the breast cancer dataset. Next, we use the **`load_breast_cancer()`** function to download the dataset. This function returns a dictionary-like object known as a **`Bunch`** object, which we assign to a variable called **`cancer`**.

To get the features and target variable from **`cancer`**, we use its **`data`** and **`target`** attributes:

```
X = cancer.data
y = cancer.target
```

Here, we assign the features to **`X`** and the target variable to **`y`**, both of which are NumPy arrays.

To get the feature and class names of the dataset, we use the **`feature_names`** and **`target_names`** attributes, respectively:

```
print(cancer.feature_names)
print(cancer.target_names)
```

Part of the output for the statements above are shown below:

```
['mean radius' 'mean texture' ... 'worst fractal dimension']
['malignant' 'benign']
```

Now, let's examine the first elements in **`X`** and **`y`**:

```
print(X[0])
print(y[0])
```

Part of the output for these statements is shown below:

```
[1.799e+01 1.038e+01 ... 1.189e-01]
0
```

The output above tells us that for the first instance in the dataset, **`mean radius = 1.799e+01`**,
`mean texture = 1.038e+01`, and so on. In addition, the class for this instance is **`0`**, which corresponds to the **`'malignant'`** label.

After downloading the dataset, we are ready to apply PCA. Before we do that, let's split the dataset into training and test subsets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)
```

Now, we need to create two pipelines. The first pipeline does not include the dimensionality reduction step, while the second does. This allows us to compare the performance of the original dataset with the reduced dataset.

To create a pipeline without PCA, we use the code below:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

scale_step = ('scaler', StandardScaler())
SVM_step = ('SVM', SVC(random_state=0))

# Creating the pipeline without PCA
pipeline = Pipeline([scale_step, SVM_step])

# Training the Model
pipeline.fit(X_train, y_train)
```

This code should be self-explanatory. Let's evaluate the model using cross-validation:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(pipeline, X_train, y_train, cv=5,
scoring="accuracy")
print(scores.mean())
```

If you run the code above, you'll get **`0.9758241758241759`** as the output.

9.4.1 Dimensionality Reduction with PCA

Now, let's create another pipeline that includes dimensionality reduction using PCA:

```
from sklearn.decomposition import PCA
pca_step = ('pca', PCA(n_components = 5))
pca_pipeline = Pipeline([scale_step, pca_step, SVM_step])
pca_pipeline.fit(X_train, y_train)
```

To perform PCA in sklearn, we need to import the **`PCA`** class from the **`sklearn.decomposition`** module. We use the **`n_components`** parameter to specify the number of components or percentage of variance we wish to retain.

Here, we specify the number of components as 5. Alternatively, we can specify the percentage of variance. To do that, we assign a value between 0 and 1 to **`n_components`**. For instance, if we wish to retain 95% of the variance, we pass **`n_component = 0.95`** to the constructor.

After creating the pipeline, we assign it to a variable called **`pca_pipeline`** and use its **`fit()`** method to train our model. Let's evaluate the performance of the model now:

```
scores = cross_val_score(pca_pipeline, X_train, y_train, cv=5,
scoring="accuracy")
print(scores.mean())
```

This gives us **`0.9692307692307693`** as the output, which is only slightly lower than the accuracy we got on the full dataset. Therefore, even though we reduced the number of dimensions from 30 to 5, the

accuracy of our model is not severely affected.

We can check the amount of variance explained by each component in the reduced dataset. To do that, we need to access the **`PCA`** object in the **`pca_pipeline`** pipeline. We do that using the pipeline's **`named_steps`** attribute:

```
# Getting the PCA object in the pipeline
pca_obj = pca_pipeline.named_steps['pca']
```

Here, we use **`pca_pipeline.named_steps['pca']`** to access the **`PCA`** object in the pipeline and assign it to a variable called **`pca_obj`**.

A **`PCA`** object has an **`explained_variance_ratio_`** attribute that shows the ratio of variance explained by each component. To access this attribute, we write:

```
print(pca_obj.explained_variance_ratio_)
```

If you run the code above, you'll get the following output:

```
[0.43430767 0.19740115 0.09351771 0.06677661 0.05642452]
```

This output tells us that the first PCA component explains 43.4% of the variance in the feature set, the second explains 19.7%, and so on.

9.4.2 Hyperparameter Tuning

Next, let's move on to hyperparameter tuning using the **`GridSearchCV`** class:

```
from sklearn.model_selection import GridSearchCV

# Defining the parameters grid
params = [
{'pca__n_components':[5, 0.95], 'SVM__C': [0.1, 1, 2], 'SVM__kernel':
['linear', 'rbf']},
{'SVM__gamma': [0.1, 1, 2], 'SVM__kernel':['poly', 'rbf', 'sigmoid']}]
```

Here, we first import the **`GridSearchCV`** class from the **`sklearn.model_selection`** module. Next, we specify the parameters that we want to test using a list of dictionaries and assign it to a variable

called **params**.

Notice that the keys in **params** are **'pca__n_components'**,**'SVM__C'**, **'SVM__kernel'**, and **'SVM__gamma'**.

As we'll be using a pipeline (**pca_pipeline**) for our grid search, we need to prefix the name of the pipeline step to the hyperparameter's name, separating the two names with *double underscores*.

For instance, we prefix **pca** to **n_components** (**'pca__n_components'**) to denote that this element is for testing the **n_components** hyperparameter in the pipeline's **'pca'** step.

After defining the parameters to test, we are ready to do a grid search. To do that, we need to create a **GridSearchCV** object:

```
grid_search = GridSearchCV(pca_pipeline, params, cv = 3,
scoring='accuracy')
```

Here, we pass **pca_pipeline** and **params** to the **GridSearchCV()** constructor to instantiate an object called **grid_search**, specifying the number of folds for cross-validation as 3 and the scoring criterion as **'accuracy'**.

Now, let's call the object's **fit()** method to perform the search:

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

After the search is complete, we can get the performance and parameters of the best model using the **best_score_** and **best_params_** attributes, respectively.

In addition, we can get the best model using the **best_estimator_** attribute. This model incorporates the scaling, dimensionality reduction, and classification steps in the **pca_pipeline** pipeline, using the best hyperparameters obtained from the grid search.

```
print('Best Score = ', grid_search.best_score_)
print('Best Parameters = ', grid_search.best_params_)
final_model = grid_search.best_estimator_
```

If you run the code above, you'll get the following output:

```
Best Score =  0.9714767050075519
Best Parameters =  {'SVM__C': 1, 'SVM__kernel': 'linear',
'pca__n_components': 0.95}
```

This output shows that the best-performing model (which we assign to **`final_model`**) has an accuracy score of 0.971 (rounded off) when evaluated on the training set using a 3-fold cross-validation process.

Now, let's evaluate **`final_model`** on the test set to see if it generalizes well:

```
from sklearn.metrics import accuracy_score
y_pred = final_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
accuracy
```

If you run the code above, you'll get **`0.9736842105263158`** as the output, indicating that our model generalizes well to the test set.

That's it. We've covered most of the fundamental concepts in machine learning. With that, we are ready to work on the projects in the next three chapters. Have fun!

Project 1 - Regression

For this first project, we'll be building a regression model for a dataset stored in a file called *reviews.csv*.

The dataset stores information about mobile applications listed on a fictitious store and consists of 500 instances and seven columns. The columns are **`Category`**, **`No Of Reviews`**, **`No Of Installs`**, **`Size`**, **`Price`**, **`Days since Last Update`**, and **`Rating`**.

Our job is to build a regression model to predict the rating (**`Rating`**) of an application. This project will be very guided. The next two projects will be less guided to give you a chance to figure things out yourself.

Task 1 - Import the libraries and load the dataset

First, let's create a new notebook and name it *Project1.ipynb*.

Next, import NumPy, pandas, and matplotlib.pyplot using the **`np`**, **`pd`**, and **`plt`** aliases, respectively, and add the line **`%matplotlib inline`** to specify the backend for Matplotlib.

After importing, upload *reviews.csv* to Google Colab and load the file into a DataFrame called **`df`**. Finally, use the **`head()`** method to take a quick look at the DataFrame.

Expected Output

	Category	No Of Reviews	No Of Installs	Size	Price	Days since Last Update	Rating
0	games	510	1437	318.0	0.00	81	1.94
1	productivity	155	1547	204.0	-0.50	72	2.07
2	books	273	1162	271.0	0.81	69	1.34
3	games	110	1104	NaN	0.00	82	1.48
4	games	261	1403	224.0	1.01	98	2.80

Figure P1.1: The reviews dataset

Task 2 - Basic data exploration

Now, let's do some basic exploration. First, check if there are any missing values in **`df`**. Next, use the **`hist()`** method to plot a histogram for each numerical column in **`df`**. Finally, use the **`describe()`** method to generate some descriptive statistics for these columns.

Expected Output

```
Category                     0
No Of Reviews                0
No Of Installs               0
Size                        28
Price                        0
Days since Last Update       0
Rating                       0
dtype: int64
```

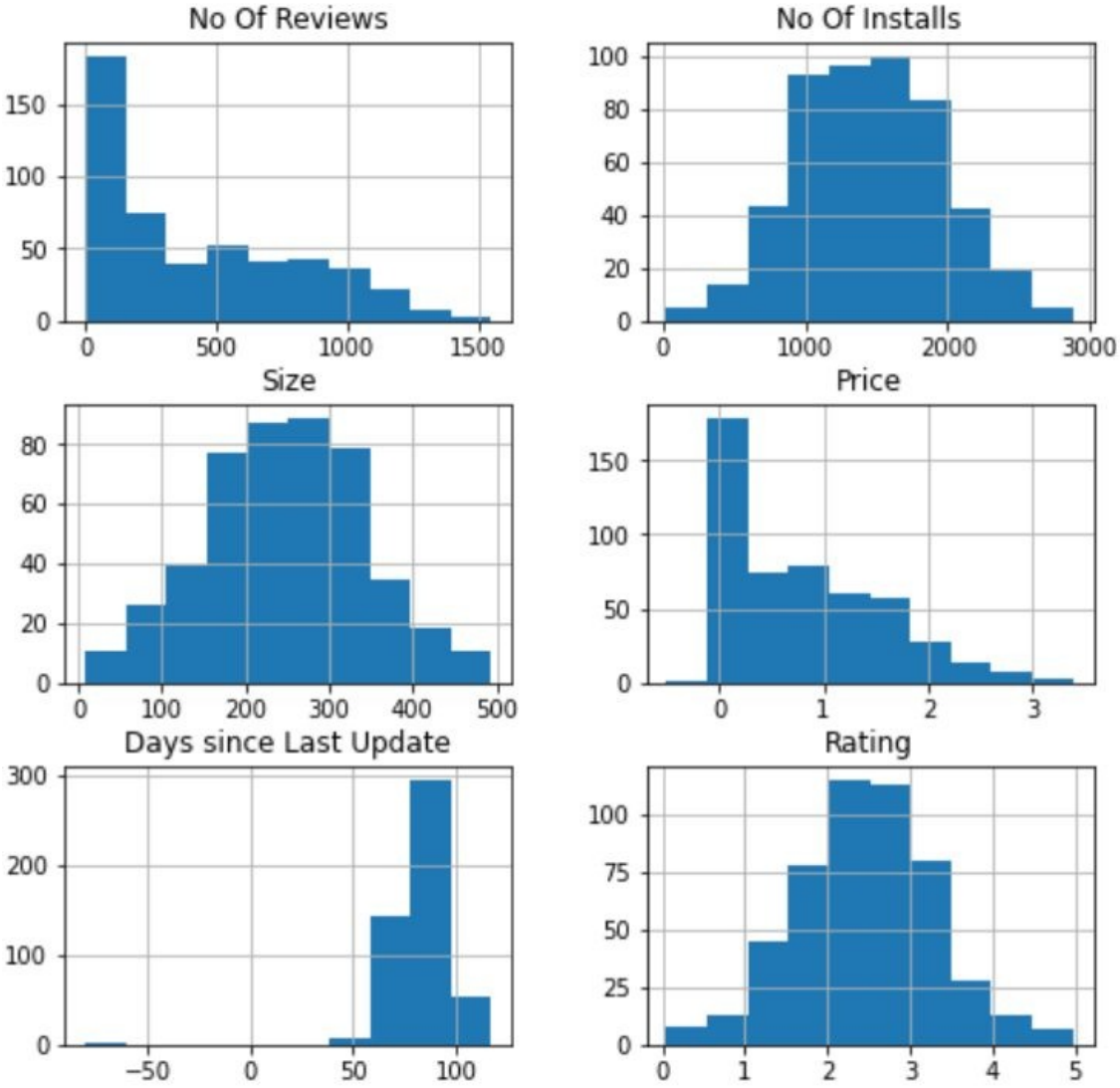

Figure P1.2: Missing Values and Histograms for the reviews dataset

	No Of Reviews	No Of Installs	Size	Price	Days since Last Update	Rating
count	500.000000	500.00000	472.000000	500.000000	500.000000	500.000000
mean	426.578000	1456.22200	247.870763	0.772200	83.274000	2.462260
std	366.600332	506.51664	95.105050	0.752114	14.960433	0.840859
min	0.000000	23.00000	8.000000	-0.500000	-80.000000	0.050000
25%	113.750000	1072.50000	184.000000	0.000000	76.000000	1.907500
50%	273.000000	1447.50000	249.000000	0.645000	84.000000	2.480000
75%	716.250000	1783.75000	312.250000	1.270000	92.000000	3.020000
max	1552.000000	2880.00000	494.000000	3.380000	117.000000	4.970000

Figure P1.3: Statistics for the reviews dataset

Task 3 - Check for invalid values

Study the output above carefully. Notice anything weird about the minimum values for the **`Price`** and **`Days since Last Update`** columns?

Both the **`hist()`** and **`describe()`** methods show that there are negative values for these two columns. Since prices and days cannot be negative, these values are likely to be errors.

Try using a boolean array to select all the rows with negative values for the **`Price`** column. Do the same for the **`Days since Last Update`** column. How many rows do you get in each case?

Hint: You can refer to Chapter 3, Section 3.8.2 for a discussion on using boolean arrays to select rows.

Answer

1 and 2, respectively

Task 4 - Remove invalid rows

Since there are only a total of 3 rows, let's drop them from our dataset. Use boolean arrays to select rows with *non-negative* values (i.e., values greater than or equal to 0) for both columns and assign the resulting DataFrame back to **`df`**. Verify that there are 497 rows left using the

`shape` attribute.

Expected Output

```
(497, 7)
```

Task 5 - Split data into training and test subsets

Now, we need to split our dataset into training and test subsets.

First, import the **`train_test_split()`** function from the **`sklearn.model_selection`** module.

Next, select all the features in **`df`** (i.e., all columns except **`Rating`**) as a *DataFrame* and assign it to **`X`**. In addition, select the **`Rating`** column as a *Series* and assign it to **`y`**.

Now, let's split **`X`** and **`y`** into training and test subsets using the **`train_test_split()`** function. Allocate 20% for the test set, use a **`random_state`** of **`0`**, and assign the resulting arrays to **`X_train`**, **`X_test`**, **`y_train`**, and **`y_test`**.

Task 6 - Check for correlation among features

Now, let's check if the features in our training subset are correlated with each other. Recall that when we discussed the **`corr()`** method in Chapter 3, Section 3.10, we mentioned that correlation among features could be a problem for some algorithms. Linear regression is one such algorithm.

Use the **`corr()`** method to check if the *features in our training set* are correlated. For this project, we do not want the correlation coefficients to be greater than 0.7 or smaller than –0.7.

Expected Output

	No Of Reviews	No Of Installs	Size	Price	Days since Last Update
No Of Reviews	1.000000	0.514130	-0.018121	-0.086976	-0.078577
No Of Installs	0.514130	1.000000	-0.004516	-0.018700	-0.054035
Size	-0.018121	-0.004516	1.000000	-0.038070	-0.002414
Price	-0.086976	-0.018700	-0.038070	1.000000	-0.031706
Days since Last Update	-0.078577	-0.054035	-0.002414	-0.031706	1.000000

Figure P1.4: Correlation coefficients among features

All correlation coefficients are within the acceptable range.

Task 7 - Plot scatter plots

Next, let's plot six scatter plots for the *training* subset. Each plot should show the **`Rating`** column (on the y-axis) against one of the other columns (e.g., **`Rating`** against **`Size`**).

Use a figure with 2 rows, 3 columns, and a figure size of (15, 10), and label the axes for each plot.

Expected Output

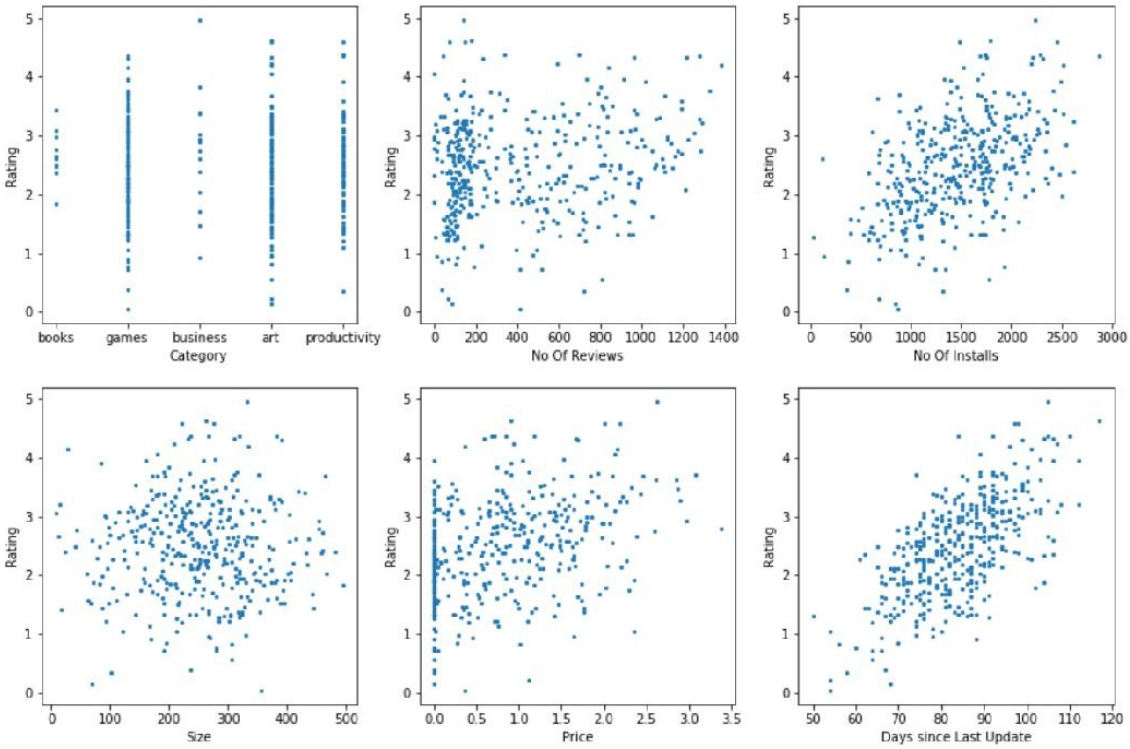

Figure P1.5: Scatter plots of the Rating column against other

columns

Task 8 - Data preprocessing

Now, we need to do some data preprocessing on **`X_train`**. We saw in Task 2 that the **`Size`** column has missing values. Therefore, we need to replace those missing values with the mean of that column.

In addition, we need to scale all the numerical columns in **`X_train`** and encode the **`Category`** column using one hot encoding.

To do the above, let's import the **`StandardScaler`** and **`OneHotEncoder`** classes from **`sklearn.preprocessing`**. In addition, we need to import the **`SimpleImputer`** class from **`sklearn.impute`**.

Next, declare a variable called **`num_col`** and assign the list **`['No Of Reviews', 'No Of Installs', 'Size', 'Price', 'Days since Last Update']`** to it. This list stores the names of all the numerical features.

Declare another variable called **`cat_col`** and assign **`['Category']`** to it.

Now, we are ready to replace the missing values in the **`Size`** column.

Instantiate a **`SimpleImputer`** object with **`strategy='mean'`** and assign it to **`imp`**. Next, use **`imp`** to call the **`fit_transform()`** method and pass **`X_train[num_col]`** to the method. Although only the **`Size`** column has missing values, it is all right to pass all the numerical columns to the **`fit_transform()`** method; columns with no missing values will not be affected. Assign the resulting array to **`tf_num`**.

Next, instantiate a **`StandardScaler`** object and assign it to **`scaler`**. Use **`scaler`** to call the **`fit_transform()`** method and pass **`tf_num`** to the method; assign the resulting array back to **`tf_num`**.

Finally, instantiate a **`OneHotEncoder`** object and assign it to **`ohe`**. We do not want the object to return a sparse matrix. In addition, we want it to drop the first category when encoding. Try instantiating the object

yourself, making sure you pass the correct arguments to the constructor.

Next, use **`ohe`** to call the **`fit_transform()`** method and pass **`X_train[cat_col]`** to the method. Assign the resulting array to **`tf_cat`**.

Last but not least, concatenate **`tf_num`** and **`tf_cat`** (which are both NumPy arrays) along axis 1 and assign the final array to **`X_train_transformed`**.

Verify that you have done the steps above correctly by printing the value of **`X_train_transformed[0]`**.

Expected Output

```
[-1.14922  1.1215984  -1.13561632  0.0732169  0.01224327  1.  0.  0.
0.]
```

Task 9 - Train our model

Now, we are ready to train our model.

To do that, import the **`LinearRegression`** class from **`sklearn.linear_model`** and instantiate a **`LinearRegression`** object called **`model`**.

Use **`model`** to train a linear regression model using **`X_train_transformed`** and **`y_train`**. After training, print the values of **`coef_`** and **`intercept_`**.

Expected Output

```
[-0.01263761  0.44527099  0.00512994  0.33109219  0.57813034  0.15265019
-0.09387284  0.01393827  0.09654561]
2.4437510828137157
```

Task 10 - Evaluate the model

Now, let's evaluate our model on the *training set*.

First, import the **`mean_squared_error()`** and **`r2_score()`** functions from **`sklearn.metrics`**.

Next, compute the RMSE and R^2 score metrics for the training set using these functions and print their values.

Hint: You need to get the predicted values for **`X_train_transformed`** first.

Expected Output

```
0.313351772053805
0.8579549928483164
```

Do the same with the test set.

Hint: You need to *transform* **`X_test`** and use the transformed dataset to get the predicted values for the test set.

Expected Output

```
0.2845552987500199
0.8958679954075467
```

After completing Task 10, the training and evaluation of our model is complete.

The following tasks involve using the **`ColumnTransformer`** and **`Pipeline`** classes to simplify our workflow. We'll get the same model. The main purpose is to illustrate how to use the two classes.

Task 11 - Use a column transformer and pipeline to simplify our workflow

First, let's import the **`ColumnTransformer`** and **`Pipeline`** classes from the **`sklearn.compose`** and **`sklearn.pipeline`** modules, respectively.

Next, instantiate a **`Pipeline`** object and assign it to a variable called **`num_preprocessing`**.

This pipeline should include two transformers. The first is for replacing missing values with the mean using a **`SimpleImputer`** object, and the second is for scaling the resulting features using a **`StandardScaler`**

object.

Next, instantiate a **`ColumnTransformer`** object and assign it to a variable called **`full_preprocessing`**.

A column transformer allows us to specify the columns for each transformation. Here, we want the column transformer to transform the columns in **`num_col`** using the **`num_preprocessing`** pipeline. In addition, we want it to transform the column in **`cat_col`** using a **`OneHotEncoding`** object (with **`sparse=False`** and **`drop="first"`**).

You can refer to Chapter 5, Section 5.3.2 for two examples of using a **`ColumnTransformer`** object. In that section, we used built-in transformers (such as a **`SimpleImputer`** object) for the transformations. Alternatively, we can use a pipeline, such as the **`num_preprocessing`** pipeline created above.

Try creating the **`ColumnTransformer`** object yourself and assign the resulting object to **`full_preprocessing`**. We do not need to specify the **`remainder`** parameter for this task as all the columns will be transformed by the transformer.

Finally, create one more pipeline called **`final_pipeline`**. This pipeline includes the **`full_preprocessing`** transformer for preprocessing the dataset and a **`LinearRegression`** object for training the model. Try creating this pipeline yourself.

Task 12 - Train the model using the pipeline

Now, we can use **`final_pipeline`** to train our model (using the **`fit()`** method). Try doing this yourself.

Task 13 - Evaluate the model

Finally, we can use **`final_pipeline`** to evaluate our model. Try evaluating the model on the training set using the RMSE and R^2 score metrics and print the resulting scores.

Expected Output

```
0.313351772053805
0.8579549928483164
```

Do the same for the test subset.

Expected Output

```
0.2845552987500199
0.8958679954075467
```

As expected, we get the same results as the model we trained previously without using a pipeline. However, using a pipeline (and a column transformer) simplifies the workflow substantially.

Task 14 - Do cross-validation using the pipeline

We mentioned in Chapter 6, Section 6.7 that one advantage of using a pipeline is that we can use it to do cross-validation. Let's do that now.

First, import the **`cross_val_score()`** function from **`sklearn.model_selection`**.

Next, use this function to do a 4-fold cross-validation on the training subset with **`"neg_root_mean_squared_error"`** as the scoring criterion. Repeat the same with **`"r2"`** as the criterion.

Finally, print the mean of the (positive) scores of each validation.

Hint: You can refer to the examples in Chapter 6, Section 6.7 for help on doing cross-validation.

Expected Output

```
0.32508192142610537
0.8409908098104693
```

Once Task 14 is done, Project 1 is complete. Congratulations!

Now, we are ready to move on to Project 2.

Project 2 - Classification

The dataset that we'll be using for this project consists of 1797 handwritten digits (from 0 to 9) and can be downloaded using the **`load_digits()`** function in **`sklearn.datasets`**.

Each instance is labeled and has 64 features with no missing values. For instance, the features of the first instance are given by the following array:

```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4., 12.,
0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8., 0.,  0.,  4.,
11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5., 10., 12.,  0.,  0.,
0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

Each element in this array represents a pixel of an 8x8 image. To display the array as an image, we need to reshape it to a (8, 8) 2D array. The first row of the resulting array is shown below:

```
[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.]
```

Each number in this array gives us the pixel's intensity and ranges from 0 (black) to 15 (white). For instance, the first number 0 tells us that the first pixel for this row is black, while the fourth number 13 tells us that the fourth pixel is almost white.

Our job is to build a model to predict the class of an instance. There are a total of 10 classes, one for each digit from 0 to 9.

Task 1 - Import the libraries and load the dataset

First, let's import NumPy, pandas, and pyplot using the **`np`**, **`pd`**, and **`plt`** aliases and import the **`load_digits()`** function from **`sklearn.datasets`**.

Next, use the **`load_digits()`** function to load the dataset into a variable called **`digits`**. Similar to the **`load_breast_cancer()`** function discussed in Chapter 9, this function gives us a **`Bunch`** object.

Try getting the features and target variable from **`digits`** and assign them to **`X`** and **`y`**, respectively.

Task 2 - Display the first instance as an image

Now, let's display the image of the first instance.

First, reshape **`X[0]`** to a (8, 8) array and assign the result to a variable called **`some_digit`**.

To display **`some_digit`** as an image, we need a built-in function in pyplot called **`imshow()`**. This function displays data as an image and accepts an array (e.g., **`some_digit`**) as the first argument. We can also pass **`cmap='gray'`** to the function to indicate that we want to display the array as a grayscale image.

Try displaying **`some_digit`** as an image yourself.

Expected Output

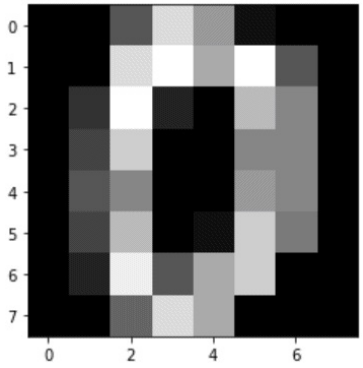

Figure P2.1: Displaying the first instance

Task 3 - Print the label of the first instance

What digit does the image above resemble? Print the label of the first instance to check if you got the correct digit.

Expected Output

```
0
```

Task 4 - Split data into training and test subsets

Next, let's split our dataset into training and test subsets using the **`train_test_split()`** function. Allocate 20% for the test set and use a **`random_state`** of `0`. Assign the resulting arrays to **`X_train`**, **`X_test`**, **`y_train`**, and **`y_test`**.

Task 5 - Create a pipeline to train our model

Now, let's create a pipeline to scale our features, perform PCA, and train a SVM model.

Use a **`StandardScaler`**, **`PCA(n_components = 0.95)`**, and **`SVC(random_state=0)`** estimator for the three steps and name the steps **`'scaler'`**, **`'pca'`**, and **`'svm'`**, respectively.

Assign this pipeline to a variable called **`pipeline`**.

Task 6 - Perform a grid search to find the best hyperparameters

After creating the pipeline, we can do a grid search to find the best hyperparameters for the **`SVC`** estimator.

First, declare a variable called **`params`** and assign a dictionary to it. This dictionary should allow us to test the **`C`** hyperparameter of the **`'svm'`** step, using the values 1, 5, 8, and 10. For each value of **`C`**, we want to try different kernels. The kernels to try are **`'linear'`**, **`'poly'`**, **`'rbf'`**, and **`'sigmoid'`**.

After declaring and initializing **`params`**, instantiate a **`GridSearchCV`** object and assign it to a variable called **`grid_search`**. This object should allow us to do a grid search using **`pipeline`** as the estimator and **`params`** as the dictionary of hyperparameters to test. In addition, it should enable us to use 4 folds for the cross-validation process and **`'accuracy'`** as the scoring criterion.

After instantiating the object, use it to perform a grid search on the training set and print the best score and parameters found by the search.

Expected Output

```
0.9860820953265242
{'svm__C': 8, 'svm__kernel': 'poly'}
```

Task 7 - Get the ratio of variance explained by each component

For the previous task, we used PCA to reduce the dimensionality of our dataset. Determine the ratio of variance explained by each component and print the results.

Hint: You need to use the **`named_steps`** attribute of the *best estimator returned by the grid search*. This best estimator is a pipeline. Refer to Chapter 9, Section 9.4.1 for an example of how we can use the **`named_steps`** attribute of a pipeline.

After getting the ratios, determine the number of components in the model.

Expected Output

```
[0.12164624 0.09634853 0.08578334 0.06457027 0.04897962 0.04183235
0.03929765 0.03282099 0.02979124 0.02809632 0.02741238 0.02602094
0.02304403 0.02207157 0.02049244 0.01784251 0.01735509 0.01662399
0.01624181 0.01510787 0.01347544 0.01294908 0.0117134  0.01062522
0.01039421 0.00941729 0.00932626 0.00840604 0.00827709 0.00789892
0.00749741 0.00715596 0.00665508 0.00641068 0.00594299 0.00567498
0.00514703 0.00473037 0.00454567 0.00420127]
```

Answer

40 components

Task 8 - Evaluate the best estimator on the test set

Last but not least, let's evaluate the performance of the best estimator on the test set. Import the **`accuracy_score()`** function from **`sklearn.metrics`** and use it to determine how well our model generalizes to the test set.

Print the value of the resulting accuracy score.

Expected Output

```
0.9833333333333333
```

This score is very similar to the score we got on the training set in Step 6. Hence, our model generalizes well to the test set.

With that, Project 2 is complete, and we are ready to move on to the last project.

Project 3 - Clustering

This last project uses the same dataset as Project 2 and is also concerned with classifying instances in the dataset. However, it uses a different technique to reduce the dimensionality of the data.

Task 1 - Import the libraries, load the dataset, and split it into training and test subsets

First, let's repeat what we did for tasks 1 and 4 in the previous project. We need to import the necessary libraries and modules, load the dataset, and split it into training and test subsets.

Task 2 - Elbow plot

Next, we need to train different **`KMeans(random_state = 0)`** models to cluster the instances in **`X_train`**.

Use one of the following **`n_clusters`** values - 10, 15, 20, 25, 30, 35, 40, 45, 50 – to train each model and plot the inertias as an elbow plot.

Hint: You can refer to Chapter 8, Section 8.3 for help.

Expected Output

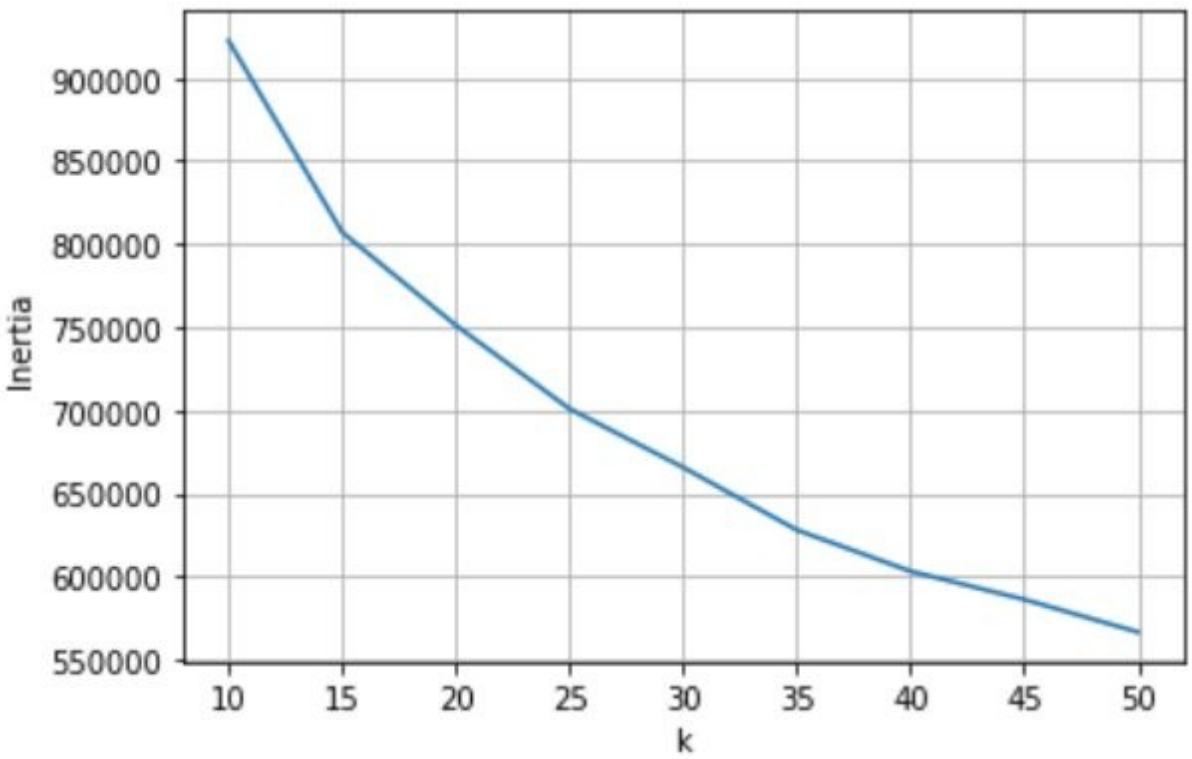

Figure P3.1: Elbow plot for the digits dataset

Task 3 - Select number of clusters

Select the number of clusters based on the elbow plot above. We'll use 15 for the tasks below.

Task 4 - Get the distances of the first instance from the centroids

Instantiate a **`KMeans`** object using **`random_state=0`** and **`n_clusters=15`** and assign it to a variable called **`kmeans`**.

Use **`kmeans`** to call the **`fit_transform()`** method, passing **`X_train`** as input to the method. Assign the resulting array to a variable called **`transformed`** and print the value of **`transformed[0]`**.

Expected Output

```
[40.27473174 48.89414703 47.03967829 52.34630829 55.2135072  43.87537153
43.99011546 44.39907444 54.96211714 56.20646651 27.03319993 52.79477229
50.82501222 49.00040622 43.86789918]
```

Task 5 - Verify the cluster of the first instance

Our k-means model has 15 centroids as there are 15 clusters. Each element in the array above gives us the distance of **`X_train[0]`** to one of the centroids.

The 11th element in this array is the smallest (27.03319993). Therefore, **`X_train[0]`** is assigned to the 11th cluster (i.e., Cluster 10). Verify this using the **`predict()`** method and print the result.

Expected Output

```
[10]
```

Task 6 - Create a pipeline to train our model

In Project 2, we used PCA to reduce the dimensionality of our dataset to 40 components. Instead of using PCA, we can use clustering for dimensionality reduction. For instance, we can use the distances calculated by our k-means algorithm as the reduced feature set to train our model.

The easiest way to use these distances as features is to create a pipeline. Recall that when we call the **`fit_transform()`** method of a pipeline, sklearn calls the **`fit_transform()`** method of all but the last estimator in the pipeline and passes the result of each estimator to the next.

Therefore, If we create a pipeline with a **`KMeans`** object as the first estimator, sklearn calls the **`fit_transform()`** method of that object to transform the original features into distances. These distances are then passed to subsequent estimators to train the model.

For this task, we'll create a pipeline with three estimators.

The first estimator is a **`KMeans(random_state=0, n_clusters=15)`** estimator, followed by a **`StandardScaler()`** estimator, and a **`SVC(random_state = 0)`** estimator. We'll name the steps **`'cluster'`**, **`'scaler'`**, and **`'svm'`**, respectively.

Try creating this pipeline yourself and assign it to a variable called

`pipeline`.

Task 7 - Perform a grid search to find the best hyperparameters

After creating the pipeline, we can use it to do a grid search. Repeat what you did for Task 6 in Project 2 for this task.

Expected Output

```
0.9763424636335499
{'svm__C': 8, 'svm__kernel': 'rbf'}
```

This output shows that even though we only used 15 features for this project, the best estimator performs almost as well as the best estimator in Project 2, which uses 40 features.

Task 8 - Evaluate the best estimator on the test set

Last but not least, let's evaluate the performance of our best estimator on the test set using the **`accuracy_score()`** function in **`sklearn.metrics`** and print the value of the resulting accuracy score.

Expected Output

```
0.9805555555555555
```

Here, we see that our model generalizes well to the test set. In fact, it performs slightly better on the test set than the training set.

With that, we've come to the end of this project and also the end of the book.

Congratulations! I sincerely hope you've found the book useful and have enjoyed the learning process.

You can find the suggested solutions for the three projects in Appendices A, B, and C. In addition, you can download the CSV files, completed notebooks, and images shown in this book at https://www.learncodingfast.com/machine-learning.

Thank you once again for your support!

Appendix A - Suggested Solution for Project 1

In the suggested solution below, code from different cells within the same task are separated by a line with two hyphens (--).

Task 1

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
df = pd.read_csv('reviews.csv')
df.head()
```

Task 2

```
df.isnull().sum()
--
df.hist(figsize=(10, 10))
--
df.describe()
```

Task 3

```
cond1 = df['Price'] < 0
df[cond1]
--
cond2 = df['Days since Last Update'] < 0
df[cond2]
```

Task 4

```
cond1 = df['Price'] >= 0
cond2 = df['Days since Last Update'] >= 0
df = df[cond1 & cond2]
df.shape
```

Task 5

```
from sklearn.model_selection import train_test_split
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.2, random_state = 0)
```

Task 6

```
X_train.corr()
```

Task 7

```
columns = ['Category', 'No Of Reviews', 'No Of Installs', 'Size',
'Price', 'Days since Last Update']

i = 0

fig, axs = plt.subplots(2, 3, figsize=(15, 10))

for row in range(0, 2):
  for col in range(0, 3):
    axs[row, col].scatter(X_train[columns[i]], y_train, s = 5)
    axs[row, col].set_xlabel(columns[i])
    axs[row, col].set_ylabel('Rating')
    i += 1
```

Task 8

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

num_col = ['No Of Reviews', 'No Of Installs', 'Size', 'Price', 'Days
since Last Update']
cat_col = ['Category']

imp = SimpleImputer(strategy='mean')
tf_num = imp.fit_transform(X_train[num_col])

scaler = StandardScaler()
tf_num = scaler.fit_transform(tf_num)

ohe = OneHotEncoder(sparse = False, drop = 'first')
tf_cat = ohe.fit_transform(X_train[cat_col])

X_train_transformed = np.concatenate((tf_num, tf_cat), axis=1)
print(X_train_transformed[0])
```

Task 9

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train_transformed, y_train)

print(model.coef_)
```

```
print(model.intercept_)
```

Task 10

```
from sklearn.metrics import mean_squared_error, r2_score

y_train_pred = model.predict(X_train_transformed)

rmse = mean_squared_error(y_train, y_train_pred, squared = False)
r2 = r2_score(y_train, y_train_pred)

print(rmse)
print(r2)
--
test_tf_num = imp.transform(X_test[num_col])
test_tf_num = scaler.transform(test_tf_num)
test_tf_cat = ohe.transform(X_test[cat_col])
X_test_transformed = np.concatenate((test_tf_num, test_tf_cat), axis=1)

y_test_pred = model.predict(X_test_transformed)

rmse = mean_squared_error(y_test, y_test_pred, squared = False)
r2 = r2_score(y_test, y_test_pred)

print(rmse)
print(r2)
```

Task 11

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

num_preprocessing = Pipeline(
    [('imp', SimpleImputer(strategy='mean')),
     ('scaler', StandardScaler())])

full_preprocessing = ColumnTransformer(
    [('num', num_preprocessing, num_col),
     ('cat', OneHotEncoder(sparse=False, drop='first'), cat_col)])

final_pipeline = Pipeline(
    [('pre', full_preprocessing),
     ('model', LinearRegression())])
```

Task 12

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

Task 13

```
y_train_pred = final_pipeline.predict(X_train)

rmse = mean_squared_error(y_train, y_train_pred, squared=False)
r2 = r2_score(y_train, y_train_pred)

print(rmse)
print(r2)
--
y_test_pred = final_pipeline.predict(X_test)

rmse = mean_squared_error(y_test, y_test_pred, squared=False)
r2 = r2_score(y_test, y_test_pred)

print(rmse)
print(r2)
```

Task 14

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(final_pipeline, X_train, y_train, cv=4,
scoring="neg_root_mean_squared_error")
neg_rmse_cv = scores.mean()
print(-neg_rmse_cv)

scores = cross_val_score(final_pipeline, X_train, y_train, cv=4,
scoring="r2")
r2_cv = scores.mean()
print(r2_cv)
```

Appendix B - Suggested Solution for Project 2

In the suggested solution below, code from different cells within the same task are separated by a line with two hyphens (--).

Task 1

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

from sklearn.datasets import load_digits

digits = load_digits()

X = digits.data
y = digits.target
```

Task 2

```
some_digit = X[0].reshape((8, 8))
plt.imshow(some_digit, cmap='gray')
```

Task 3

```
print(y[0])
```

Task 4

```
from sklearn.model_selection import train_test_split

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

Task 5

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import SVC

pipeline = Pipeline(
    [('scaler', StandardScaler()),
     ('pca', PCA(n_components = 0.95)),
     ('svm', SVC(random_state = 0))])
```

Task 6

```
from sklearn.model_selection import GridSearchCV

params = {'svm__C':[1, 5, 8, 10], 'svm__kernel': ['linear', 'poly',
'rbf', 'sigmoid']}

grid_search = GridSearchCV(pipeline, params, cv = 4, scoring =
'accuracy')
grid_search.fit(X_train, y_train)

print(grid_search.best_score_)
print(grid_search.best_params_)
```

Task 7

```
best_model = grid_search.best_estimator_
print(best_model.named_steps['pca'].explained_variance_ratio_)
--
print(best_model.named_steps['pca'].n_components_)
```

Task 8

```
from sklearn.metrics import accuracy_score

y_test_pred = best_model.predict(X_test)
print(accuracy_score(y_test, y_test_pred))
```

Appendix C - Suggested Solution for Project 3

In the suggested solution below, code from different cells within the same task are separated by a line with two hyphens (--).

Task 1

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

from sklearn.datasets import load_digits

digits = load_digits()

X = digits.data
y = digits.target
--
from sklearn.model_selection import train_test_split

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

Task 2

```
from sklearn.cluster import KMeans
inertias = []

for k in range(10, 51, 5):
  kmeans = KMeans(random_state = 0, n_clusters = k)
  kmeans.fit(X_train)
  inertias.append(kmeans.inertia_)

plt.plot(range(10, 51, 5), inertias)
plt.xlabel('k')
plt.ylabel('Inertia')
plt.grid()
```

Task 4

```
kmeans = KMeans(random_state=0, n_clusters=15)
transformed = kmeans.fit_transform(X_train)
print(transformed[0])
```

Task 5

```
X_train0_2D = X_train[0].reshape((1, -1))
print(kmeans.predict(X_train0_2D))
```

Task 6

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

pipeline = Pipeline(
    [('cluster', KMeans(random_state=0, n_clusters=15)),
     ('scaler', StandardScaler()),
     ('svm', SVC(random_state=0))])
```

Task 7

```
from sklearn.model_selection import GridSearchCV

params = {'svm__C':[1, 5, 8, 10], 'svm__kernel': ['linear', 'poly',
'rbf', 'sigmoid']}

grid_search = GridSearchCV(pipeline, params, cv=4, scoring = 'accuracy')
grid_search.fit(X_train, y_train)

print(grid_search.best_score_)
print(grid_search.best_params_)
```

Task 8

```
from sklearn.metrics import accuracy_score

best_model = grid_search.best_estimator_
y_test_pred = best_model.predict(X_test)

print(accuracy_score(y_test, y_test_pred))
```