AI PUBLISHING

Statistics for Beginners in Data Science

Theory and Applications of Essential Statistics Concepts using Python

Our Books are designed to teach beginners Data Science and AI

Edited by AI Publishing
EBook Converted and Cover by Gazler Studio
Published by AI Publishing LLC

ISBN-13: 978-1-7347901-1-5

How to Contact Us

If you have any feedback, please let us know by sending an email to contact@aispublishing.net.

Your feedback is immensely valued, and we look forward to hearing from you. It will be beneficial for us to improve the quality of our books.

To get the Python codes and materials used in this book, please click the link below:

https://www.aispublishing.net/book-statistics-ds

About the Publisher

At AI Publishing Company, we have established an international learning platform specifically for young students, beginners, small enterprises, startups, and managers who are new to data sciences and artificial intelligence.

Through our interactive, coherent, and practical books and courses, we help beginners learn skills that are crucial to developing AI and data science projects.

Our courses and books range from basic introduction courses to language programming and data sciences to advanced courses for machine learning, deep learning, computer vision, big data, and much more, using programming languages like Python, R, and some data science and AI software.

AI Publishing's core focus is to enable our learners to create and try proactive solutions for digital problems by leveraging the power of AI and data sciences to the maximum extent.

Moreover, we offer specialized assistance in the form of our free online content and eBooks, providing up-to-date and useful insight into AI practices and data science subjects, along with eliminating the doubts and misconceptions about AI and programming.

Download the Color Images

We request you to download the PDF file containing the color images of the screenshots/diagrams used in this book here:

https://www.aispublishing.net/book-statistics-ds

Get in Touch with Us

Table of Contents

Preface

§ Book Approach

This book will give you the chance to have a fundamental understanding of statistical analysis, which is needed for anyone who wants to enter the data science and analysis field. To achieve this, the book not only contains an in-depth theoretical and analytical explanation of all concepts but also includes dozens of hands-on, real-life projects that will help you understand the concepts better.

We will start with a light introduction about what statistics is and how it works in practice. Then, we will start exploring Python programming as all the projects are developed using it, and it is currently the most used programming language in the world. We will also explore the most-used libraries for data science, such as NumPy, Pandas, and Statsmodel.

Following that, we will discuss data exploration and analysis by explaining how to structure the datasets and summarize them properly. After that, we will focus on data visualization for a while as we discuss different types of graphs and when to use what.

Given all this, we will then discuss how to work with two or more variables and categorical data. After that, we will see how to perform statistical tests such as P-value, ANOVA, Chi-square, and Fisher's exact test. This will be followed by a confidence interval as we are going to spend some time on it.

Finally, we will end the book with two chapters on regression analysis and classification analysis.

§ Who Is This Book For?

This book is for anyone interested in statistics who wants to know how they are applied, without spending too much time in understanding and studying mathematics.

It is for anyone interested in starting to work on real-world projects with real-world data. This book focuses more on explaining the concepts in simple language rather than going through complex mathematical equations. This doesn't mean that mathematics will be completely ignored. But instead, it will also be explained, in simple terms, when necessary.

§ How to Use This Book?

To get the utmost benefit from this book, you need to read every single part very carefully. So, try your best not to skip any part. Also, you will find that I refer to additional materials sometimes. Make use of them as they will enhance your skills and understanding even more. You will better understand how to search for advanced topics after finishing this book.

Also, you will notice that there are a lot of hands-on projects in this book. Run them yourself, and also try other approaches that you might find in the additional materials.

About the Author

This book was developed by Ahmed Wael, who is pursuing his career in communication and information engineering, with a concentration in machine learning and big data. He is working in the field of AI, ranging from image processing and computer vision, deep learning and neural networks, natural language processing, data visualization, and many more. He is also a graduate of the Machine Learning Nano Degree at Udacity, where he is currently a mentor, tutoring over 100 students from around the world in the fundamentals of machine learning and deep learning.

If you have any queries regarding the eBook or just want to connect, feel free to reach him on GitHub or LinkedIn.

1

Introduction to Statistics

1.1. What Is Statistics?

Statistics is an interdisciplinary field concerned with developing and studying techniques to collect, analyze, and present data. There are two main ideas in this field, which are variation and uncertainty. To address these two ideas, statistics uses probability as it is the mathematical language used to describe uncertain events and explain their variation.

There are two branches of statistics, which are descriptive statistics and inferential statistics. Descriptive statistics is concerned with summarizing the data from a sample using metrics such as the mean and standard deviation, while inferential statistics is concerned with drawing conclusions from data that are subject to random variations such as sampling variation and observational errors.

There are a lot more details to discuss statistics, which will be done throughout the book.

1.2. How to Design a Study?

The first step in any data science and statistics project is to design a study. There are six main steps to do so, which are the following:

1. Identifying the population of study and the variables of interest.
2. Developing a detailed plan for data collection that is representative of the population.
3. Starting the data collection process.
4. Describing the data using descriptive statistics such as mean and variance.
5. Interpreting the data using inferential statistics.
6. Identifying any errors or limitations.

This, in a nutshell, is how to design a solid study.

1.3. How to Collect Data?

For data collection, there are five main steps that you should follow, which are as follows:

1. Determining what information to collect.
2. Setting a timeframe for the whole process.
3. Determining the best way to collect the data. We will discuss them in detail after defining these steps.
4. Collecting the data and storing them in a database management platform such as an SQL server.
5. Analyzing the data and making decisions.

Now, let us discuss the different methods of collecting data.

1. **Surveys**: This is done by directly asking the target

population for information either in-person, via phone, or online.

2. **Online Tracking:** You can also collect data through online information that users leave without knowing. In fact, any website can store at least 40 data points from each visitor.
3. **Online Marketing Analytics:** This can be achieved by launching marketing campaigns and keeping track of the customers' activity.
4. **Social Media:** You can collect a lot of valuable information from any social media website such as Facebook or Twitter, as users are voluntarily giving you their opinions and interests.

1.4. How to Describe Data?

In chapter 3, we are going to discuss in detail how to describe any type of data that you might encounter. But, for now, let us have a glimpse of this important topic.

There are a lot of ways to describe data, but in a nutshell, we mostly focus on the quality and quantity of the data. That is, we focus on the condition of the data and how much data we have. Of course, there are some trade-offs that we need to keep in mind.

For example, having larger datasets can lead to better results, but at the expense of a longer time to collect them. Also, the value types are another thing to be aware of, as data can come in different formats such as numeric, Boolean (true/false), and categorical (string). Being aware of that can help you in the later stages of the process, as each data type requires special treatment. Finally, another note is to keep track of the coding schemes. For example, one sub-set of your dataset can represent male and female as M and F, while another sub-set can represent them as 1 and 2.

1.5. How to Analyze Data?

Data analysis is the hardest part of any data science project, as it is a combination of science and art. However, there are some good practices that we highlight for now, and we are going to discuss everything in detail as we go.

First, we need to know that data can be split into two major categories, which are Numerical data (quantitative data) and Categorical data (qualitative data).

We can say that the data are categorical if the different values that the data can have cannot be used in mathematical operations. Categorical data can be split even more into ordinal (ordered) data and nominal (unordered) data. The rating of a movie is a good example of ordinal categorical data, while blood type is a good example of nominal data.

On the other hand, numerical data can be used in mathematical operations. Numerical data can be split even more into discrete numerical data and continuous numerical data. Discrete numerical data can only have a pre-defined set of values. Examples of that can be the number of bedrooms in a house. Continuous data can have any value from negative infinity to infinity. Examples of that can be the speed of a car. But of course, depending on the nature of the variable in the data, even the continuous variables should be restricted by a range.

For analyzing quantitative data, we can highlight these three techniques:

1. **Regression Analysis:** They are basically tools that can help us make predictions and forecast future values and trends. They simply measure the relationship between independent variables, which are the data that we can use to predict future values, and a dependent variable, which is the one that we want to predict.
2. **Hypothesis Testing:** Using this technique, we can compare the data against the assumptions and hypotheses we made about anything. It can also be used for forecasting, as we will see. There will be an entire section in chapter 5 about this topic. Also, note that it is called *T-value testing*.
3. **Monte Carlo simulation:** This is a method that uses probability theories to calculate the effect of unpredictable variables on a specific factor, which is crucial for predicting risk and uncertainty. To achieve that, it uses random numbers and data to *simulate* a pool of possible outcomes to any scenario based on any results.

Now, for analyzing qualitative data, we can highlight two main techniques:

1. **Content Analysis:** This technique basically highlights interesting trends in the answers. This can be very useful when working on things like open-ended surveys.
2. **Narrative Analysis:** This technique focuses on the way stories and ideas are communicated in a corporate. This can include how operational processes are viewed, how employees feel about their jobs, and how customers perceive an organization.

In the upcoming chapters, we will discuss data analysis in detail.

1.6. How to Make Conclusions?

Finally, we incorporate everything we mentioned to draw one or more conclusions about the data. This can be done using data visualization, statistical tests, or even along the data collection process.

2

Getting Familiar with Python

This chapter will give you a sound understanding about Python programming language in general, and more specifically, how to utilize it for statistical learning. We will start by assuming that you do not know anything about python programming, so we will explore all the basics together as this is a fundamental part of understanding statistical concepts after that. After understanding all the basics of Python, we will introduce how Python can be used for statistical analysis, in terms of the tools and the libraries that Python offers, which enable us to perform powerful analysis in a few lines of code.

2.1. The Python Language

Python is currently the most popular programming language in the world, according to PYPL, with a share of 29.71 percent. It seems that this dominance will continue in the future also, as the trend is +4.1 percent, which indicates that more people, either programmers or people starting their programming career, will use Python in the future.

Python is basically a general-purpose, high-level, interpreted programming language that was developed by Guido Van Rossum back in 1991.

General purpose means that it can be used for a wide spectrum of applications, such as data science, game development, web development, and many more.

High-level means that python is more humanly readable and generic for any computer architecture. This is opposed to the low-level programming languages, such as C and C++, which are more difficult for humans to interpret but much easier for the computer to do so. This makes executing the code much faster on a low-level programming language. Therefore, low-level programming languages are used more in very specific applications where the time of execution is a crucial metric to evaluate the performance of the program. However, if you do not care whether the code took 0.0001s or 0.0002s to run, then a high-level programming language, such as Python, is the way to go.

Finally, Python is an interpreted language, which means that the code you execute is executed directly without compiling it first into machine language instructions. While, on the other hand, in compiled languages, such as C, the code is first converted into machine language and then gets executed. This makes the interpreted language slower to run than the compiled language. However, as we have just said, if you do not care about the fractions of a second speedup that compiled and low-level languages offer, then you can start working with Python without hesitation.

Python has gone through many modifications and enhancements since it was first introduced nearly 28 years ago. Currently, there are two main stable versions of it, which are 2.7 and 3.6. In this book, we will be using 3.6 as it has more features and capability, and because Python 2.7 will be deprecated very soon, which means that there will be no support for it in the future.

Now, before we dive into the syntax of Python, we need to install it. We can do so using one of the following three methods:

1. Official Python Website: This is very easy to follow, but it will install only Python with no external libraries. Thus, **this method is not recommended**.
2. Miniconda: This will install the conda package manager along with Python. This method suffers from the same disadvantage of the first method, as all the external libraries must be installed manually.
3. Anaconda Distribution: This will install all the packages that you will need in many chapters of this eBook. Also, the installation of any additional packages is very easy and straightforward, and we will mention it when we need it. This is **the recommended method.**

Further Readings – Anaconda

If you want to know more details on how to use Anaconda, check this link:

https://docs.conda.io/projects/conda/en/latest/index.html. Additionally, for quick reference, you could check the following cheat sheet:

https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf on their site.

Hands-on Time – Using Python in Anaconda

Throughout the following sections, keep Python Jupyter notebook open, and execute all the examples.

2.2. Python Syntax

Given that you installed Python successfully, let us start exploring it. We will start with the language syntax, which basically means the grammar or the rules of the language, just like any spoken language such as French or English. Also, like any spoken language, every programming language has its own syntax, which can vary either a lot or very little.

We have five main rules in Python, which can be listed as follows:

1. **Line Structure:** This means that any Python code is divided into logical lines, and every line is ended by a token called a *newline*. While you do not write this token yourself, as it is embedded in the language, it is very important to know that it exists so you can understand how to write a syntax error-free code. This also means that a single logical line can be composed of one or more physical lines. Moreover, if a line has only comments or is just blank, it will be ignored by the interpreter.
2. **Comments:** They are very important in the documentation, as you will need this when you are working on a big project or want to share your code with someone else. To tell Python that this part is a comment, we use a hash *(#)* character.
3. **Joining lines:** This is usually needed when you are

writing a long logical line of code, and you want it to be all visible on the screen. We can do so using the backslash (\) character.

4. **Multiple statements on a single line:** If you want to write two separate logical lines into one line, you can do so using the semicolon (;) character.
5. **Indentation:** This is the most important rule, because, as your code gets more complex, you will need to define nested blocks of code. Thus, we will need a way to tell Python the flow of your nested blocks. In other languages, like Java or C++, curly brackets*{}* are used, while in Python, we use tabs to do the trick. All the statements within the same block should have the same indentation level.

Now, let us understand all Python data structures, both basic and advanced ones. To start with, we need to know that any code that we write is saved in the memory. In order to use a specific part of the code where we add two numbers, for example, we need to store the result with a name that both you and Python agree on, which is called *variable*. Of course, you can ignore that. But then, you will need to know all the memory locations for all the results and the data in your code, which is impractical. Let us see an example.

In the following snippet of code, we are adding two numbers together, but can you tell me where the result will be stored. In the memory, of course, but where? Can you tell me how we can access this data again for any modification? It is extremely difficult to do so.

```
1+2
```

On the other hand, with a very slight modification, we can do the following.

```
x= 1+2
```

Now, the result is stored in a variable called *x,* which we can refer to from now on throughout our code.

Given that we now understand what is meant by a variable, let us talk about the basic data types. The first category of data types is **number**, which can take three different formats, which are **integer, float, and complex**. We will focus only on integer and float, as complex variables are not used in regression analysis. In the ensuing code snippet, you can see yourself how we can define these types.

```
x = 3
y = 3.5
print(type(x))
print(type(y))
```

From the code snippet, we can see that you can print the type of any variable using *type* built-in function.

Moreover, you can do any mathematical operation between numerical variables as follows.

```
x = 1
y = 2.5
z = x*y
print(type(z))
<class ‘float’>
```

As you can see, the result is automatically saved as a float variable because we are multiplying an integer with a float.

Moving forward, let us discuss the second category of data types, which is a **string**. Strings are just sequences of character data, and we can use either double quotes or single ones to define a string variable as follows.

```
s = «Hello World! «
print(s)
print(type(s))
Hello World!
<class ‘str’>
```

We can access specific elements of “Hello World” using indexing as follows:

```
new_s = s[2:5]
print(new_s)
llo
```

We can also concatenate different strings together.

```
newer_s = s + new_s
print(newer_s)
Hello World! llo
```

Moreover, we can multiply a string by a number. This will make the string get repeated several times equal to the number.

```
repeated_s = s*4
print(repeated_s)
Hello World! Hello World! Hello World! Hello World!
```

However, we cannot add a number to a string directly.

```
Add_s_sum = s+4
----------------------------------------------------------
TypeError                    Traceback (most recent call last)
<ipython-input-8-c46ee2a4b7fb> in <module>----> 1 add_s_num
= s +4
TypeError: can only concatenate str (not “int”) to str
```

To resolve this error, you need to convert the number first to a string, and then you can add them together.

```
add_s_sum = s + str(4)
print(add_s_num)
Hello World! 4
```

We can perform this operation, called type casting, to any variable.

```
f_to_i = int(3.2)
i_to_f = float(3)
print(f_to_i)
print(type(f_to_i))
print(i_to_f)
print(type(i_to_f))
3
<class ‘int’>
3.0
<class ‘float’>
```

Now, let us move to the next data type, called **Boolean**. It is a data type that gets created to be used in comparisons and conditions, as the only values in it are True and False.

```
is_true = 1.2 > 1
print(is_true)
print(type(is_true)
True
<class ‘bool’>
```

Congratulations! You now know all about the basic data types in Python. Now, let us move to complex data types.

We will start with **lists**, which can be defined as a container of variables, of any type, stored together.

```
l = [1,2,5.1,’hi’]
print(l)
print(type(l))
[1,2,5.1,’hi’]
<class ‘list’>
```

As you can see, to define a list, we use square brackets, and the indexing starts from zero. We can access any number of consecutive elements as follows.

```
new_list = l[0:2]
print(new_list)
[1, 2]
```

We can also use negative indexing to access the list from the end instead of from the beginning.

```
print(l[-1])
hi
```

Moreover, we can add or concatenate two lists together, and we can remove values from a list.

```
l_1 = [4,15,7]
l_2 = l_1 + l
print(l_2)
[4, 15, 7, 1, 4, 5.1, ‘hi’]
```

```
l_2.append(8)
print(l_2)
[4, 15, 7, 1, 4, 5.1, ‘hi’, 8]
```

```
l_2.remove(8)
print(l_2)
[4, 15, 7, 1, 4, 5.1, ‘hi’]
```

That is all you need to know about lists for now.

The second complex data type is called **Tuple**, which is a special case of the list where the elements cannot be changed.

As you have seen, lists can be altered, which means they are mutable, while on the other hand, tuples are immutable.

The following code snippets show that to define a tuple, you use the same syntax of the list but with a circular bracket instead of the square ones.

```
t = (1,2,5.1,’hi’)
print(t)
print(type(t))
(1, 2, 5.1, ‘hi’)
<class ‘tuple’>
```

However, to index a value in a tuple, we use the same exact syntax of the list.

```
print(t[1])
2
```

Let us see what takes place when we try to change the value of some index.

```
t[1] = 10
-----------------------------------------------------------
TypeError                    Traceback (most recent call last)
<ipython-input-8-c46ee2a4b7fb> in <module>----> 1 t[1] = 10
TypeError:’tuple’ object does not support item assignment
```

For the next data type, we have a **Dictionary**, which is, just like its name, an address book, where you can find the address of anyone by just knowing his/her name. However, this name needs to be **unique**, as otherwise, you will not be entirely sure if this the correct address or not. The person’s name is called the **key**, while the address is called the **value**. The address is not unique, as many people can have the same address.

Take the heights of people as another example. Suppose we have John and Mary to whom we want to assign heights. Both can have the same heights, but the opposite is not true, as we cannot say that John is 180 cm and then say that he is 170 cm.

Moreover, if the height is the key, then we cannot assign the same height to different people.

To define a dictionary, we use curly brackets, and we use a colon to connect keys to values. Please also notice that while tuples and lists are ordered, dictionaries are not ordered, and we use the keys to index the values.

```
dic = { «john» : 170, « Mary» : 170}
print(dic)
print(type(dic))
{'john': 170, ' Mary': 170}
<class 'dict'>
```

Ok, so what happens when we assign two values to the same key?

```
dic_2 = {170 : «john», 170 : «Mary»}
print(dic_2)
{170: 'Mary'}
```

As you can see, the first value was overwritten by the second one.

Finally, let us talk about **sets**, which are complex data types that can only have a unique value. Just like the dictionary, sets are defined by using curly brackets, do not have an order, and cannot be indexed.

```
s = {1,20,4,5,6,1,2,3,4,5,3}
print(s)
print(type(s))
{1, 2, 3, 4, 5, 6, 20}
<class 'set'>
```

```
s[1]
-----------------------------------------------------------
TypeError                          Traceback (most recent call last)
<ipython-input-8-c46ee2a4b7fb> in <module>----> 1 s[1]
TypeError:'set' object does not support indexing
```

That's all you need to know about Python data structure.

But the question now is, when should you use what? To answer this question, you can follow this list of use cases:

- **Lists:** The most generic data type. Use it when your data do not have any special cases, and you want to use indexing.
- **Tuples:** It is mainly used when you know that the data should not be changed, no matter what.
- **Dictionaries:** Used when we want to have some sort of relation between some unique variables and other non-unique variables.
- **Sets:** Used when we know that any repeated data will be redundant.

Further Readings

If you want to know more about Python data structures, you can go here:

https://www.tutorialspoint.com/python/python_variable_types.htm

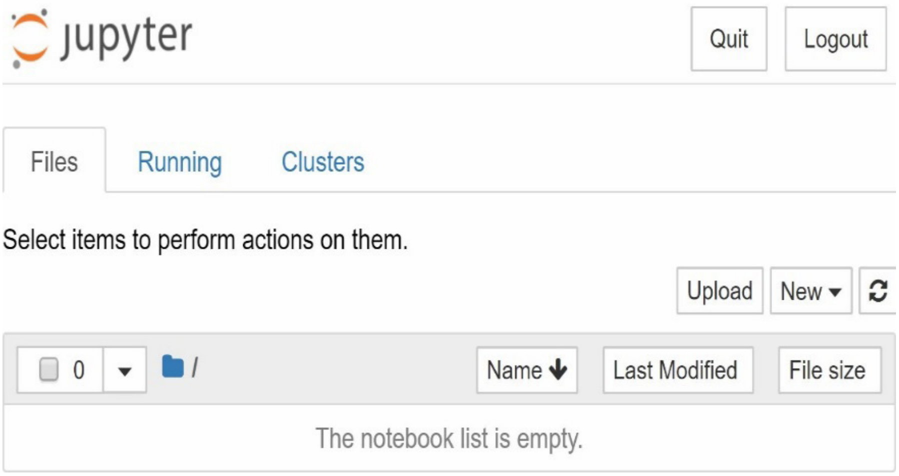
2.3. Jupyter Notebook

Now, let us explore the tools and the libraries that Python has regarding data science.

The first one of them is the Jupyter notebook, which is one of the fundamental tools that any data analyst uses right now. It is a web application that you can utilize to create and share documents containing code, text, equations, and visualizations.

If you installed Python using the third method (Anaconda Distribution) explained earlier in this chapter, you should have already installed Jupyter Notebook.

When you open the application, you will see this User Interface.

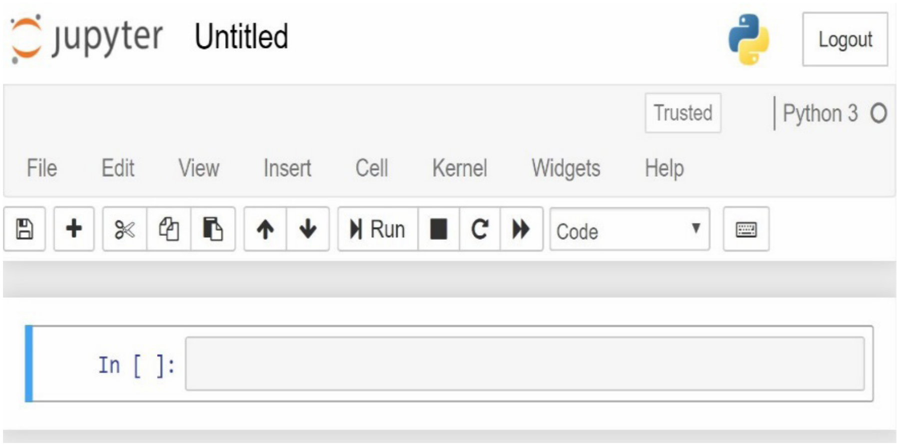


Select items to perform actions on them.

The notebook list is empty.

From here, you can go anywhere, create notebooks, upload files, and do much more.

To start, click on *New* from the right corner. You will then see a new notebook created which looks like this.



The most important part here is that you can either write in a code cell or a markdown cell. The markdown cell is to beautify and make the code more documented and clearer, but it is ignored by the Python interpreter.

Do not worry about all the other tabs, for now, as we will be using Jupyter notebooks heavily in all our exercises.

2.4. Using NumPy

NumPy is short for Numerical Python, which is a library consisting of multi-dimensional array objects, as well as, a collection of routines for processing those arrays. Its main use is for mathematical and logical operations on arrays.

NumPy is also installed with Anaconda distribution.

To understand and practice the capabilities of NumPy, let us start writing some code using it.

We can import NumPy using "import" and we usually use a short name for our libraries as we will be calling them too many times.

```
importnumpy as np
```

Create an array using NumPy by doing the following.

```
a=np.array([1,2,3])
print(a)
print(type(a))
[1 2 3]
<class ‘numpy.ndarray’>
```

Now, let us see how to get the shape of an array. This is crucially important in data science, as we will always work with arrays and matrices.

```
a.shape
(3,)
```

Let us create a multi-dimensional array.

```
a_2d=np.array([[1,2,3],[4,5,6]])
print(a_2d)
print(type(a_2d))
print(a_2d.shape)
[[1 2 3]
 [4 5 6]]
<class ‘numpy.ndarray’>
(2, 3)
```

Finally, let us see how to perform basic operations using NumPy.

```
a1=np.array([1,2,3])
a2=np.array([4,5,6])
print(a1+a2)
print(a1-a2)
print(a1*a2)
print(a1/a2)
print(np.sqrt(a1))
print(np.mean(a1))
print(np.median(a1))
print(np.max(a1))
print(np.min(a1))
[5 7 9]
[-3 -3 -3]
[4 10 18]
[0.25 0.4  0.5]
[1.         1.41421356 1.73205081]
2.0
2.0
3
1
```

Further Readings

To know more about NumPy, you can go here:

https://www.numpy.org/devdocs/user/quickstart.html

2.5. Using Pandas

Pandas is another very critical library in data science. It provides high-performance data manipulation and analysis tools with its powerful data structures.

The main unit of Pandas is the DataFrame, which is like an excel sheet with dozens of built-in functions for any data preprocessing or manipulation needed. There is also a data type called Series and another one called Panel. Each one of them will be explained when needed.

With Pandas, dealing with missing data or outliers can be very easy. Not only that but manipulating complete columns or rows of data can also be easy.

Pandas also supports reading and writing different file types.

Let us look at the fundamentals of Pandas. Again, it is really important that you execute the following code snippets yourself in order to understand better.

We start by importing Pandas.

```
import pandas as pd
```

The following table summarizes the different Pandas data structures.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, size immutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns
Panel	3	General 3D labeled, size-mutable array

Series is a one-dimensional array-like structure with homogeneous data, while the size is immutable. Also, the values of the data are mutable.

```
pd.Series([1,2,3])
0    1
1    2
2    3
dtype: int64
```

DataFrame, a 2-dimensional array with heterogeneous data, has mutable size, and mutable value.

```
pd.DataFrame(data= {‘col1’: [1, 2], ‘col2’: [3, 4]})
```

	col1	col2
0	1	3
1	2	4

Pandas Panels are not used widely. Thus, we will focus only on Series and Data Frames.

However, you can use Panels when your data are 3D.

Pandas also has many data reading functions such as:

- read_csv()
- read_excel()
- read_json()
- read_html()
- read_sql()

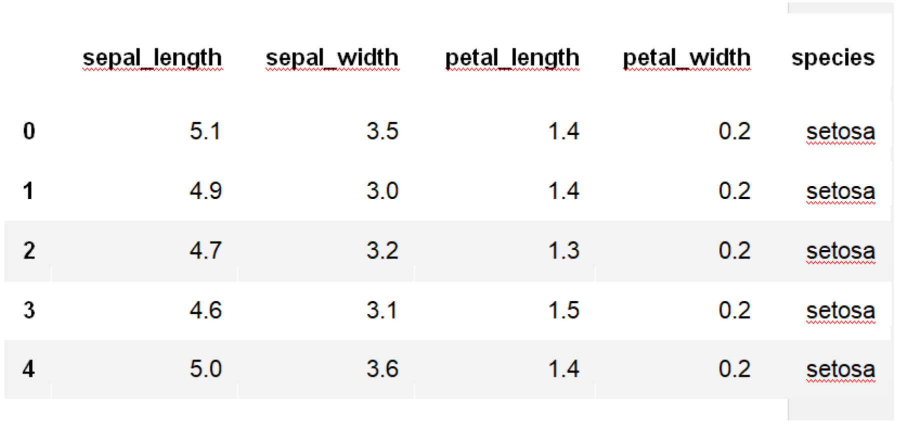
Let us now work with a real-world dataset!

The first step is to change the directory to the one containing the dataset. This can be done using the OS library.

```
import os
os.chdir(‘D:’)
os.getcwd()
‘D:\\’
```

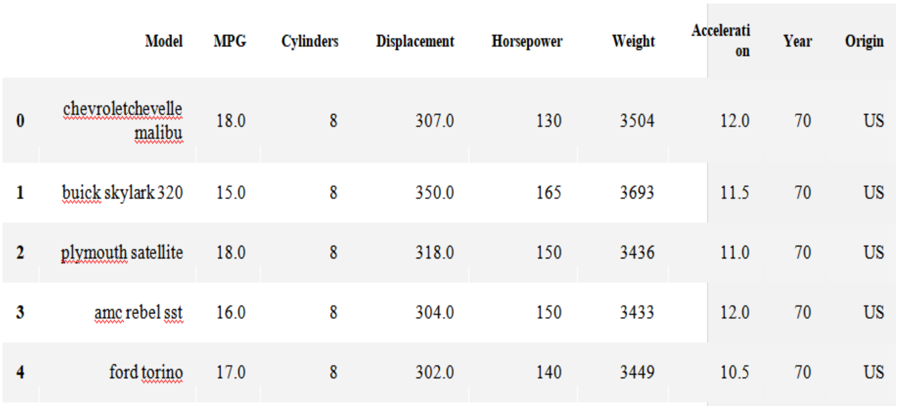
Let us now use the different reading function that we have just mentioned. Pandas has a function called *head* that enables us to view the first few elements of a specific DataFrame.

```
iris_df=pd.read_csv('iris.csv')
iris_df.head()
```



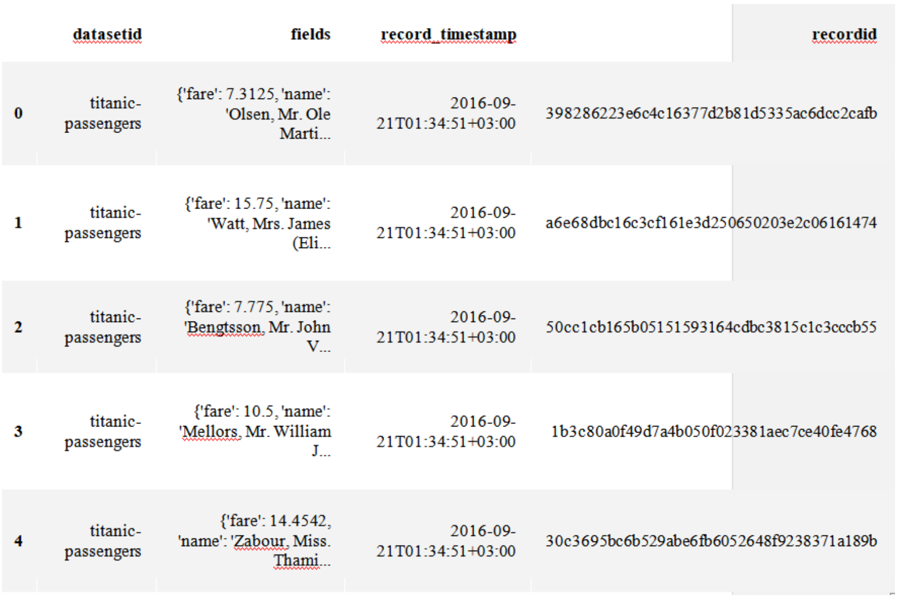
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
cars_df=pd.read_excel('cars.xls')
cars_df.head()
```



	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
0	chevroletchevelle malibu	18.0	8	307.0	130	3504	12.0	70	US
1	buick skylark 320	15.0	8	350.0	165	3693	11.5	70	US
2	plymouth satellite	18.0	8	318.0	150	3436	11.0	70	US
3	amc rebel sst	16.0	8	304.0	150	3433	12.0	70	US
4	ford torino	17.0	8	302.0	140	3449	10.5	70	US

```
titanic_df=pd.read_json('titanic.json')
titanic_df.head()
```



	datasetid	fields	record_timestamp	recordid
0	titanic-passengers	{'fare': 7.3125, 'name': 'Olsen, Mr. Ole Marti...	2016-09-21T01:34:51+03:00	398286223e6c4c16377d2b81d5335ac6dcc2cafb
1	titanic-passengers	{'fare': 15.75, 'name': 'Watt, Mrs. James (Eli...	2016-09-21T01:34:51+03:00	a6e68dbc16c3cf161e3d250650203e2c06161474
2	titanic-passengers	{'fare': 7.775, 'name': 'Bengtsson, Mr. John V...	2016-09-21T01:34:51+03:00	50cc1cb165b05151593164cdbc3815c1c3cccb55
3	titanic-passengers	{'fare': 10.5, 'name': 'Mellors, Mr. William J...	2016-09-21T01:34:51+03:00	1b3c80a0f49d7a4b050f023381aec7ce40fe4768
4	titanic-passengers	{'fare': 14.4542, 'name': 'Zabour, Miss. Thami...	2016-09-21T01:34:51+03:00	30c3695bc6b529abe6fb6052648f9238371a189b

Now, let us work with the *Cars* dataset and see how to select a column from it.

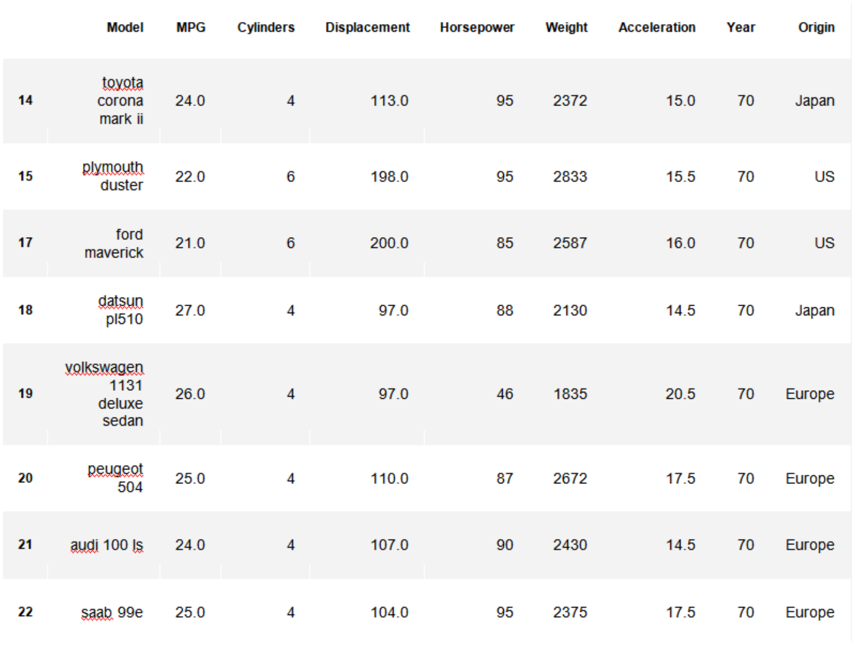
```
cars_df[‘MPG’]
0      18.0
1      15.0
2      18.0
3      16.0
```

We can also choose a specific value in a specific column and row.

```
cars_df.iloc[19,1]
26.0
```

Moreover, we can choose the values that satisfy a condition.

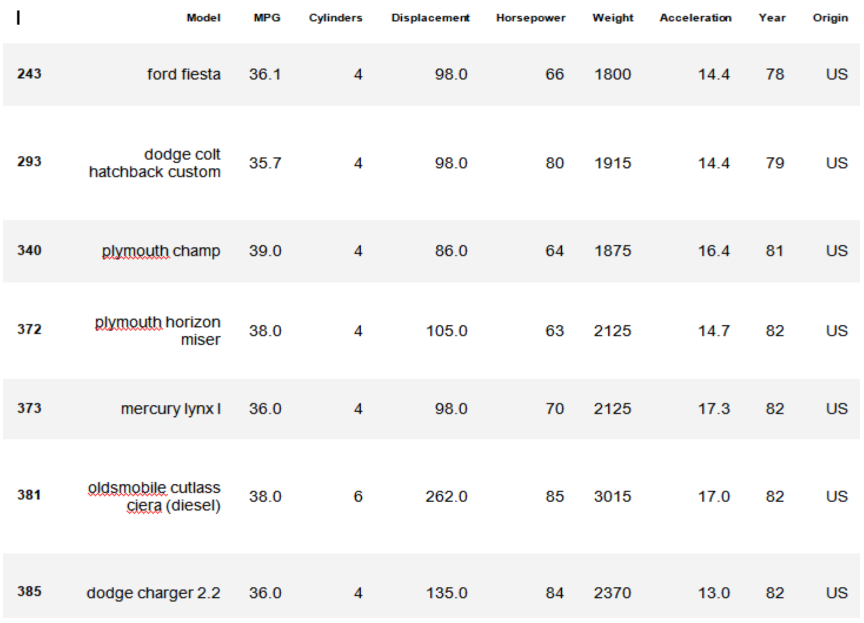
```
cars_df.loc[cars_df.MPG>20,]
```



	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
14	toyota corona mark ii	24.0	4	113.0	95	2372	15.0	70	Japan
15	plymouth duster	22.0	6	198.0	95	2833	15.5	70	US
17	ford maverick	21.0	6	200.0	85	2587	16.0	70	US
18	datsun pl510	27.0	4	97.0	88	2130	14.5	70	Japan
19	volkswagen 1131 deluxe sedan	26.0	4	97.0	46	1835	20.5	70	Europe
20	peugeot 504	25.0	4	110.0	87	2672	17.5	70	Europe
21	audi 100 ls	24.0	4	107.0	90	2430	14.5	70	Europe
22	saab 99e	25.0	4	104.0	95	2375	17.5	70	Europe

This can be done even with multiple conditions.

```
cars_df.loc[(cars_df.MPG>35)&(cars_df.Origin=='US'),]
```



	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
243	ford fiesta	36.1	4	98.0	66	1800	14.4	78	US
293	dodge colt hatchback custom	35.7	4	98.0	80	1915	14.4	79	US
340	plymouth champ	39.0	4	86.0	64	1875	16.4	81	US
372	plymouth horizon miser	38.0	4	105.0	63	2125	14.7	82	US
373	mercury lynx l	36.0	4	98.0	70	2125	17.3	82	US
381	oldsmobile cutlass ciera (diesel)	38.0	6	262.0	85	3015	17.0	82	US
385	dodge charger 2.2	36.0	4	135.0	84	2370	13.0	82	US

Finally, we can create a new column in the DataFrame that the data is saved in.

```
cars_df['MPG_per_cylinder']=cars_df['MPG']/cars_
df['Cylinders']
```

Further Readings

To know more about Pandas, you can go here:

https://pandas.pydata.org/pandas-docs/stable/

2.6. Data Visualization with Python

Matplotlib is the fundamental library in Python for plotting 2D and even some 3D data. You can use it to plot different plots such as histograms, bar plots, heatmaps, line plots, scatter plots, and many others.

Let us see how to work with it. We start by importing it.

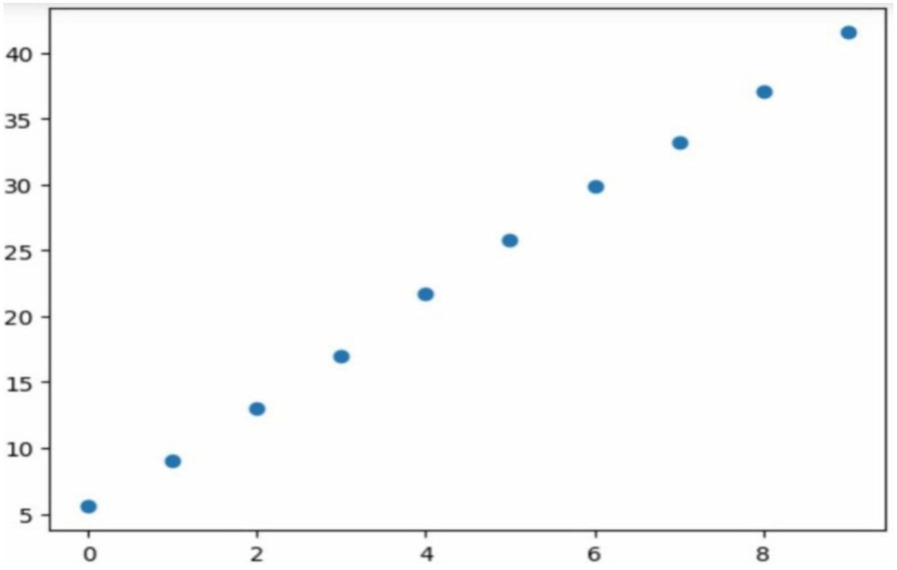
```
import matplotlib.pyplot as plt
import numpy as np
```

Then, we generate some random data to plot.

```
x = np.arange(10)
y = 4*x + 5 + np.random.random(size=x.size)
print(x)
print(y)
[0 1 2 3 4 5 6 7 8 9]
[5.55702208  9.05720778 13.01066085 17.02045126 21.69267435
25.80298672
 29.917964   33.16520441 37.12348891 41.62939263]
```

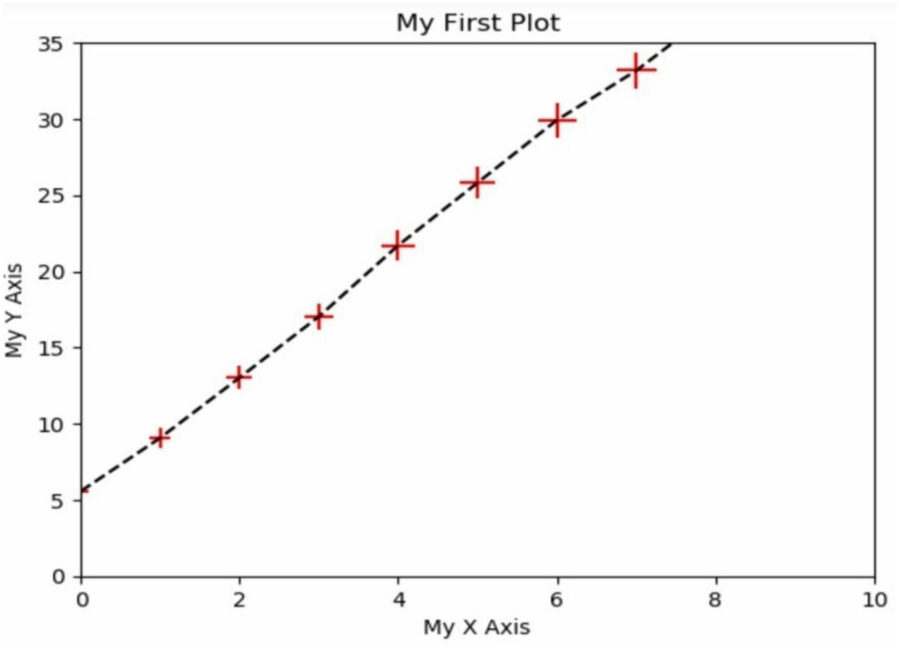
After that, we plot using the scatter method.

```
plt.scatter(x, y)
```

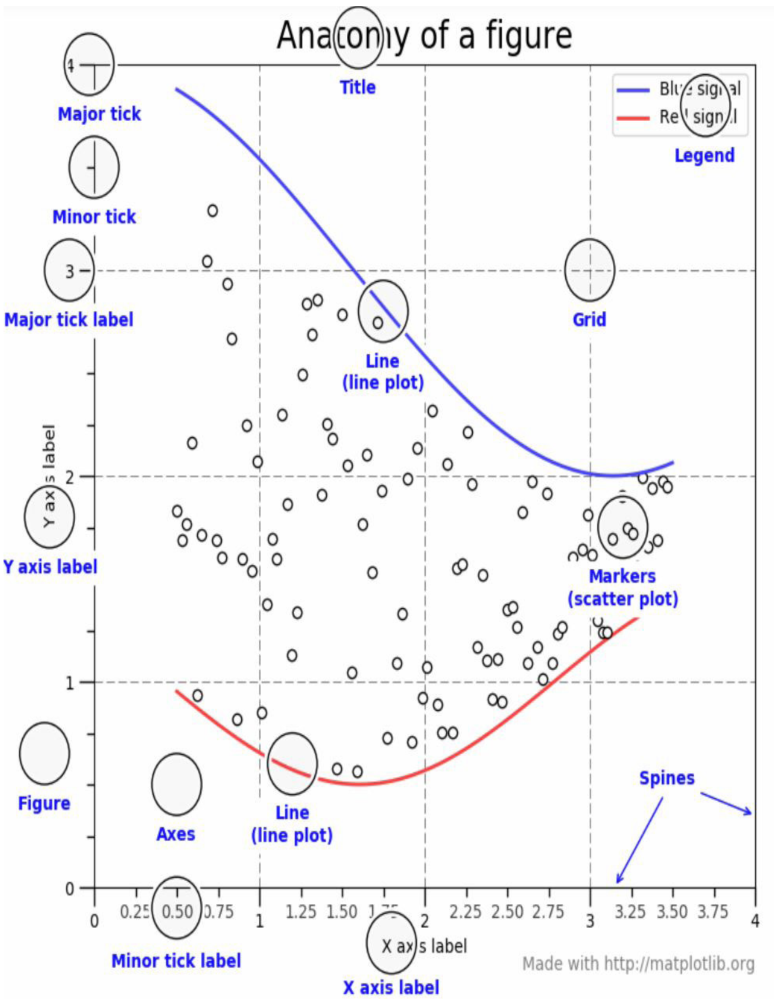


We can make the plot more beautiful.

```
# Use `o` as marker; color set as `r` (red); size
proportion to Y values
plt.scatter(x, y, marker='+', c='r', s=y*10)
# How about adding a line to it? Let's use `plt.plot()`
# set line style to dashed; color as `k` (black)
plt.plot(x, y, linestyle='dashed', color='k')
# set x/y axis limits: first two are xlow and xhigh; last
two are ylow and yhigh
plt.axis([0, 10, 0, 35])
# set x/y labels
plt.xlabel('My X Axis')
plt.ylabel('My Y Axis')
# set title
plt.title('My First Plot')
```

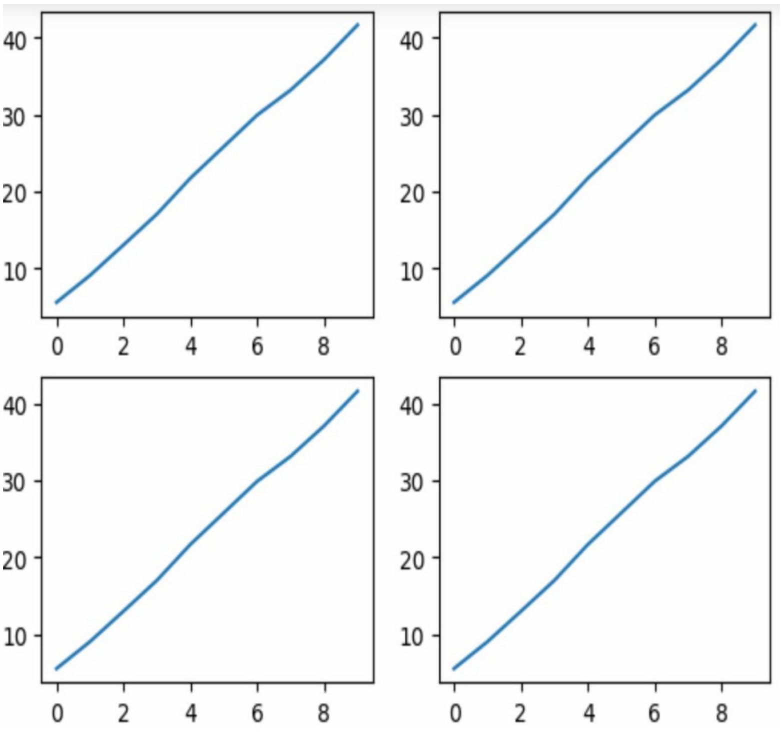


Let us understand the anatomy of the figure by the following figure.



We can also have many sub-plots as follows.

```
# Now the returned `ax` would be array with a shape a 2x2
fig, ax_arr = plt.subplots(nrows=2, ncols=2)

# Now the returned `ax` would be array with a shape a 2x2
for ax_row in ax_arr:
    for ax in ax_row:
ax.plot(x, y)
```



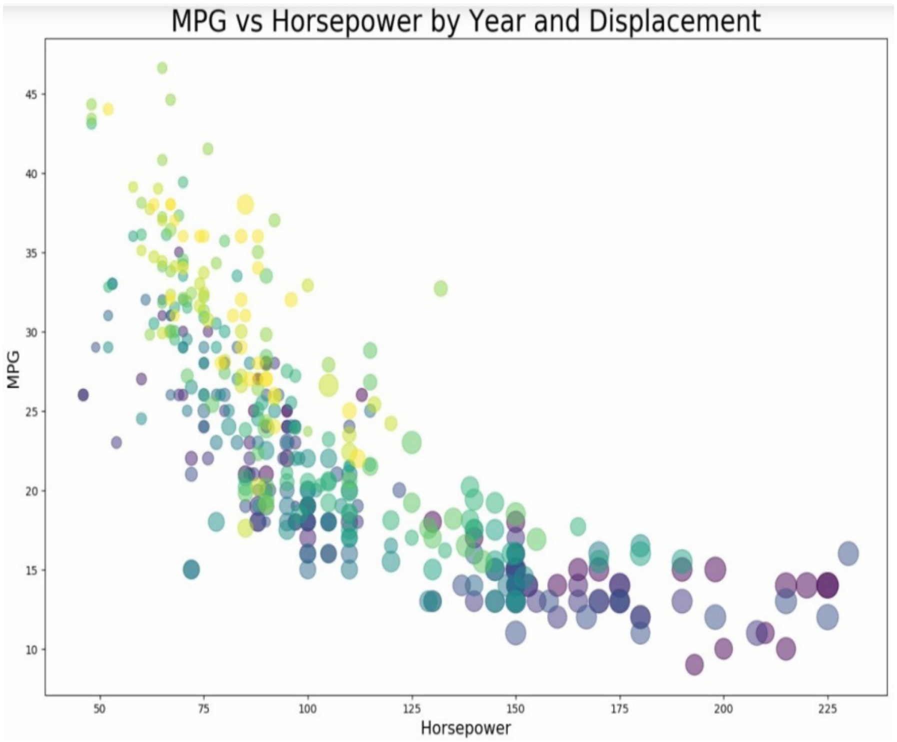
Now, let us use the visualization on a real dataset to enhance our understanding. We will be using the *Cars* dataset once again.

We start by importing the libraries, fixing the path, and loading the dataset.

```
import pandas as pd
import os
os.chdir(‘D:’)
os.getcwd()
cars_df=pd.read_excel(‘cars.xls’)
```

Then, we simply call the scatter method and pass our dataset variables.

```
plt.figure(figsize=(15,10))
plt.scatter(cars_df.Horsepower,cars_df.MPG,c=cars_
df.Year,s=cars_df.Displacement, alpha=0.5)
plt.xlabel(r'Horsepower', fontsize=15)
plt.ylabel(r'MPG', fontsize=15)
plt.title('MPG vs Horsepower by Year and Displacement',
fontsize=25)
plt.show()
```



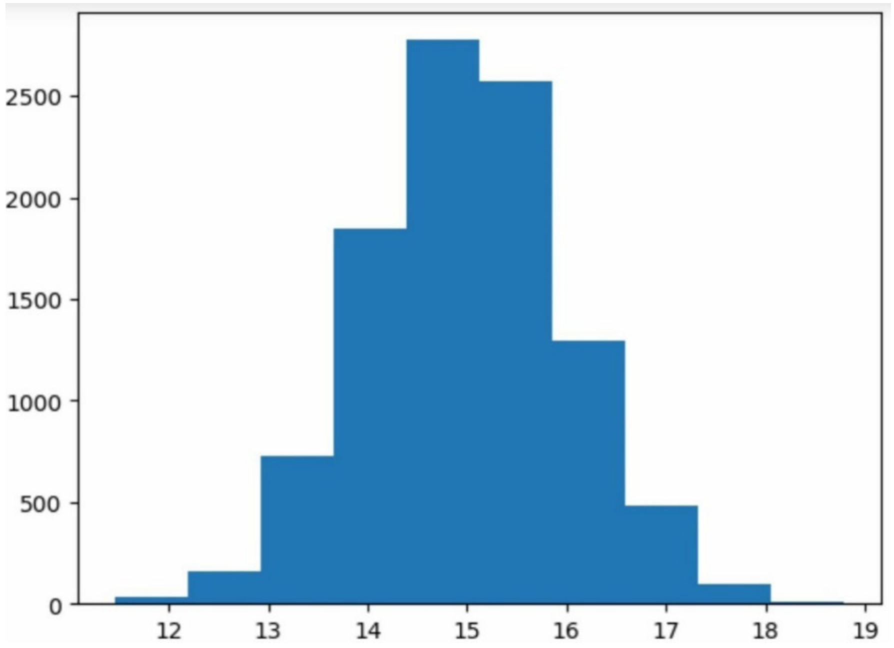
Now, let us experiment and see different kinds of plots, which are the histograms, the box plot, the bar plot, and the line plot. We will start with the **histogram**.

Let us create some random data with Gaussian distribution.

```
mu, sigma = 15, 1
gaussian_arr=np.random.normal(mu,sigma, size=10000)
np.mean(gaussian_arr), np.std(gaussian_arr, ddof=1)
(14.990546050758532, 1.0051987624650212)
```

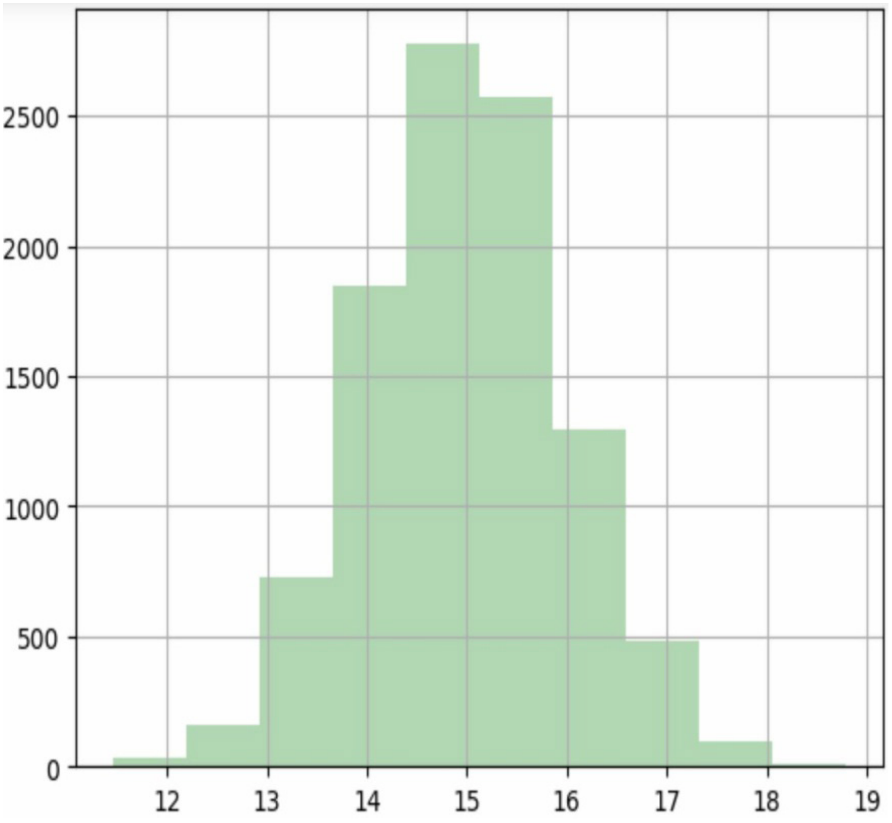
Now, let us plot this data using a histogram.

```
fig, ax = plt.subplots()
freq_arr, bin_arr, _ = ax.hist(gaussian_arr)
```



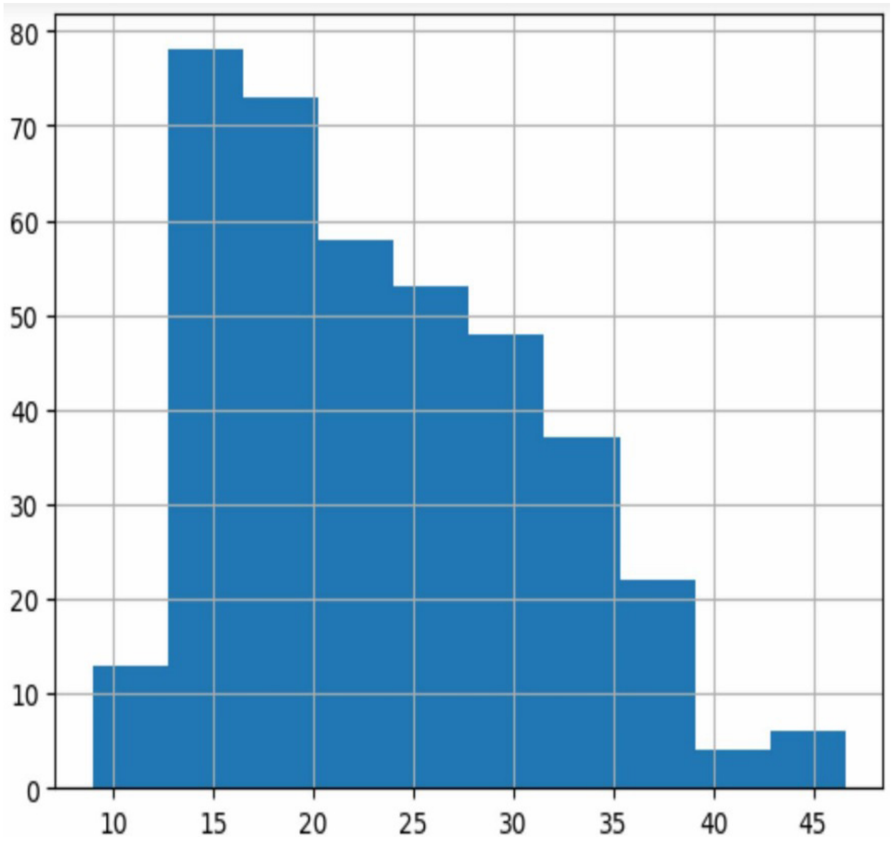
Then, we try to make it look better.

```
fig, ax = plt.subplots()
# Facecolor set to green; transparency (`alpha`) level: 30%
freq_arr,bin_arr,_=ax.hist(gaussian_arr,facecolor='g',
alpha=0.3)
# Addgrid
ax.grid()
```



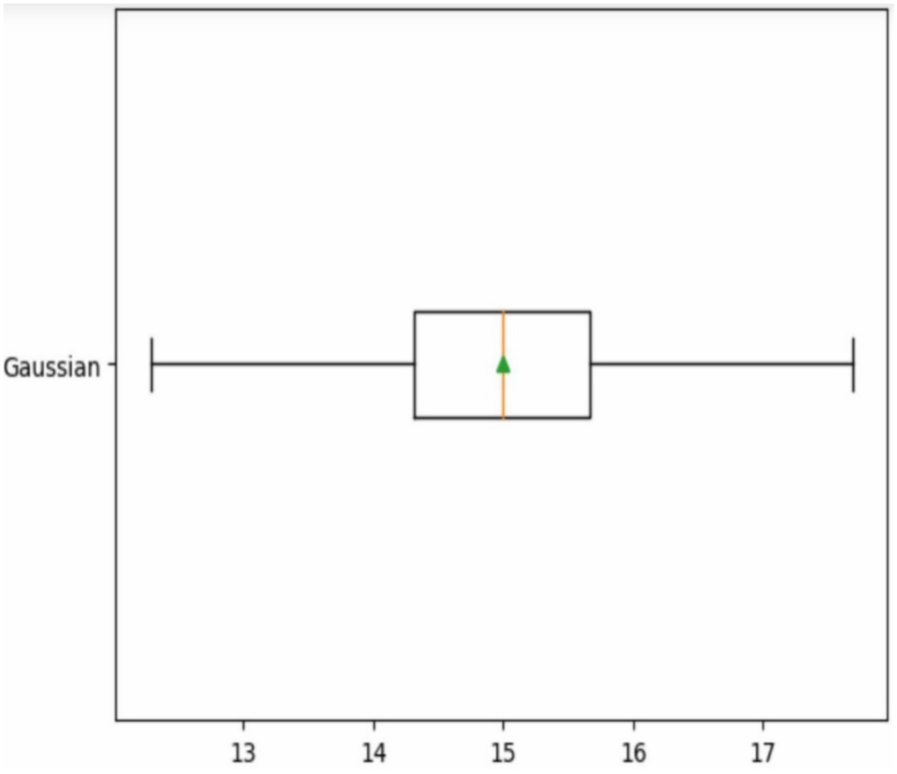
After that, we repeat the same code but on our Cars' dataset.

```
cars_df.MPG.hist()
plt.show()
```



We then use the same data but using a **box plot**.

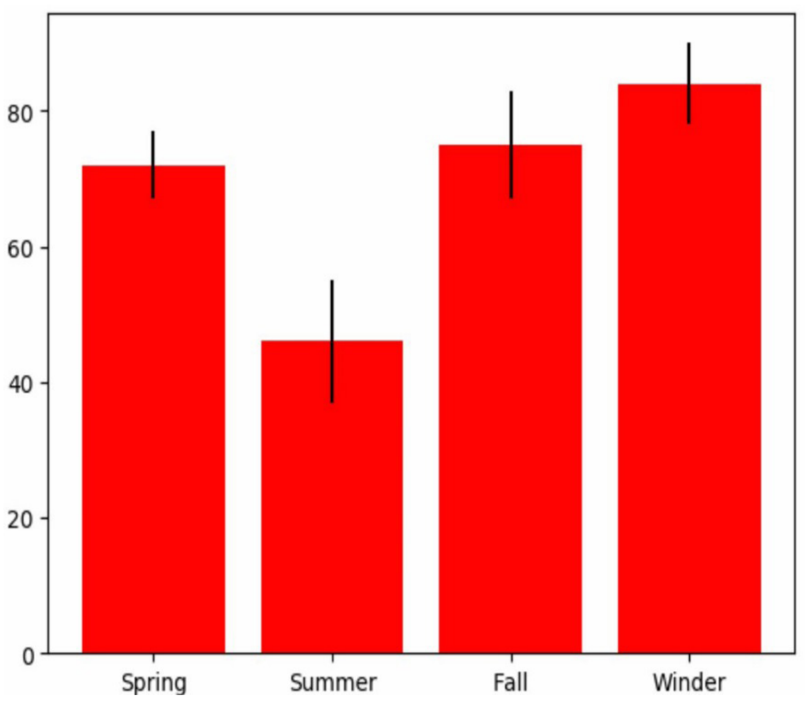
```
fig, ax = plt.subplots()
ax.boxplot(gaussian_arr,
           vert=False,  #verticle
showfliers=False, # do not show outliers
showmeans=True, # show the mean
           labels=['Gaussian'] # group name (label)
)
```



From there, we can experiment with **the bar plots** and see how they look and are used. Here, we combine them with error bars that are frequently used when we have uncertainty about our data.

```
bar_arr = np.array([‘Spring’, ‘Summer’, ‘Fall’, ‘Winder’])
freq_arr = np.random.randint(0, 100, 4)
yerr_arr = np.random.randint(5, 10, 4)
```

```
fig, ax = plt.subplots()
ax.bar(bar_arr, freq_arr, # X and Y
yerr = yerr_arr, # error bars
       color=’red’,
)
```

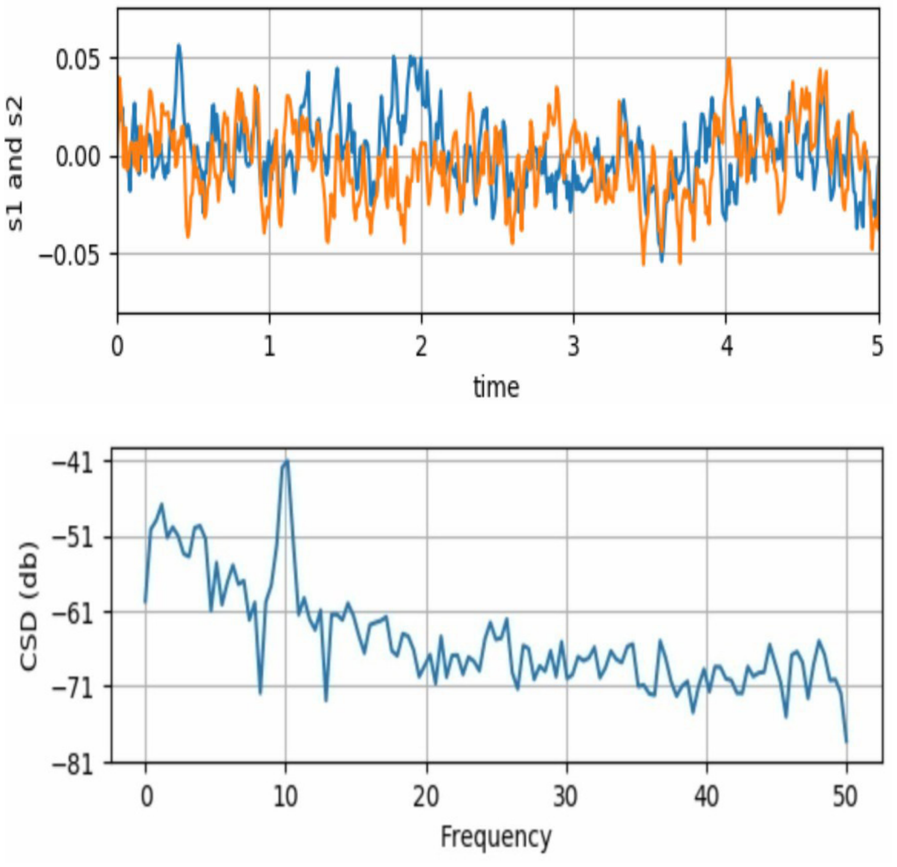


The last type of plot that we are going to mention is the **line plot**. We will artificially generate the data with the following distribution, so they can be interpreted easily in the plots.

```
dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))
nse2 = np.random.randn(len(t))
r = np.exp(-t / 0.05)
cnse1 = np.convolve(nse1, r, mode='same') * dt
cnse2 = np.convolve(nse2, r, mode='same') * dt
s1 = 0.01 * np.sin(2 * np.pi * 10 * t) + cnse1
s2 = 0.01 * np.sin(2 * np.pi * 10 * t) + cnse2
```

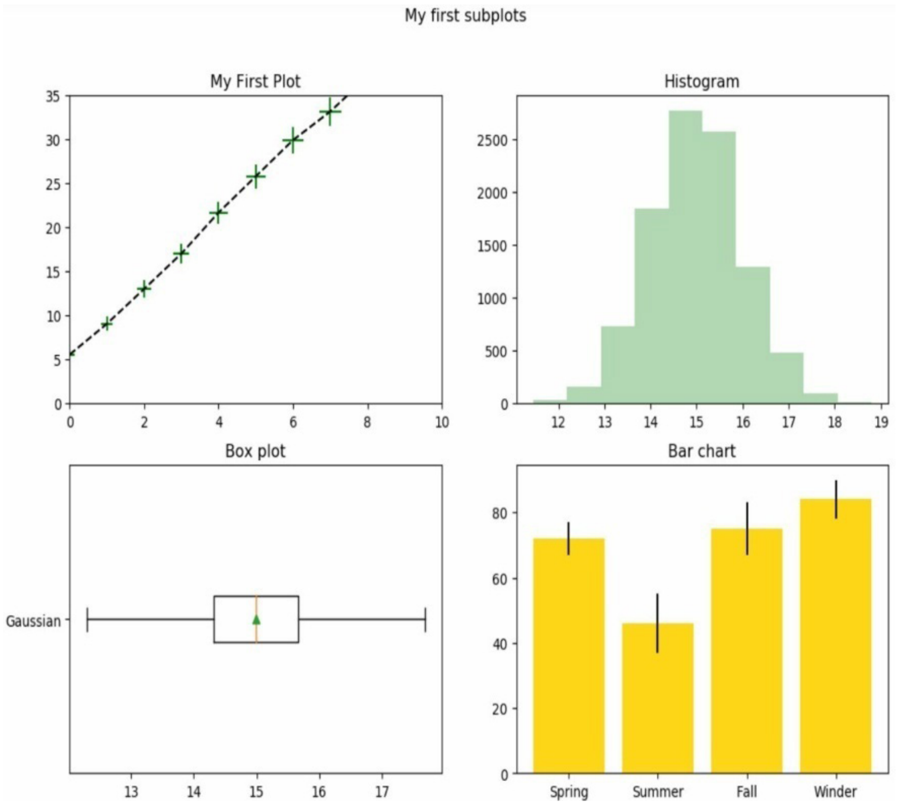
Now, we can create two plots in one plot using the sub-plots function.

```
fig, (ax1, ax2) = plt.subplots(2, 1)
# make a little extra space between the subplots
fig.subplots_adjust(hspace=0.5)
ax1.plot(t, s1, t, s2)
ax1.set_xlim(0, 5)
ax1.set_xlabel('time')
ax1.set_ylabel('s1 and s2')
ax1.grid(True)
cxy, f = ax2.csd(s1, s2, 256, 1. / dt)
ax2.set_ylabel('CSD (db)')
plt.show()
```



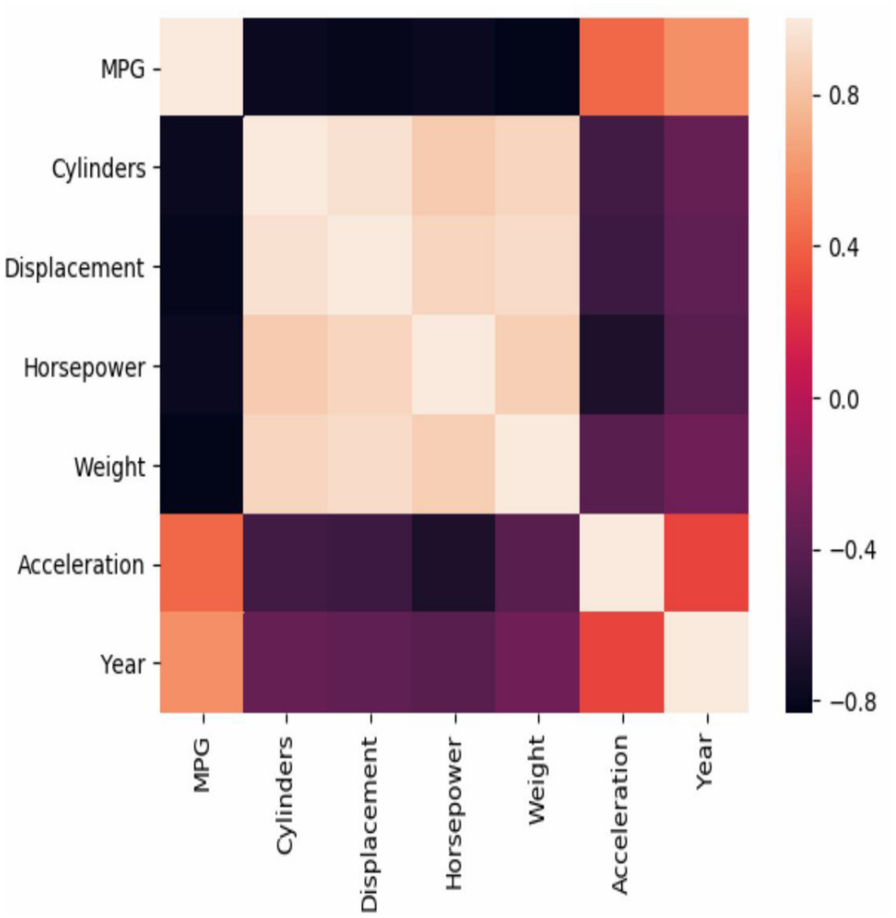
Finally, we can combine the four different types of plots that we discussed in a single plot.

```
fig, ax_arr = plt.subplots(nrows=2, ncols=2,sharex=False,
sharey=False)
fig.set_figwidth(12)
fig.set_figheight(8)
# set global title
fig.suptitle("My first subplots")
## first
ax_arr[0, 0].scatter(x, y, marker='+', c='g', s=y*10)
ax_arr[0, 0].plot(x, y, linestyle='dashed', color='k')
ax_arr[0, 0].axis([0, 10, 0, 35])
ax_arr[0, 0].set_title('My First Plot')
## second
ax_arr[0, 1].hist(gaussian_arr, facecolor='g', alpha=0.3)
ax_arr[0, 1].set_title('Histogram')
## third
ax_arr[1, 0].boxplot(gaussian_arr, vert=False,
showfliers=False,
showmeans=True, labels=['Gaussian'])
ax_arr[1, 0].set_title('Box plot')
## last one
ax_arr[1,1].bar(bar_arr, freq_arr,
yerr = yerr_arr, color='gold')
ax_arr[1, 1].set_title('Bar chart')
```



One final thing before we move on, it's worth mentioning that there is another less frequently used library called S**eaborn**, which can help us generate some good-looking graphs.

```
import seaborn as sns
sns.heatmap(cars_df.corr())
plt.show()
```



Further Readings

To know more about Matplotlib, you can get back to this tutorial here: https://matplotlib.org/contents.html.

2.7. StatsModels for Statistics

Given that this book focuses on one of the fundamental statistical methods, we will be using the StatsModels library. Statsmodel is a Python library that provides many functions that help to estimate many different statistical models, along with conducting tactical tests and statistical data exploration.

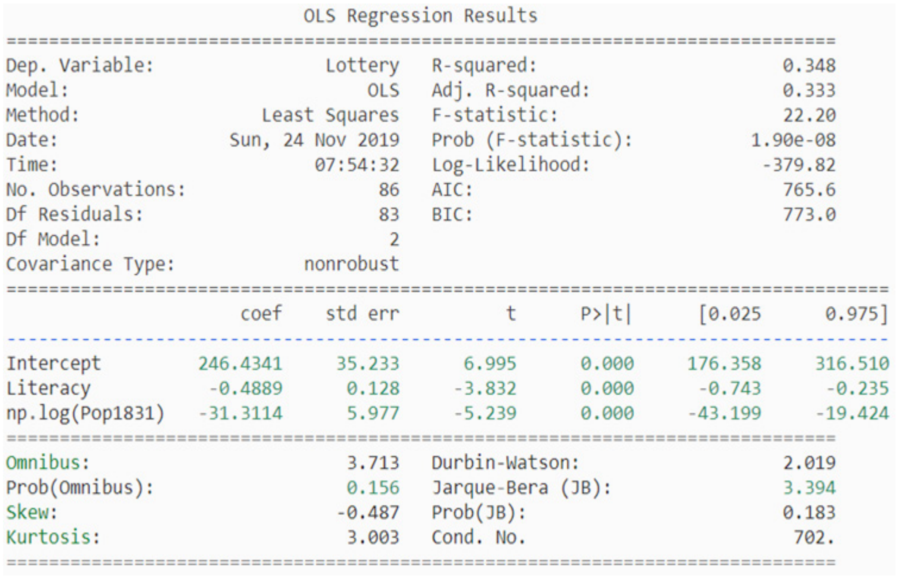
Let us see a very basic example of what Statsmodel can offer. While we will not go into details about what each line means right now, we will see how powerful this library is.

```
import numpy as np
import statsmodels.api as sp
import statsmodels.formula.api as smf
dat=sm.datasets.get_rdataset("Guerry","HistData").data
results=smf.ols('Lottery~Literacy+np.log(Pop1831)',data =
dat).fit()
```

Here, we are using a dummy dataset and doing a type of regression analysis called Ordinary Least Squares regression, which we will discuss in the last two chapters of the book. As you can see, this can be done using only one line of code.

Then, using only one line of code, we can get the result of dozens of statistical tests very easily.

```
print(results.summary())
```



```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Lottery   R-squared:                       0.348
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     22.20
Date:                Sun, 24 Nov 2019   Prob (F-statistic):           1.90e-08
Time:                        07:54:32   Log-Likelihood:                -379.82
No. Observations:                  86   AIC:                             765.6
Df Residuals:                      83   BIC:                             773.0
Df Model:                           2
Covariance Type:            nonrobust
===================================================================================
                      coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------
Intercept         246.4341     35.233      6.995      0.000     176.358     316.510
Literacy           -0.4889      0.128     -3.832      0.000      -0.743      -0.235
np.log(Pop1831)   -31.3114      5.977     -5.239      0.000     -43.199     -19.424
==============================================================================
Omnibus:                        3.713   Durbin-Watson:                   2.019
Prob(Omnibus):                  0.156   Jarque-Bera (JB):                3.394
Skew:                          -0.487   Prob(JB):                        0.183
Kurtosis:                       3.003   Cond. No.                         702.
==============================================================================
```

2.8. Why not Just Use Excel?

So, one might wonder why we do not just use excel to perform any data analysis that we want. In fact, we can! But the thing is, Python is much more powerful, easily integrable, scalable, and modular than excel. Any task that you believe can be done using excel is, without a doubt, feasible in Python. But the opposite is not true. Python has many more capabilities than excel.

Another reason to use Python instead of Excel is that all data analysis positions now require a good understanding of Python while excel is getting buried.

2.9. Summary and Exercises in Python

To summarize, in this chapter, we got hands-on experience with Python and many of its powerful data science libraries. We started with installing Python and understanding its syntax. Then, we dived into the details of Python data structures. After that, we introduced a lot of libraries and tools that we will use throughout the book, such as NumPy, Pandas, Matplotlib, and others.

If you want to start working on the exercises, try to implement everything we discussed yourself. Also, you can try exploring the other default options of all the functions we used, and see how the results will change.

3

Data Exploration and Data Analysis

In this chapter, we will understand together how to explore the data and perform preliminary analysis on it. We will start with an introduction about probability and Bayes' theorem. Then, we will see how to estimate the mean and the median of the data. After that, we will walk-through how to estimate the variability and the measures of symmetry. Following that, we will explore the measures of relationship. Then, we will understand a very important concept called the five-number summary. Finally, we will see how to perform all this in Python code.

3.1. Probability and Bayes' Theorem

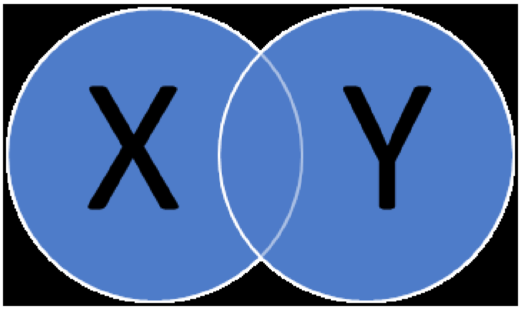
Before we scrutinize the different aspects of probability, let us first get motivated on why we should learn about them.

As you know, there are only very few things in the world that we can be sure about 100 percent, while most things we are sure about only to some extent. Thus, we need probability in order to provide a rational and scientific way to deal with this uncertainty.

Bayes' theorem is one of the pioneering probability theorems that was used in statistical learning applications. Before we discuss it, we must understand three fundamental concepts in probability, which are marginal probability, joint probability, and conditional probability.

Let us start with the **Marginal Probability**. Assume we have some event *X.* Then the marginal probability is the probability that this event occurs *regardless* of any other events. We formulate this mathematically as $P(X)$. For example, suppose that we have 6 green balls and 6 black balls. Then, the marginal probability of picking a green ball is $P(green) = 0.5$.

Now, let's talk about the second fundamental concept, which is **Joint Probability**. It is mathematically formulated as $P(X \cap Y)$, which is the intersection between two events, so they must occur together. We can visualize it using a Venn diagram as follows.



As we can see, the intersection between the two events is the joint probability. For example, if you have a traditional deck card with 52 cards, then the probability of choosing a red 7 card is $P(red \cap 7) = \frac{2}{52}$ because there are only two cards that satisfy these two conditions.

The third probability concept is the **Conditional Probability**, which is a measure of the probability of an event given that some other event had occurred. It is mathematically formulated as $P(X|Y)$, which we can translate to the probability of event *X*, given that event *Y* had occurred. For example, the probability of picking up a 7 from the deck given that it is a red card is $P(7|red) = \frac{1}{26}$, because given that it is a red card, we know that it has to be one of the 26 cards which are red.

The beauty of probability comes from the linking between the three concepts into one equation which is

$$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}$$

The proof of this theorem is out of the scope of this book, so we only need to understand that these three main probability concepts are interconnected.

Now, given that we understand the basic concepts of probability, let us discuss **Bayes' theorem**.

In a nutshell, Bayes' theorem has the advantage of providing us with a method to update our beliefs based on new evidence. Let us understand how.

For example, suppose that we are trying to estimate a probability that a given person will be accepted for graduate studies or not. If you only have his or her exam grades, you will provide a different probability than if you have additional evidence such as the number of published papers.

Bayes' theorem can be written as follows:

$$P(X|Y) = \frac{P(Y|X) * P(X)}{P(Y)}$$

This theorem combines both conditional probability and marginal probability. Also, it is derived from joint probability.

Here, we want to get the probability of event *X* given that *Y* is the new evidence that we have. We call this the *posterior*, which would be *the probability of getting accepted, given that this person has published papers*.

We call $P(Y|X)$ the *likelihood*, as it the probability of observing the new evidence, given our initial hypothesis. This can be translated for our example as follows: *the probability of having published papers given that the person gets accepted*.

The marginal probability $P(X)$ is called the *prior*, as it is the probability of our hypothesis without any additional prior information. Referring to our example, we can say that this maps to *the probability of getting accepted*.

Finally, $P(Y)$ is the *marginal likelihood*, which could be translated to *the probability of having published papers.*

In order to understand Bayes' theorem, let us look at a numerical example. Assume that the probability of getting accepted at this university is 10 percent. Assume also that the probability of publishing papers is 30 percent, which means that out of every 10 people applied for this university, there are 3 people who have published papers. Also, assume that 20 percent of people who got accepted have published papers, so $P(published|accepted) = 0.2$.

Without having the new evidence, which is the published papers, we would have said that the probability of getting accepted is the prior probability, which is $P(accepted) = 0.1$ $P(accepted) = 0.1$. But now, using Bayes' rule, we can have a more precise calculation as follows:

$$P(accepted|published) = \frac{P(published|accepted) * P(accepted)}{P(published)} = \frac{0.2 * 0.1}{0.3} = 0.066$$

And that is how Bayes' theorem works.

3.2. Estimates Mean, Median, and Mode

We can combine the estimation of the mean, median, and mode into one term, which is the *measures of central tendency*. This is because they are used to describe the data by identifying the central position. This identification can be made by three measures which are:

1. The **mean** is equal to the sum of all the values divided by the number of values, which is simply taking the average.
2. The **median** is calculated by sorting the dataset and getting the middle value.
3. The **mode** which is basically the most occurring value in the dataset.

Let us take a numerical example and calculate the three measures. Suppose our data is the following:

{13,40,50,50,90,18,30,50,30,70}

So, first, we calculate the **mean** using the following equation:

$$mean = \frac{13 + 40 + 50 + 50 + 90 + 18 + 30 + 50 + 30 + 70}{10} = 44.1$$

Then, we calculate the **median** by first sorting the data

{13,18,30,30,40,50,50,50,70,90}

After that, we take the middle value as our median, which will be $\frac{40+50}{2} = 45$. As the number of examples is even, so we take the average of the two middle values.

Finally, we calculate the **mode** by observing the most occurrent value, which is 50 in our case.

3.3. Estimates Variability

Estimating the variability of the data means that we want to measure the spread of the data.

There are many measures to achieve this task, but we will focus only on the most important three, which are:

1. **The range** is the difference between the smallest and the largest value of the data. Note that this does not consider all the values in the data, but instead, it takes only the minimum and the maximum values. For example, if we have {10,8,20,40,12,15,30,25} as our examples, then the range will be (8-40) only.
2. **Variance** measures how far is the sum of the squared distances from each point to the mean. This can indicate the dispersion around the mean as it is the average of all squared deviations. The equation to calculate the variance is $\sigma^2 = \frac{\sum_{i=1}^{n} x_i - \mu}{n}$.
3. **Standard deviation** is essentially the square root of the variance. It is the most commonly used measure as it has the same units as the data.

Let us take a numerical example to understand how we can calculate the measures of spread. Suppose that we have the following dataset: {3,5,6,9,10}

$$mean = \frac{3+5+6+9+10}{5} = 6.6$$

$$\begin{aligned} &Variance \\ &= \frac{(3-6.6)^2+(5-6.6)^2+(6-6.6)^2+(9-6.6)^2+(10-6.6)^2}{5} \\ &= 6.64 \end{aligned}$$

$$standard\ deviation = \sqrt{Variance} = 2.576$$

3.4. Measures of Symmetry

Now, let us discuss the measures of symmetry or shape. In this measure, we try to identify if the data is centered, which means the number of examples on the left side of the center is nearly equal to the number of examples on the right side of the center.

The most used measure in this class is the skewness of the data distribution. We say that the data is **positively skewed** if the mean > median > mode, and **negatively skewed** if the mean<median<mode. By identifying any skewness in the data, we can use different preprocessing techniques to make the data symmetric.

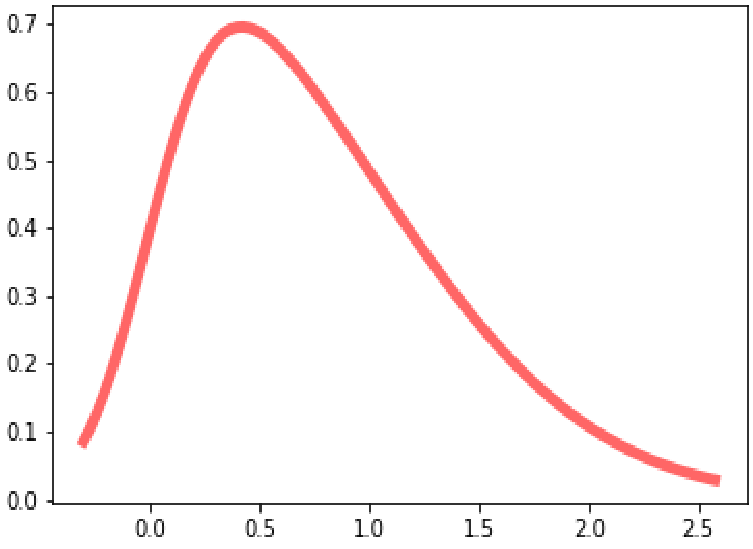


Figure 1: Right-Skewed (Positive Skewed)

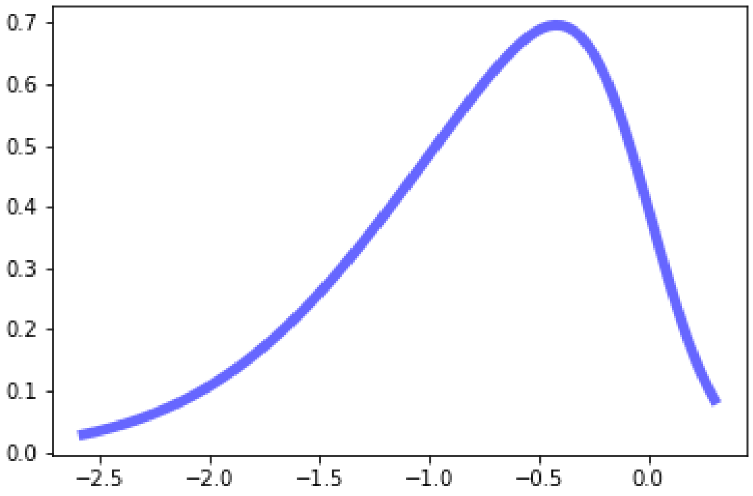


Figure 2: Left-Skewed (Negative Skewed)

3.5. Measures of Relationship

These measures are mainly used to compare and find the relation between two or more different variables. There are two main measures to do so, which are:

1. **Covariance,** which measures the relationship between the variability of two or more different variables by calculating the effect of changing one variable values on the other variable's values. We use this measure to have an idea about the direction of the relationship—whether the variables tend to move in tandem or show an inverse relationship. However, covariance does not indicate the strength of this relationship, nor the dependency between the variables as it is not normalized. The equation to calculate the covariance is the following:

$$Cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{n}$$

2. **Correlation** is simply the covariance after normalization. This normalization is done by calculating the standard deviation of both variables and dividing the covariance by them. By using correlation, we can measure the strength of the relationship between different variables. This is because the correlation is a pure value that does not have any units. The range of the correlation is from −1 to 1, as −1 indicates a pure negative correlation. This means when one variable increases, the other variable decreases in the same way. If the correlation value is 1, then there is a pure positive correlation. Finally, if the correlation is zero, then the two variables are

independent of each other. The equation to calculate the correlation is the following:

$$\rho = \frac{Cov(X,Y)}{\sigma_x \sigma_y}$$

3.6. Five-Number Summary

Before we end this chapter with the hands-on exercise, we need to address a key concept, which is the five-number summary. It consists of the most extreme values in the dataset, which are the minimum and the maximum values, along with the lower and upper quartiles, and the median. So, what is "*quartile*"?

Suppose that we have 100 observations in our dataset. We can divide it into four *quarters*. If the values of the dataset are [1, 2, 3, ..., 99, 100], then we can say that the first quartile *Q1* is 25, the second quartile *Q2* is 50, the third quartile *Q3* is 75, right?

But if the dataset does not have this distribution, then we calculate Q1 as the median of the lower half of the data, and Q3 as the median of the upper half of the data.

Suppose we have the following numbers [1, 2, 5, 6, 7, 9, 12, 15, 18, 19, 27], we arrange them, and we find the median which is **9**. Then, we split the data into two parts (1, 2, 5, 6, 7), 9, (12, 15, 18, 19, 27). Now, it is obvious that Q1 is **5,** while Q3 is **18**.

Now, back to the five-number summary. We use it as each value in this summary describes a specific part of the dataset as follows:

- **Median:** the center of the dataset.

- **Upper and Lower quartiles:** spans the middle half of the dataset.
- **Minimum and Maximum:** provide additional information about the actual dispersion of the data.

That is why we consider the five-number summary a useful measure of spread.

3.7. Hands-on Exercises in Python

Now, let us examine how we can convert all these concepts to code. We will start with the probability concepts. In a standard deck of 52, there are 4 aces.

We can get the probability of drawing an ace by dividing the number of possible event outcomes, which is 4, by the sample space, which is 52.

$$P(X) = \frac{4}{52}$$

```
cards_numb = 52
aces_num = 4
ace_probability = aces_num / cards_numb
percent_ace_probability = ace_probability * 100
print(str(round(percent_ace_probability, 0)) + '%')
8.0%
```

Now, we can convert this piece of code into a function that can take the event outcomes and the sample space and, based on that, would return the probability.

```
# Determine the probability of drawing a heart
hearts = 13
heart_probability = event_probability(hearts, cards)
print(str(heart_probability) + '%')
25.0%
```

```
face_cards_num = 12
face_card_probability = event_probability(face_cards_num,
cards)
23.1%
```

```
queen_of_hearts = 1
probability_queen_of_hearts =
event_probability(queen_of_hearts, cards)
print(str(probability_queen_of_hearts) + '%')
1.9%
```

Now, let us see how to get the mean, median, and mode using Python.

```
n_num = [13,40,50,50,90,18,30,50,30,70]
n = len(n_num)
get_sum = sum(n_num)
mean = get_sum / n
print(mean)
44.1
```

```
n_num.sort()
if n % 2 == 0:
    median1 = n_num[n//2]
    median2 = n_num[n//2 - 1]
    median = (median1 + median2)/2
else:
    median = n_num[n//2]
print(median)
45
```

```
from collections import Counter

data = Counter(n_num)
get_mode = dict(data)
for k,v in get_mode.items():
    if v==max(list(data.values())):
mode = k
print(mode)
50
```

Finally, let us also see how to implement the range, the variance, and the standard deviation.

```
numbers=[10,8,20,40,12,15,30,25]
minimum = min(numbers)
maximum = max(numbers)
range_numbers = maximum - minimum
print(range_numbers)
32
```

```
import numpy as np
numbers = [3,5,6,9,10]
variance= np.var(numbers)
print(variance)
6.64
```

```
std = np.sqrt(variance)
print(std)
2.576
```

4

Pandas, Matplotlib, and Seaborn for Statistical Visualization

In this chapter, we will explore in great detail how to perform data visualization using Pandas, Matplotlib, and Seaborn. We will start by discussing pie charts, bar charts, and box plots. Then we will go through frequency tables and histograms. Finally, we will end with the time-series charts. All the explanations in this chapter will be using hands-on examples in Python.

4.1. Pie Charts

Pie charts are widely used in visualization techniques in business and data analysis. To plot a pie chart, we do the following in Python.

First, we import the Matplotlib and assign some dummy variables as labels.

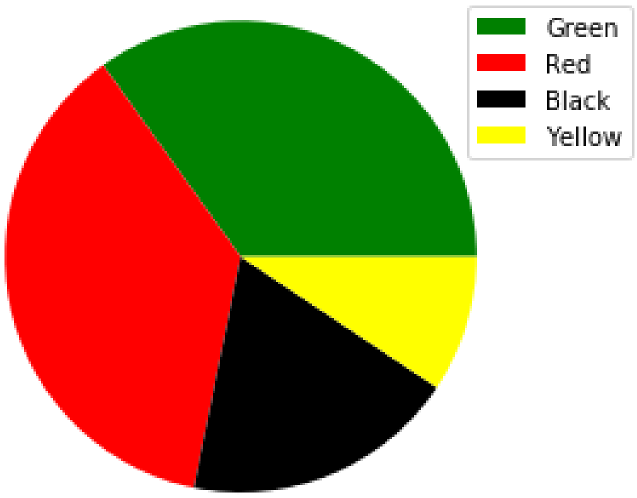
```
import matplotlib.pyplot as plt
labels = [‘Green’, ‘Red’, ‘Black’, ‘Yellow’]
```

Then, we assign a size and a color for each part of the pie chart.

```
sizes = [38.4, 40.6, 20.7, 10.3]
colors = [‘green’, ‘red’, ‘black’, ‘yellow’]
```

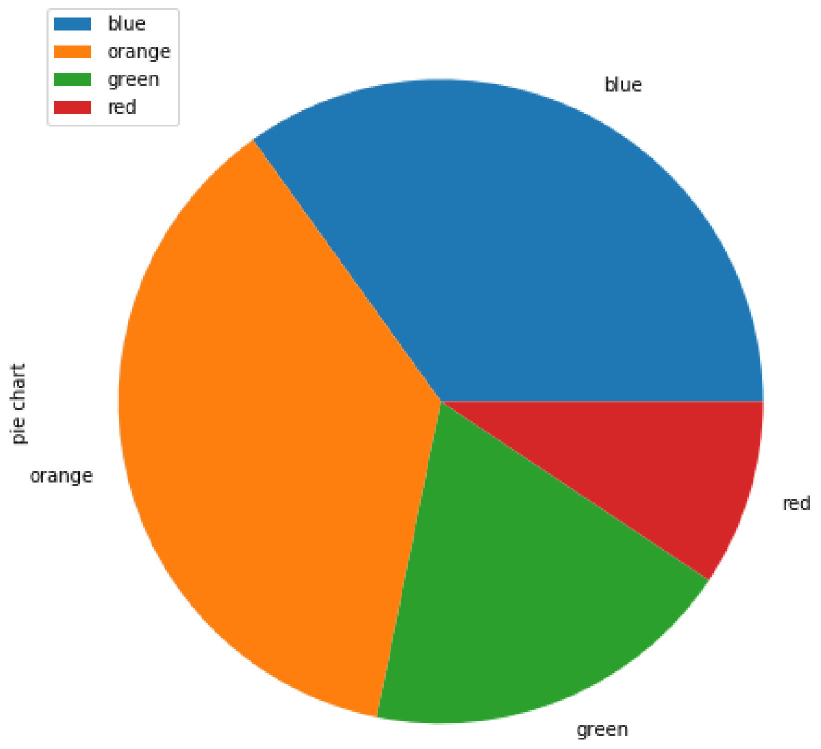
Then, we call the pie function from Matplotlib. We give it the sizes and the colors. After that, we insert a legend along with the axis. Finally, we show the plot using the show function.

```
patches, texts = plt.pie(sizes, colors=colors)
plt.legend(patches, labels, loc=”best”)
plt.axis(‘equal’)
plt.tight_layout()
plt.show()
```



Let us plot the same plot but using the Pandas library.

```
import pandas as pd
df = pd.DataFrame([38.4, 40.6, 20.7, 10.3], index=[‘blue’,
‘orange’, ‘green’, ‘red’], columns=[‘pie chart’])
# make the plot
df.plot(kind=’pie’, subplots=True, figsize=(8, 8))
```



Now, let us move to bar graphs.

4.2. Bar Graphs

We will plot the same data using both Seaborn and Matplotlib.

Let us start with Seaborn. The first step is to import the needed libraries.

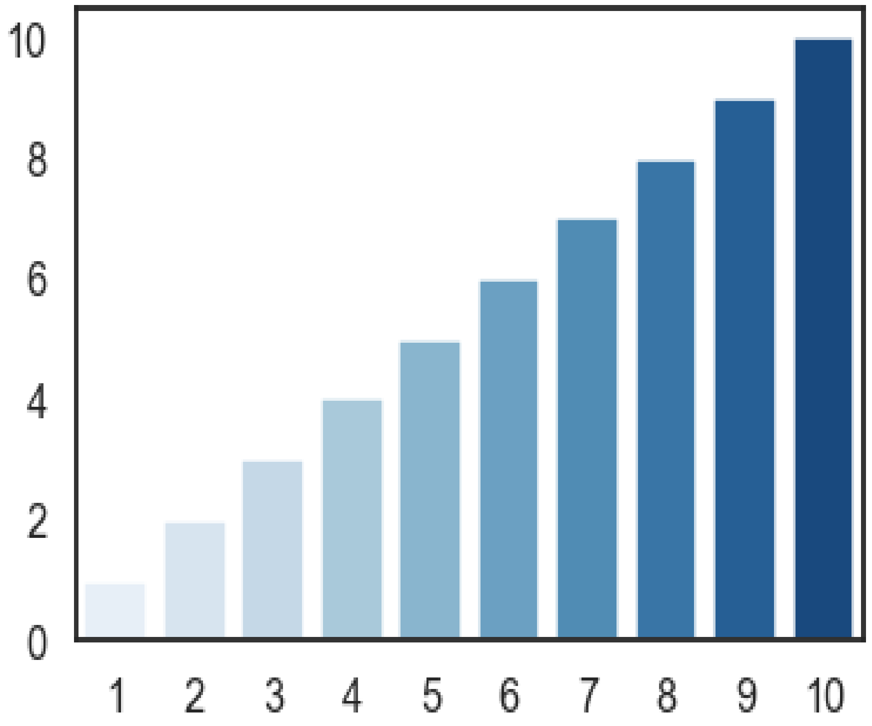
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Then, we set up the Matplotlib figure and generate some dummy data.

```
f, ax1 = plt.subplots(1, 1)
x = np.array(list(range(1,11)))
y1 = np.arange(1, 11)
```

Then, we plot the Seaborn bar plot by passing the needed parameters to the library's function as follows.

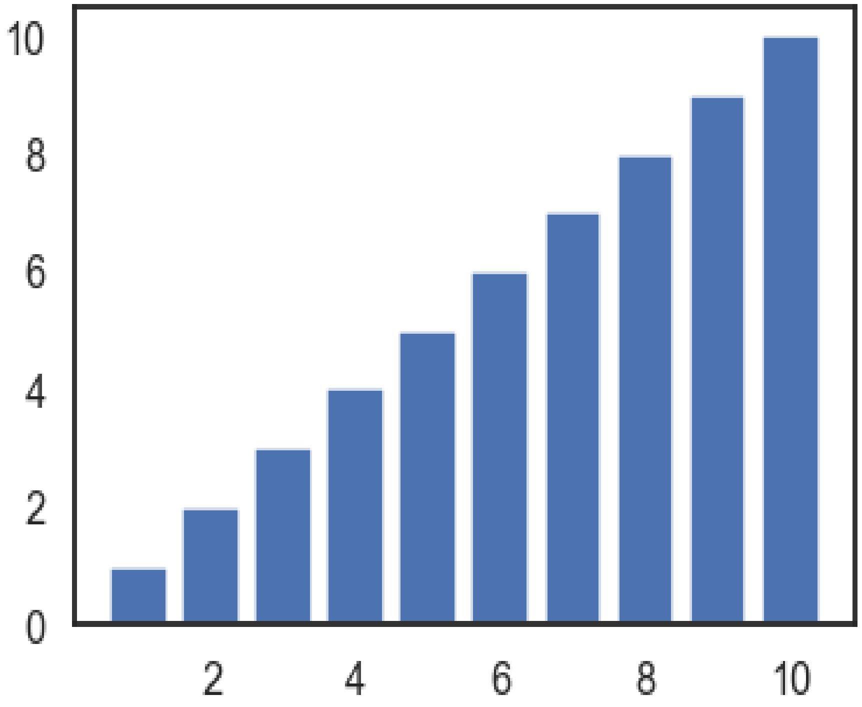
```
sns.barplot(x=x, y=y1, palette=”Blues”, ax=ax1)
plt.show()
```



Of course, you can add titles and legends very easily, as stated in the documentation.

Now, let us do the same using Matplotlib. It is extremely simple, although not that beautiful.

```
plt.bar(x,y1, label = ‘Bar1’)
```



4.3. Box Plots

In the previous chapter, we discussed a key concept called the five-number summary, which consists of the minimum, the maximum, the median, and the first and third quartiles. To visualize the five-key summary, we use a box plot.

We start by importing the needed libraries.

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

Then, we generate some random data.

```
s = np.random.rand(50) * 100
c = np.ones(25) * 50
high = np.random.rand(10) * 100 + 100
low = np.random.rand(10) * -100
data = np.concatenate((s, c, high, low))
df = pd.DataFrame(data)
```

Before generating the box plot, let us print the five-number summary.

```
df.describe()
```

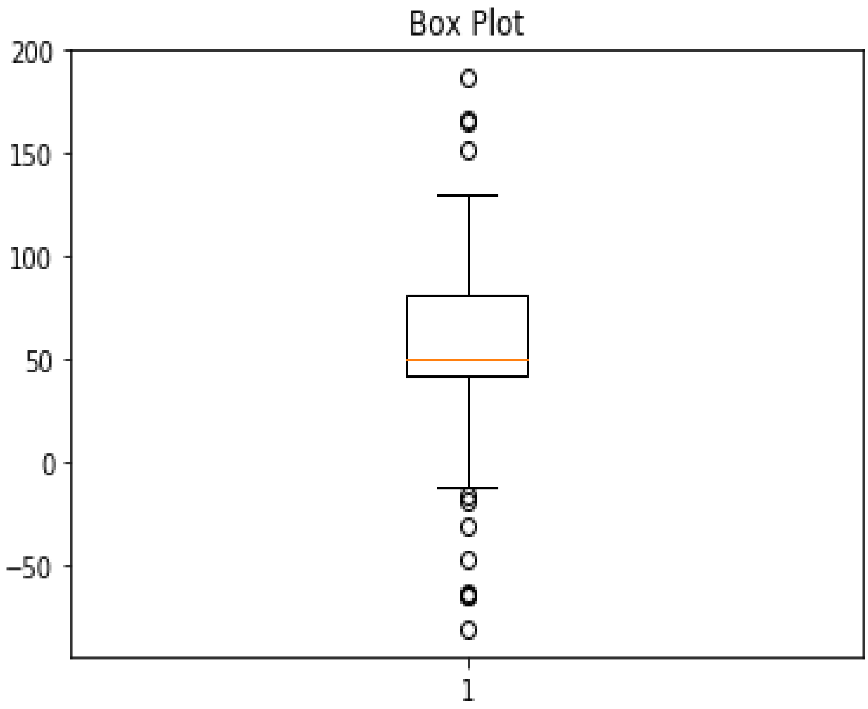
	0
count	95.000000
mean	49.081276
std	51.327901
min	-93.891546
25%	28.195712
50%	50.000000
75%	67.254546
max	197.333765

```
print(min(data))
-81.2
```

```
data.sort()
n = len(data)
if n % 2 == 0:
    median1 = data[n//2]
    median2 = data[n//2 - 1]
    median = (median1 + median2)/2
else:
    median = data[n//2]
print(median)
50
```

Now, let us plot the box plot and see if it visualizes our results.

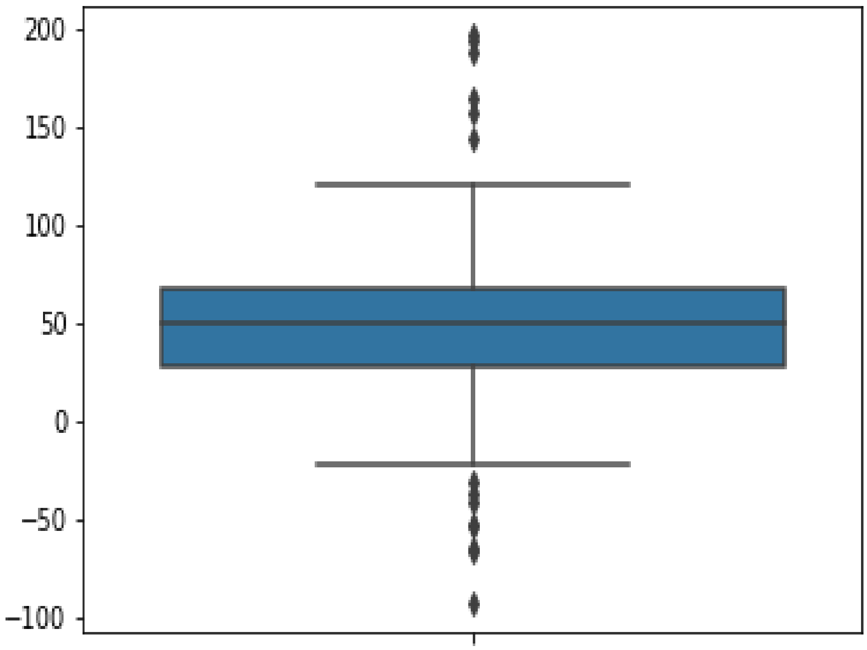
```
fig1, ax1 = plt.subplots()
ax1.set_title('Box Plot')
ax1.boxplot(data)
```



As we see, the most extreme values are around -81 and 197, while the orange line is around 50, the median. Also, the lower end of the box is around 28, the first quartile, and the upper end is around 67, the third quartile.

Let us plot the same graph using Seaborn also.

```
import seaborn as sns
ax = sns.boxplot(x=df,orient='v')
```



4.4. Frequency Table and Histogram

A frequency table is, just as its name states, a counting table for the frequency of each variable. This can then be visualized using a histogram, which is critical for understanding the distribution of the data and detecting any problem with it visually, such as skewness.

We start by importing pandas.

```
import pandas as pd
```

Then, we create a dummy column where the variables are repeated several times.

```
df = pd.DataFrame({‘names’: [‘Ahmed’, ‘SciencesAI’,
‘Ahmed’, ‘Ahmed’, ‘Jack’,
                          ‘Rose’, ‘Jack’, ‘Ahmed’,
‘SciencesAI’,’SciencesAI’]})
```

Now, we can create a frequency table using the *value_counts* function in Pandas.

```
count = df['names'].value_counts()
print(count)
Ahmed         4
SciencesAI    3
Jack          2
Rose          1
```

Now, let us visualize it as a histogram.

```
import seaborn as sns
sns.catplot(x="names", kind="count",data=df)
```

4.5. Time-Series Charts

Finally, we are going to end this chapter by understanding how to plot time-series data in Python.

Time series is considered as a sequence of observations that are recorded at fixed time intervals, which can be annually, monthly, daily, hourly, or anything. Visualizing the time series can help us understand any abnormalities in the data and can also help us gain insights.

As this is the last part of the chapter, we are going to work with a real dataset, which is available online and can be loaded easily using Pandas. This dataset contains the monthly sales figures of anti-diabetic drugs in Australia during the period from 1991 to 2008.

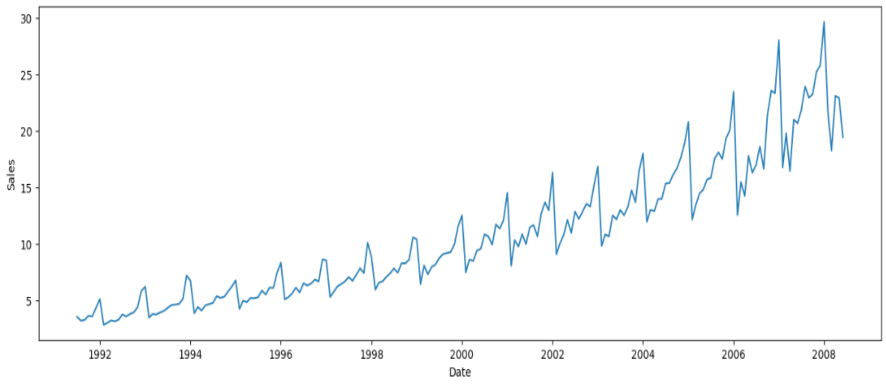
We start, as always, by loading the needed libraries, along with loading the dataset in a Pandas DataFrame.

```
from dateutil.parser import parse
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
plt.rcParams.update({'figure.figsize': (10, 7), 'figure.dpi':
120})

# Import as Dataframe
df = pd.read_csv('https://raw.githubusercontent.com/
selva86/datasets/master/a10.csv', parse_dates=['date'])
df.head()
```

	date	value
0	1991-07-01	3.526591
1	1991-08-01	3.180891
2	1991-09-01	3.252221
3	1991-10-01	3.611003
4	1991-11-01	3.565869

Now, let us plot the data as a time series.

```
plt.figure(figsize=(16,5), dpi=100)
plt.plot(df['date'], df['value'])
plt.gca().set(xlabel='Date', ylabel='Sales')
plt.show()
```


Sales
Date
30
25
20
15
10
5
1992
1994
1996
1998
2000
2002
2004
2006
2008

5

Exploring Two or More Variables and Categorical Data

In this chapter, we will explore in detail how to deal with two or more variables, as well as how to deal with categorical data, as they require special treatment. To do so, we will discuss what is meant by the mode and the expected value. After that, we will dive more into a data visualization technique suitable for visualizing more than one variable, which is a scatter plot. Then, we will discuss a very important concept for dealing with multi-variable data called correlation. From there, we will discuss the difference between categorical and numerical data. Finally, we will end this chapter with a discussion and exercise on visualizing multiple variables.

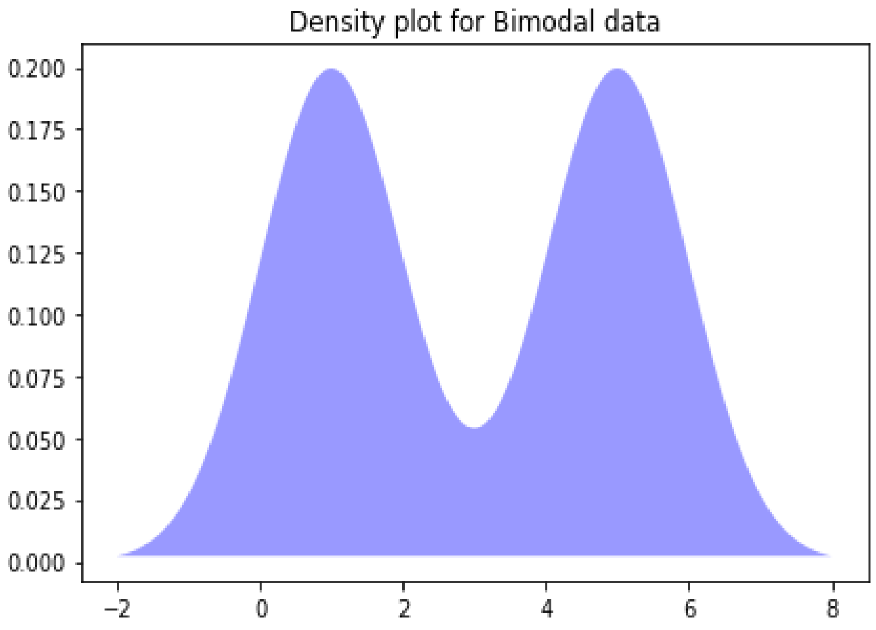
5.1. Mode and Histogram Plot

To start with, let us understand what is meant by the *mode* of the data. You might say that we already covered this concept in chapter three when we discussed data exploration and analysis, and that is partially true. In chapter three, we were only focusing on numerical data with only one variable. But here, we are discussing categorical and multi-variable data.

To recap, we said that we have three measures of center, which are the mean, the median, and the mode. And we discussed what each one of them means, when it should be used, and how to calculate it. As we are focusing here on the mode, let me tell you that it is basically *the most frequently observed value.*

We say that the data have no mode if no value is repeated more than the other values; unimodal if they have one value that is repeated more than the other values; bimodal if they have two values that are repeated more than the other values an equal number of times; trimodal, if they have three values that are repeated more than the other values an equal number of times, and so on.

The following figure is a clear example of a bimodal data.



To generate the same figure, let us start with our first exercise. Open your Python Jupyter notebook or terminal and start writing code.

We start by importing our needed libraries, which are NumPy, Matplotlib, and SciPy.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

Then we generate 100 random values that follow a normal distribution with two peaks equal to each other.

```
N = 100
np.random.seed(1)
X_plot = np.linspace(-2, 8, 1000)[:, np.newaxis]
true_dens = (0.5 * norm(1, 1).pdf(X_plot[:, 0])
             + 0.5 * norm(5, 1).pdf(X_plot[:, 0]))
```

Finally, we prepare the figure and populate it with data.

```
fig, ax = plt.subplots()
ax.fill(X_plot[:, 0], true_dens, fc='blue',
        alpha=0.4)
plt.show()
```

Now, if the data that we just simulated and visualized had normal or near-normal distribution, then we observe that the mean, the median, and the mode are very close to each other, and they even have the same value in case of the symmetrical unimodal distribution.

While the mean and the median can be calculated only for numerical variables, the mode can be calculated for both numerical and categorical variables. However, the mode is not commonly used with numerical variables as a measure of center. Nevertheless, it is fairly used with categorical variables as it makes more sense than the mean and the median.

Moreover, the mode is more robust to outliers than the mean as the outliers, by definition, are not frequently found in your data, and the mode is simply the most frequent value.

The figure that we have just plotted from our simulated data is called a *density* plot, which is a smooth estimation of the actual data frequency. We can plot the actual data frequency using another visualization plot called the *histogram*.

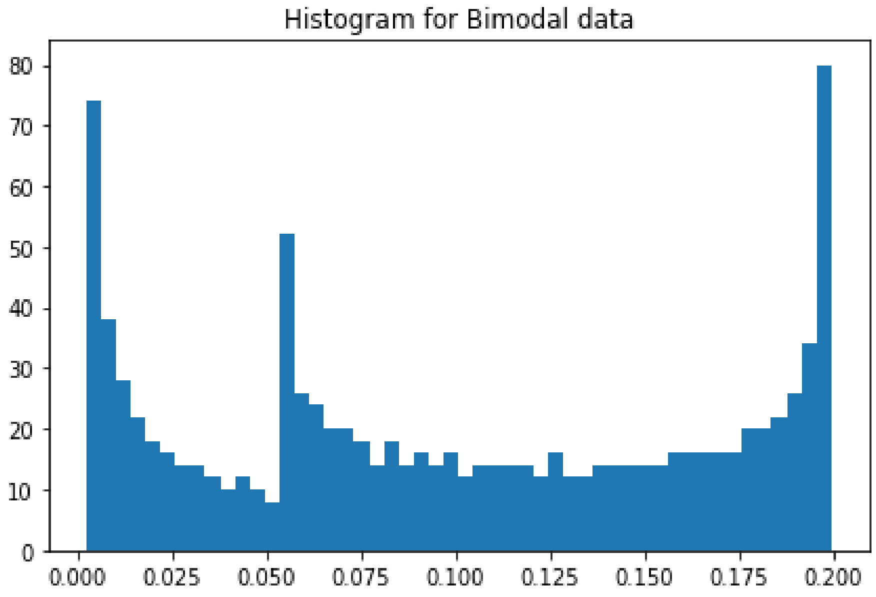
To do so, we will start by simulating the same data.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
```

```
N = 100
np.random.seed(1)
X_plot = np.linspace(-2, 8, 1000)[:, np.newaxis]
true_dens = (0.5 * norm(1, 1).pdf(X_plot[:, 0])
             + 0.5 * norm(5, 1).pdf(X_plot[:, 0]))
```

Now, to plot it, we use the *hist* function in Python Matplotlib.

```
fig, ax = plt.subplots()
freq_arr, bin_arr, _ = ax.hist(true_dens,bins=50)
ax.set_title(‘Histogram for Bimodal data’ )
plt.tight_layout()
plt.show()
```



5.2. Expected Value

Now, let us discuss the expected value, which is a purely statistical concept that is crucial for any data scientist.

Imagine you have a coin, and you flip it 10 times. Imagine also that you got tails 9 times. Would you say that the probability of getting tails for this coin flip is 90 percent? The answer is, of course, *No.* You would still say that it is 50 percent. Why? Because the *expected value* for each trial is 50 percent. So, if you flip this same coin 100,000 times or so, you will get roughly 50,000 heads and 50,000 tails.

We say that the expected value is the average value of a *random* variable over *a large* number of experiments.

When we have more than one variable, the math starts to get complicated. But we can simplify the expected value to be *the* *probability-weighted sum of values*, or in other words, we can write it down as,

$$E(x) = \sum_{i=1}^{i=n} (x_1 * p_1, x_2 * p_2, x_3 * p_3, \ldots, x_n * p_n)$$

Where $E(x)$ is the expected value, as we have n variables, and each one of them has a $p(x_i)$ probability.

Let us see an example using Python. In this exercise, we want to find what is the expected value of a 6-headed dice. Thus, we simulate this by creating an array that has six values from 1 to 6.

```
array = [ 1.0, 2.0, 3.0,4.0, 5.0, 6.0 ]
```

Then, calculate the probability of each element as $p(x_i) = \frac{1}{len(x)}$, because according to the law of large numbers, which is a foundational law in probability, if we repeat an experiment so many times, the arithmetic means of the values will always converge to the expected value.

```
probability = 1/ len(array)
```

Finally, we calculate the expected value using the equation stated above. We do so in Python by having a loop that calculates the accumulated sum of the multiplication between the number and its probability.

```
exp = 0
for i in range(0, len(array)):
    exp += (array[i] * probability)
print(exp)
3.5
```

5.3. Scatter plots

Now, after understanding the two important concepts of the mode and the expected value, let us explore another technique for visualization called the scatter plot.

You might be wondering why we did not include this in chapter 4 when we focused on data visualization. The reason is that this visualization technique is very important to statistical analysis that we preferred to discuss it separately.

The scatter plot is more than a visualization technique as it can be extended to be an analysis process for comparing two datasets against each other to know if there is a relationship or not. This way, we can treat the scatter plot as the tool to visualize this relationship by plotting and scattering the data points on a graph.

Moreover, a scatter plot can help us to quickly describe the association between our variables. This description can include the direction, the strength, and the form of the association, in addition to detecting any outliers. We can define these description elements as follows:

- **Direction:** Is the association negative or positive?
- **Form:** Is the association non-linear or linear?
- **Strength:** Is the association weak or strong?
- **Outliers:** Are there any data points that do not follow the data scattered distribution?

These four elements are keys to understand your data better, and hence be able to perform accurate statistical analysis.

Now, let us see how we can write a Python code to simulate a scatter plot. Let us say you want to buy a new house. You go to a real-estate agency and look up a few houses for sale. You have now collected a lot of features/variables that can help you decide. But you want to analyze the relationship between the house price and its area in meters squared.

First, you import the needed libraries, which are just Matplotlib this time.

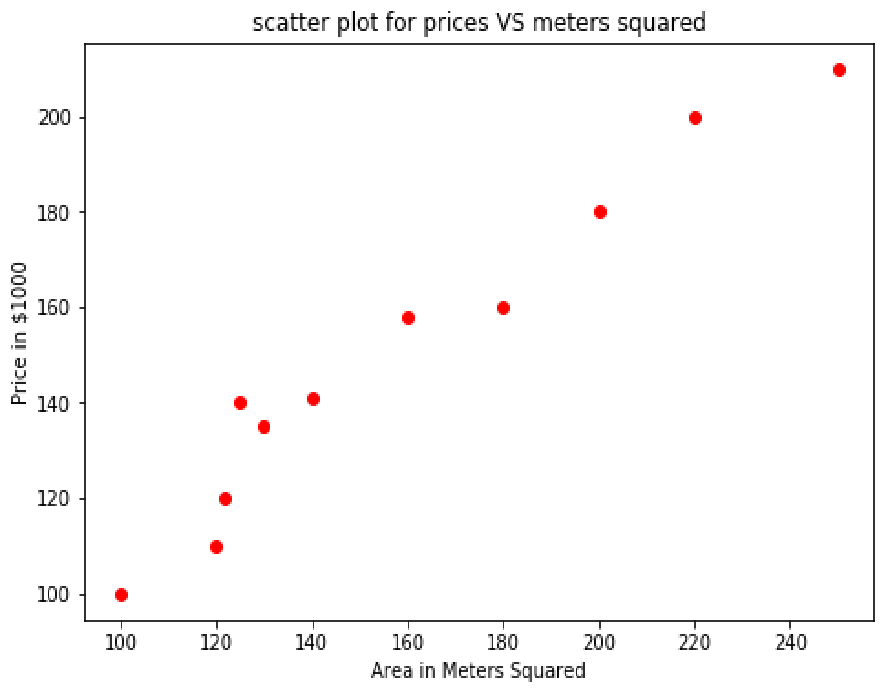
```
import matplotlib.pyplot as plt
```

Then, you enter the data that you have collected into two arrays, one for the prices and the other one for the area.

```
price = [100,110,120,140,135,141,158,160,180,200,210]
meters_squared = [100,120,122,125,130,140,160,180,200,220,
250]
```

Now, you plot it using the scatter plot technique that you have just learned.

```
fig=plt.figure()
ax=fig.add_axes([0,0,1,1])
ax.scatter(meters_squared, price, color='r')
ax.set_xlabel('Area in Meters Squared')
ax.set_ylabel('Price in $1000')
ax.set_title('scatter plot for prices VS meters squared')
plt.show()
```



After visualizing the scatter plot, you can come up with the following findings:

- The *direction* of the association is positive, which means that the prices increase when the area increases, and vice-versa.
- The *form* of the association is linear, as it can be modeled with a simple linear line equation.
- The *strength* of the association is strong.
- There is no presence of any outliers.

As we can see, with this simple exercise, and without calculating any mathematical or statistical functions, we can come up with important findings by simply visualizing the scatter plot and analyzing it visually.

5.4. Correlation

Now, let us go back to some more math concepts. This time, we want to focus on correlation, which is a statistical method used to show whether pairs of variables are related or not, and if so, how strongly. For example, in the house scenario that we just tackled, we came up with a finding from the scatter plot, which is that the prices and the area are strongly positively linearly correlated. We came up with this finding visually, with no calculations. However, we need more than that. We need to know *how strongly* they are correlated. We need to come up with a value that we can use for future calculations.

The correlation technique can also be used in case of more than two variables, as it is calculated between each pair of variables, so we get by the end a correlation matrix.

Without diving into the math behind the correlation, as it is quite cumbersome and out of the scope of this book, we need only to know that the main result of a correlation between a pair of variables is the *correlation coefficient*, which is a number that varies from −1 to +1. If our correlation coefficient is closer to −1, then this means that the two variables are strongly negatively correlated, and vice versa. Also, the closer the correlation coefficient is to 0 from either side, the weaker the relationship between the variables.

Another way of reporting the correlation is by squaring the correlation coefficients, which make them easier to understand. Usually, the correlation coefficients are denoted as r, and their squares are denoted as r^2.

If we have $r = 0.3$ between two variables, then this means that 9 percent of the variation is related to $0.3^2 = 0.09$. Therefore, it is easier to interpret the square of the correlation coefficient than the correlation coefficient themselves.

Before we walk through an exercise on correlation analysis, we need to highlight three notes:

1. The correlation technique can only be used with numerical data and not categorical data.
2. **Correlation does not imply causation**. This means that even if we find a positive or negative correlation between two variables, this tells us nothing about if one of them is the cause to the other or not. It simply means that there is a relationship between these two variables, and that is it.
3. The correlation coefficient can be calculated in different ways. However, the one that we are discussing here is called *Pearson's coefficient.*

Now, let us see how to use correlation analysis in practice using Python. We will be using a real-world dataset for the house prices example we discussed and simulated earlier.

We start by importing the needed libraries.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
fromscipy.stats import norm
```

Then, we load our dataset.

```
df = pd.read_csv(“House Price.csv”)
df.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub

5 rows × 81 columns

As we can see, our dataset is far more complex than the example we simulated, with 81 features.

Now, let us explore the price feature, which we care about the most, using *describe* function in Pandas.

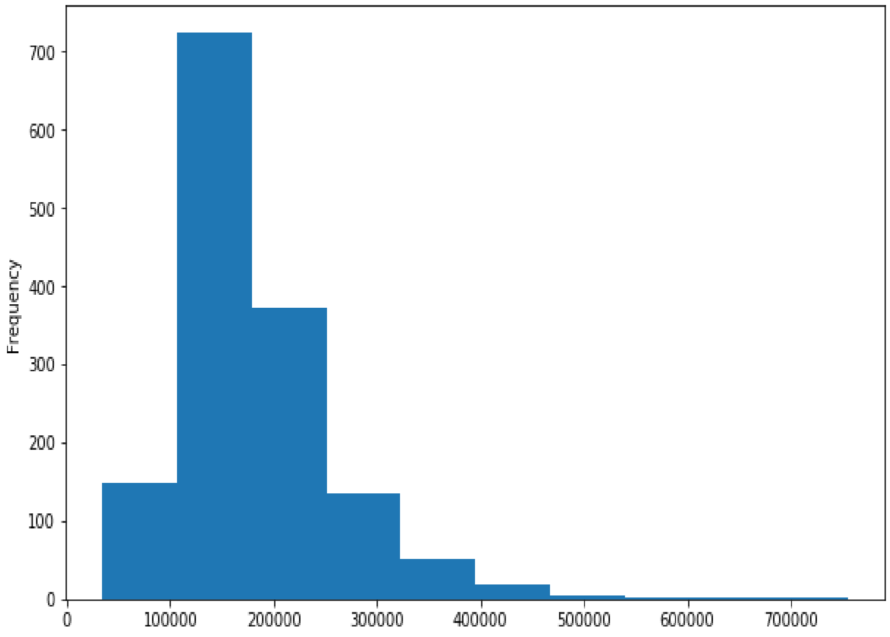
```
df.describe()
```

	0
count	1460.000000
mean	180921.195890
std	79442.502883
min	34900.000000
25%	129975.000000
50%	163000.000000
75%	214000.000000
max	755000.000000

As we can see, the minimum price for a house is 34,900, while the mean is 18,091, and the maximum is 755,000. We can also observe that we have 1460 houses in our dataset.

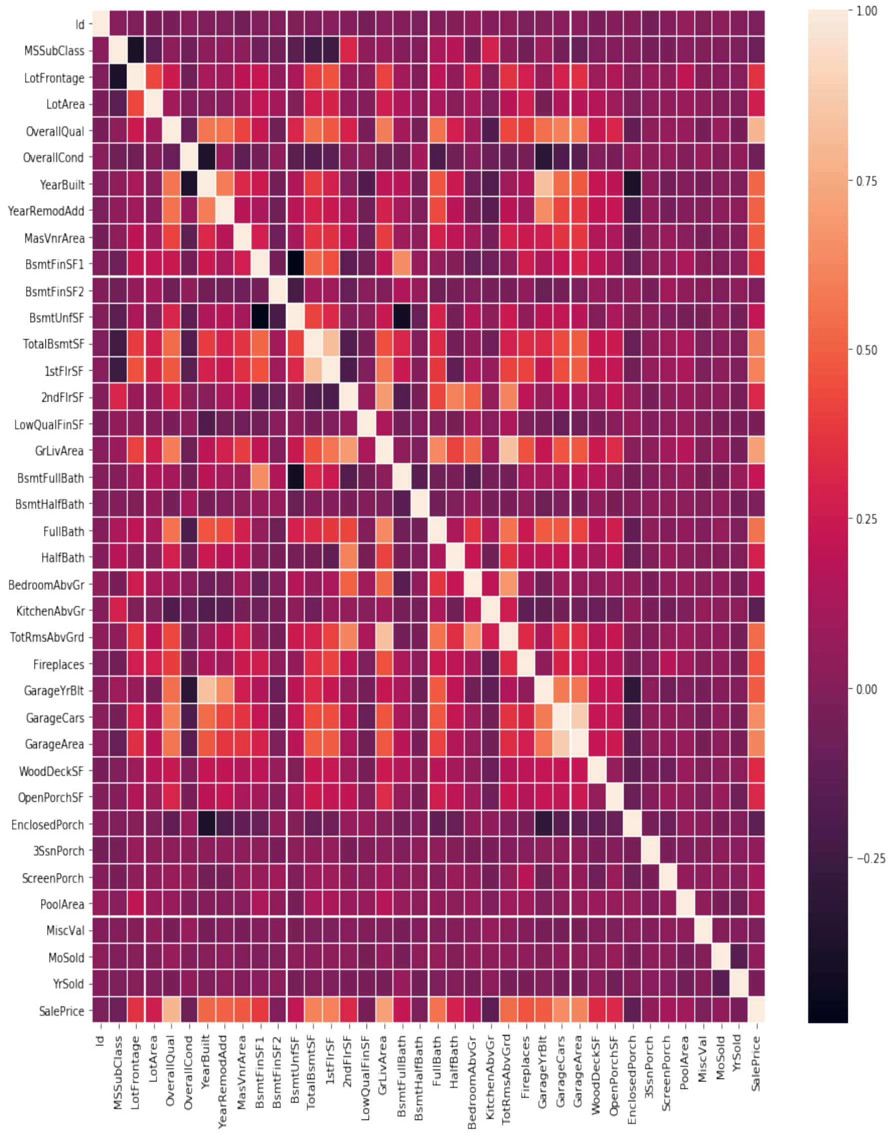
Now, let us plot the distribution of the prices using the histogram method we learned about earlier.

```
plt.figure(figsize = (10, 6))
df['SalePrice'].plot(kind ="hist")
```



Finally, let us find the correlation matrix, which can be done easily in two steps. First, we use Pandas' function called *corr* to find the correlation between each pair of variables. Then, we plot it using Seaborn's heatmap.

```
corrmat = df.corr()
f, ax = plt.subplots(figsize =(12, 10))
sns.heatmap(corrmat, ax = ax, linewidths = 0.1) 50
```



The diagonal of this matrix represents the correlation between the variable and itself, which is obviously 1. We search through this graph to find which variable has the highest correlation with the *SalePrice* variable, other than the SalePrice itself, and we see that it is the Overall Quality feature.

5.5. Categorical and Numerical Data

Till now, we have mentioned the terms *categorical data* and *numerical data* a few times, but we have not defined them in an elaborative way.

Data can be split into two major categories, which are Numerical data (quantitative data) and Categorical data (qualitative data).

We can say that the data are categorical if the different values that the data can have cannot be used in mathematical operations. Categorical data can be split even further into ordinal (ordered) data and nominal (unordered) data. The rating of a movie is a good example of ordinal categorical data, while blood type is a good example of nominal data.

On the other hand, numerical data can be used in mathematical operations. Numerical data can be split even further into discrete numerical data and continuous numerical data. Discrete numerical data can only have one of a pre-defined set of values. Examples of that can be the number of bedrooms in a house. Continuous data can have any value from negative infinity to infinity. Examples of that can be the speed of a car. But of course, based on the nature of the variable in the data, even the continuous variables should be restricted by a range.

5.6. Visualizing Multiple Variables

So far, we have reviewed how we can visualize either 1-dimensional or 2-dimensional datasets, as we were only plotting one or two variables. Now, we want to see how we can expand the visualization concepts we learned together to plot higher-dimensional datasets.

We start with three variables or 3D datasets. We can visualize them using the scatter plot technique we discussed earlier with the introduction of a third dimension.

To do so, we import our needed libraries, which are Matplotlib and another library for 3D plotting.

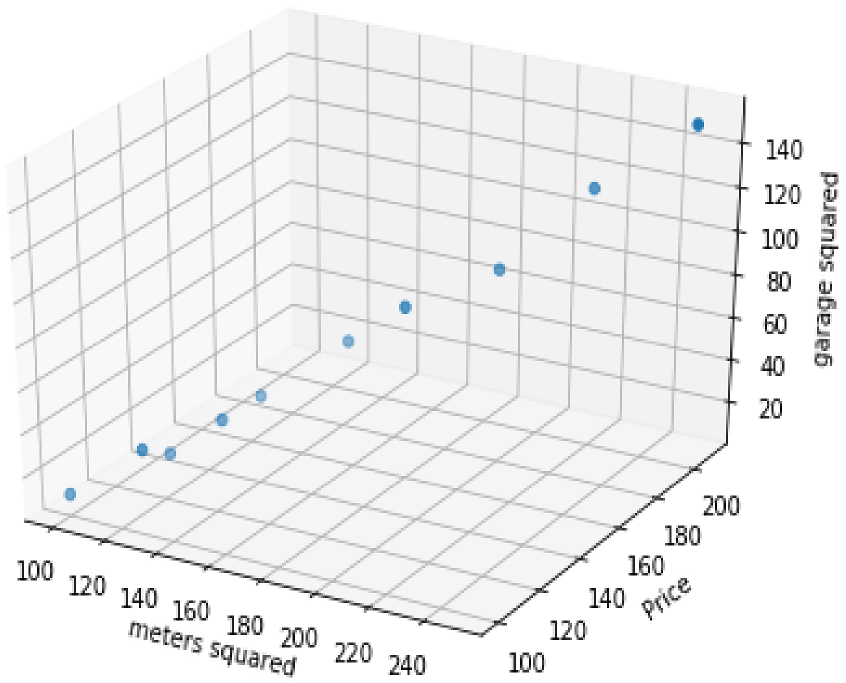
```
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D
```

Then, we use the same price and area variables that we discssed before in addition to a thrid variable representing the area of the garage.

```
price = [100,110,120,140,135,141,158,160,180,200,210]
meters_squared = [100,120,122,125,130,140,160,180,200,220,
250]
garage_squared = [10,30,22,25,30,40,60,80,90,120,150]
```

Finally, we use the same function to plot the 3D graph but with an addition of defining the axis as 3D.

```
fig = pyplot.figure()
ax = Axes3D(fig)
ax.scatter(meters_squared, price, garage_squared)
ax.set_xlabel(‘meters squared’)
ax.set_ylabel(‘Price’)
ax.set_zlabel(‘garage squared’)
pyplot.show()
```

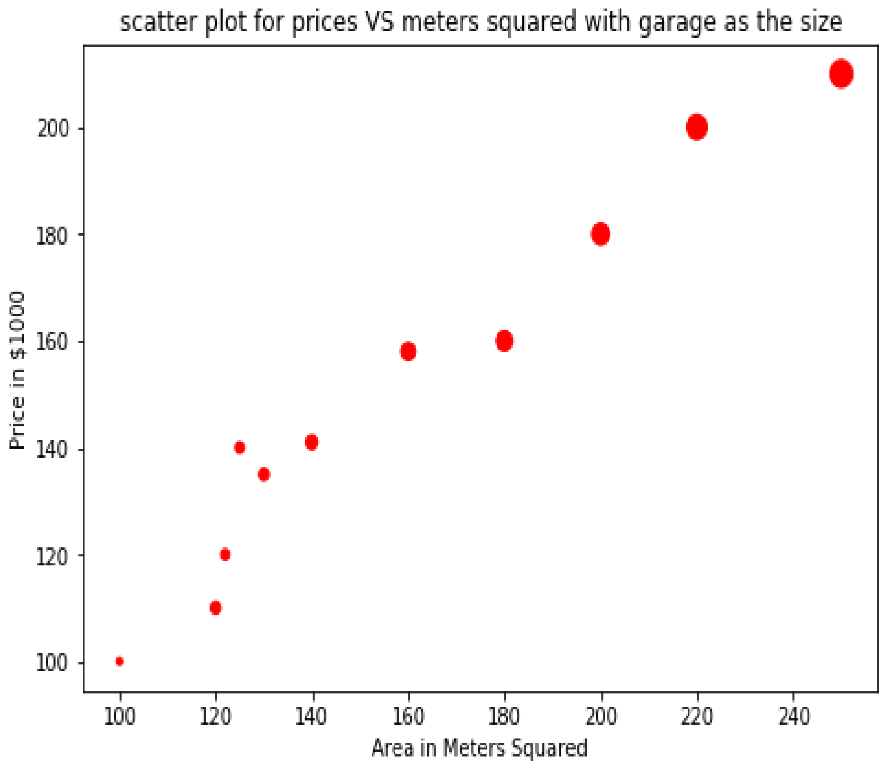


As we can see, we can visualize the data correctly.

Another way to do it is to use the same 2D graph that we plotted earlier but with a small tweak. We can add the third variable implicitly as it would be represented using colors in case of a categorical variable or size in case of a numerical variable.

We can do the color trick as follows:

```
fig=plt.figure()
ax=fig.add_axes([0,0,1,1])
ax.scatter(meters_squared, price,garage_squared, color='r')
ax.set_xlabel('Area in Meters Squared')
ax.set_ylabel('Price in $1000')
ax.set_title('scatter plot for prices VS meters squared
with garage as the size')
plt.show()
```



Finally, what if we want to visualize more than three variables? We cannot, as humans, see objects in more than three dimensions. However, we can visualize more than three variables by applying the size trick again and again.

For example, if we want to visualize 4D data, either we can use the 3D plot and have the fourth variable represented in size, or we can use the 2D plot and have the third and fourth variables represented in size and color. From there, you can extend this further and apply it to higher-dimensional data.

6

Statistical Tests and ANOVA

In this chapter, we will examine how you can perform statistical tests, both theoretically and practically. We will first start with hypothesis tests and then go to statistical significance and P-value. Following that, we will explore the analysis of variance test or *ANOVA* for short. After that, we will see what is meant by the chi-square test and how to use it. Finally, we will discuss Fisher's exact test.

As usual, we will explain everything both conceptually, and using hands-on exercises in Python.

6.1. Hypothesis Tests

So far, we have been focused on data exploration and data visualization for statistical analysis. But to close the cycle, we need to focus on the essence of statistics, which is to test a hypothesis. For example, you might have a hypothesis that taller people are smarter. Therefore, you run an experiment by conducting an IQ test to a few people with varying heights. Although your hypothesis might have been verified, if you cannot repeat the experiment and get similar results, no one will take your hypothesis and findings seriously.

Therefore, to define a hypothesis, it is an educated guess that should be testable and falsifiable, either by observation or experiment. Some examples of a hypothesis can be:

- A new way of designing bottles.
- A new model for educational systems.
- A new chemical ingredient for a medicine that is cheaper and more effective.

You can come up with your own hypothesis about *anything* if it can be tested.

After you come up with your hypothesis as just an idea, you need to write it down in a formal statement. If we want to formulate our examples above, they can be like this:

- If I (increase the diameter of the bottle and decrease its height), then (the people will buy it more).
- If I (change the educational system, so it is more oriented toward practical experience), then (students will be more prepared for the market).
- If I (add this amount of this chemical ingredient), then (the medicine will be more effective and cheaper).

So, we can define a good hypothesis statement as the one that satisfies the following criteria:

- Has an “if” and “then” statements.
- Can be tested by survey or experiment.
- The independent and dependent variables are included.
- Rely on prior research information.

Now, the next step after formulating your hypothesis is to test it, of course. These hypothesis tests are very important for two main reasons. First, we would know if our hypothesis is correct or not. Second, we would know the probability of getting this result by chance.

The first step to perform a hypothesis test is to know what your **null hypothesis** is. A null hypothesis is basically the accepted fact that you want to test against. Some examples of a null hypothesis can be:

- Antibiotics are the best way to fight bacteria.
- Studying hard is always a way to get high grades.
- The human body is 71 percent water.

From these examples, we can see that the last one is not something that you can reject (i.e., not nullifiable), so you cannot test against. On the other hand, the first two hypotheses can be rejected by some people, and thus, you can test against.

Before we dig into complex tests such as ANOVA and chi-square, let us discuss a simple test called the one-sample z test. Although this test is rarely used, it will help you understand how to perform hypothesis testing.

Your university's professor of physics claims that his students are above average intelligence. He takes a random sample of 30 students' IQ scores and finds out that they have a mean score of 112. So, would you say that there is enough evidence to support the professor's claim, given that you know that the mean IQ score of all students in the university is 100 with a standard deviation of 15?

Let us solve this problem step-by-step:

1. **State the Null hypothesis:** The population mean which is 100, or in other words, the accepted fact. Therefore: $H_0: \mu = 10$
2. **State the alternate hypothesis:** Your professor's claim that his students have above average IQ. Therefore, $H_1: \mu > 100$
3. **State the alpha level:** which is the probability of choosing wrong when the null hypothesis is correct. It is usually equal to 0.05.
4. **Find the rejection region:** which is the interval that leads us to reject the null hypothesis H_0. It can be found from a pre-defined table called the z-table, which uses the alpha stated above. In our example, it is 1.645.
5. **Perform your z-test:** It can be done using this formula $Z = \frac{\bar{x}-\mu_0}{\sigma/\sqrt{n}}$. If we apply it to our example, then it would be
$$Z = \frac{112-100}{\frac{15}{\sqrt{30}}} = 4.56$$
6. Finally, if the result we get in step 5 is larger than the rejection region found in step 4, then we reject the null hypothesis. Otherwise, you cannot reject the null hypothesis. In our example, 4.56>1.645. Thus, we can reject the null hypothesis.

6.2. Statistical Significance and P-value

Now, let us understand the concepts of statistical significance and p-value. In the previous section, we discussed statistical tests, and in the example, we discussed z-tests. We said in step 3 *state the alpha level*. Basically, the alpha level is also called the statistical significance or statistical level, and it is the probability of making the wrong decision when the null hypothesis H_0 is true.

Before we dive deeper, we need to know that there are two types of errors as follows:

1. **Type I error:** When the null hypothesis is true, we support the alternate hypothesis.
2. **Type II error:** When the alternate hypothesis is true, we do not support the alternate hypothesis

In the example of the Physics professor's hypothesis, the null hypothesis is that the average IQ of his students is not above the average of all students, and the alternate hypothesis is that the average IQ of his students is above the average of all students. If the truth is that the average IQ of his students is not above the average of all students and you decided otherwise, then this is a type I error, and we support the alternate hypothesis. If we did the opposite, then this is a type II error.

To link this with the significance level, we say that the significance level is the probability of a type I error. We can look at it also as the probability when you reject the null hypothesis. For type II error, we have another term called beta.

If you remember, we said in the previous section that we usually set the significance level at 0.05. Why is that?

We want to make σ, the area of the probability of making a type I error, as small as possible. Therefore, if we set σ to be 15 percent, then there is a huge chance that we reject the null hypothesis incorrectly. You might ask now why won't we set it as 1 percent or less, why 5 percent then?

The reason is that if you have a tiny area for a very small σ, then there is a huge chance of **not** rejecting the null hypothesis, while you should have. This will lead to a type II error.

Therefore, there is a trade-off in the choice of the alpha level, and 5 percent is usually the best value to balance the two types of errors.

Now that we know what we mean when we talk about the significance level and types of error, let us focus on how we would evaluate the test.

We use the p-value, which is the evidence against a null hypothesis, to support or reject the null hypothesis. When we have a smaller p-value, then we have stronger evidence about rejecting the null hypothesis. It is usually expressed as either a decimal number or a percentage. For example, if we have a p-value of 3 percent, then this means that there is a 3 percent chance that your results could be random. While on the other hand, a p-value of 80 percent means that there is a probability of 90 percent that our results were random and have nothing to do with your experiment. Thus, we say that we hope to get the smallest p-value possible.

So, when you perform a hypothesis test, you compare σ that you choose with the p-value you got. While we cannot control the p-value as it is a part of the result, we control the alpha level by subtracting the confidence level from 100 percent. In the next chapter, we will focus entirely on the confidence level. But for now, we need to know that it just means how confident we want to be about our results.

Let us say we want a confidence interval of 97 percent. Then, our $\sigma = 3\ percent$. The next step is to compare this 3 percent with the p-value we get from our experiment as follows:

- **A small p (< 0.03)**: We reject the null hypothesis.
- **A large p (> 0.03):** We cannot reject the null hypothesis, as the alternate hypothesis is weak.

The last thing we need to discuss here is what if you do not have an alpha level? In this case, we usually follow these thresholds agreed upon in the scientific community:

- **If p> 10 %:** Not significant.
- **If p < 10 %:** Marginally significant.
- **If p < 5 %:** Significant.
- **If p < 1 %:** Highly significant.

6.3. ANOVA

Let us now discuss one of the most important hypothesis tests called ANOVA, which is a way to figure out if our results are significant or not. This is translated to either accepting the alternate hypothesis or rejecting the null hypothesis.

There are two main categories of ANOVA tests, which are one-way and two-way tests, where the *way* refers to the number of impendent variables in our analysis. Therefore, we can say that the one-way ANOVA test has one independent variable, while the two-way ANOVA test has two independent variables.

For the one-way ANOVA, we compare the two means averages of two independent groups. We define the null hypothesis as that the two averages are equal.

Although one-way ANOVA is a very handy test and can help us gain some insights, such as if at least two groups were different from each other, it has some limitations. One of the critical limitations for one-way ANOVA is that it cannot tell us which of these groups are different. To do so, we need to perform more complex tests that are out of the scope of this book.

For the two-way ANOVA, we have two independent variables, and we usually use it when we have two nominal variables and one quantitative variable. There are some assumptions about two-way ANOVA that we need to keep in mind:

- All the groups' samples should be equal in size.
- Population mush has equal variances.
- We should have a normal distribution for the population.
- We must guarantee that our samples are independent.
- Residuals errors are normally distributed.

The steps to perform an ANOVA test are as follows:

1. Calculate the mean for each group.
2. Calculate the global overall mean.
3. Calculate the *variation within a group*, which is the total difference between each group member's value and its group mean.
4. Calculate the *variation between the groups,* which is the total difference between each group average and the global overall average.
5. Calculate the F statistic, which is the ratio between the variation within and between groups.

Now, enough with the theoretical part. Let us see how we can use Python to perform one-way and two-way ANOVA tests. Let us start with the one-way ANOVA test. The first step is to import the needed libraries as usual.

```
import pandas as pd
import scipy.stats as stats
import statsmodels.api as sm
from statsmodels.formula.api import ols
%matplotlib inline
```

Then, we load our dataset.

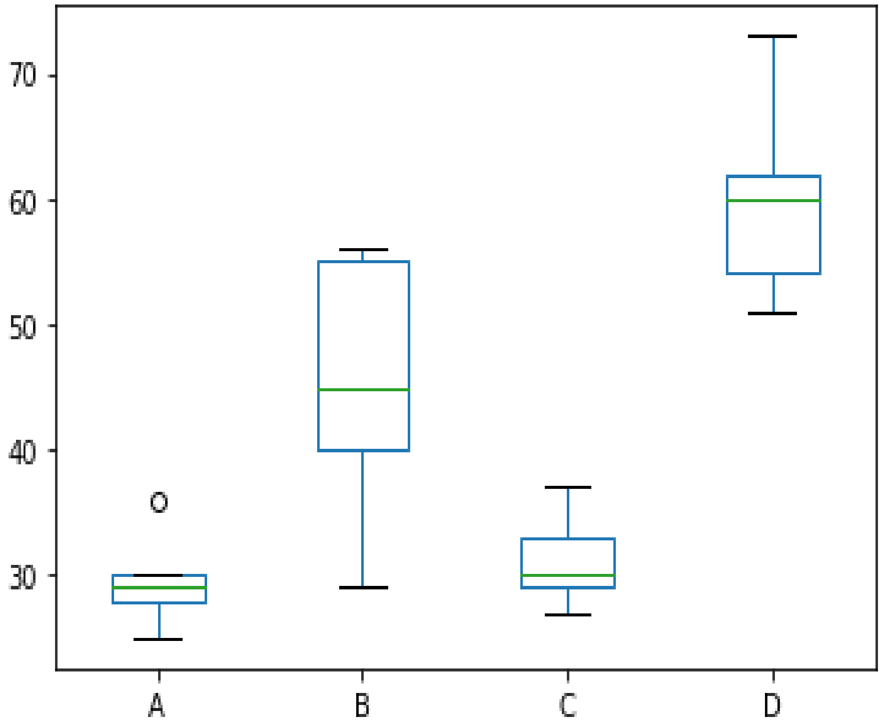
```
df = pd.read_csv("https://reneshbedre.github.io/assets/
posts/anova/onewayanova.txt", sep="\t")
df.head()
```

	A	B	C	D
0	25	45	30	54
1	30	55	29	60
2	28	29	33	51
3	36	56	37	62
4	29	40	27	73

As we can see, we have four groups for the ANOVA test.

The next step is to visualize the data using a box plot.

```
df.boxplot(column=['A', 'B', 'C', 'D'], grid=False)
```



After that, we use the SciPy function called f_oneway to get the p-value and the f-value.

```
f_value, p_value = stats.f_oneway(df['A'], df['B'],
df['C'], df['D'])
print(f_value, p_value)
17.492810457516338 2.639241146210922e-05
```

Since the p-value is very small, we can reject the null hypothesis and continue with our analysis. Also, we can conclude that there are significant differences between the variables.

Next, we melt our DataFrame, so it has the shape suitable for our ANOVA test.

```
df_melt = pd.melt(df.reset_index(), id_vars=[‘index’],
value_vars=[‘A’, ‘B’, ‘C’, ‘D’])
df_melt.head()
```

	index	variable	value
0	0	A	25
1	1	A	30
2	2	A	28
3	3	A	36
4	4	A	29

Following that, we fit an Ordinary Least Square (OLS) model, which we will discuss in detail in chapter 8 on our variable column.

```
model = ols(‘value ~ C(variable)’, data=df_melt).fit()
```

Finally, we get the results of the ANOVA test by calling the Statsmodel function.

```
anova_table = sm.stats.anova_lm(model, typ=2)
anova_table
```

sum_sq	df	F	PR(>F)	
C(variable)	3010.95	3.0	17.49281	0.000026
Residual	918.00	16.0	NaN	NaN

From this analysis, we know that the differences in the variables are statistically significant, but as we discussed earlier, we cannot know which of them are different from each other.

Now, let us work on a two-way ANOVA test project. We start by importing the libraries.

```
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
from statsmodels.formula.api import ols
```

Then, we load our dataset.

```
df_two = pd.read_csv("https://reneshbedre.github.io
/assets/posts/anova/twowayanova.txt", sep="\t")
df_two.head()
```

	Genotype	1_year	2_year	3_year
0	A	1.53	4.08	6.69
1	A	1.83	3.84	5.97
2	A	1.38	3.96	6.33
3	B	3.60	5.70	8.55
4	B	2.94	5.07	7.95

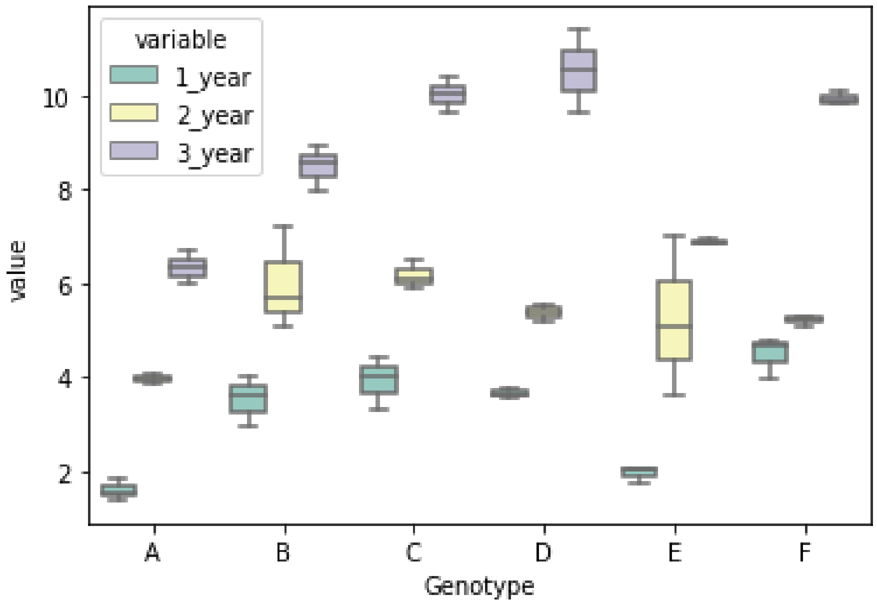
We melt the DataFrame as before.

```
df_two_melt = pd.melt(df_two, id_vars=['Genotype'], value_
vars=['1_year', '2_year', '3_year'])
df_two_melt.head() data = np.concatenate((s, c, high, low))
df = pd.DataFrame(data)
```

	Genotype	variable	value
0	A	1_year	1.53
1	A	1_year	1.83
2	A	1_year	1.38
3	B	1_year	3.60
4	B	1_year	2.94

Then, we plot our melted dataset as a box plot.

```
sns.boxplot(x="Genotype", y="value", hue="variable",
data=df_two_melt, palette="Set3")
```



Again, we fit our OLS model.

```
model = ols('value ~ C(Genotype) + C(variable) +
C(Genotype):C(variable)', data=df_two_melt).fit()
```

Finally, we calculate our two-way ANOVA model.

```
anova_table = sm.stats.anova_lm(model, typ=2)
anova_table
```

sum_sq	df	F	PR(>F)	
notype)	58.551733	5.0	32.748581	1.931655e-12
C(variable)	278.925633	2.0	390.014868	4.006243e-25
C(Genotype):C(variable)	17.122967	10.0	4.788525	2.230094e-04
Residual	12.873000	36.0	NaN	NaN

As we can see, the p-value is statistically significant for genotype, variable, and interaction. We can say that the yield outcome is significantly affected by both the genotype and the years. We can also say that the interaction between the years and the genotype highly affects the yield.

6.4. Chi-Square Test

Let us now discuss another significance test called chi-square. Two types of tests use the same distribution and statistic for different use cases:

- **Test for the goodness of fit:** We use it to determine whether a sample data agrees with a population or not.
- **Independence test:** We use it to compare two variables in a contingency table, which contains information about the frequency distribution of the variables, to know if they are related or not. It is used with categorical variables to test if they differ from each other or not.

The formula for the chi-square test is as follows:

$$x_c^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Where O is the observed value, E is the expected value, and c is the degree of freedom.

As we said, the chi-square statistic is usually used to test if there is a relationship between categorical variables or not.

If we get a very small value out of this test, then this means there is a relationship, and your observed data matches your expected data. On the other hand, if we get a very large value out of this test, then this means there is no relationship.

Now, let us see how we would perform the chi-square test for goodness using Python.

The first step is to import our libraries.

```
import numpy as np
import pandas as pd
importscipy.stats as stats
```

Then, we simulate our dataset, which is the total population, and our sample data.

```
all_population = pd.DataFrame([“white”]*100000 +
[“hispanic”]*60000 +\
       [“black”]*50000 + [“asian”]*15000 + [“other”]*35000)
our_sample = pd.DataFrame([“white”]*600 + [“hispanic”]*300 + \
[“black”]*250 +[“asian”]*75 + [“other”]*150)
```

After that, we convert the DataFrame into a cross table for ease of calculations.

```
all_population_table = pd.crosstab(index=all_population[0],
columns=»count»)
print(all_population_table)
col_0      count
0
asian      15000
black      50000
hispanic   60000
other      35000
white     100000
```

```
our_sample_table = pd.crosstab(index=our_sample[0],
columns=»count»)
print(our_sample_table)
col_0     count
0
asian        75
black       250
hispanic    300
other       150
white       600
```

Following that, we get the ratio for each group in our population.

```
all_population_ratios = all_population_table/len(all_
population)
print(all_population_ratios)
col_0        count
0
asian     0.057692
black     0.192308
hispanic  0.230769
other     0.134615
white     0.384615
```

Then, we get the expected counts.

```
expected_counts = all_population_ratios * len(our_sample)
print(expected_counts)
col_0          count
0
asian      79.326923
black     264.423077
hispanic  317.307692
other     185.096154
white     528.846154
```

After that, we apply the chi-square formula.

```
chi_squared_stat = (((our_sample_table-expected_
counts)**2)/expected_counts).sum()
print(chi_squared_stat)
18.194805
```

Now, to know if our chi-square statistic is valid or not, we need to find its p-value. We need to first find the critical value with a confidence interval of 95 percent.

```
critical_value = stats.chi2.ppf(q = 0.95, df = 4)
print(critical_value)
9.487729036781154
```

```
p_value = 1 - stats.chi2.cdf(x=chi_squared_stat,df=4)
print(p_value)
0.00113047
```

As the chi-square value is more than the critical value, we can reject the null hypothesis that the two distributions are the same.

Finally, we can summarize all this into one line using the StatsModel library.

```
stats.chisquare(f_obs= our_sample_table ,f_exp= expected_
counts)
Power_divergenceResult(statistic=array([18.19480519]),
pvalue=array([0.00113047]))
```

6.5. Fisher's Exact Test

The last test we are going to tackle is the Fisher's exact test, which is very similar to the chi-square test, as it is used to decide if there is a significant relationship between two categorical variables or not.

It is usually used when the sample sizes are small because it requires more computational power and time. It is also recommended to use it instead of the chi-square test if we have more than 20 percent of the contingency table with expected frequencies less than 5.

This test is called *exact* because while other tests such as the z-test and the chi-squared test calculate the p-value approximately, Fisher's exact test calculates the p-value exactly. Thus, it is also preferable to use this test instead of the chi-square test whenever possible.

The test uses the following permutation equation to determine the probability of all the possible contingency tables:

a	b	a + b
c	d	c + d
a + c	b + d	N = a + b + c + d

$$p = \frac{\binom{a+b}{a}\binom{c+d}{c}}{\binom{N}{a+c}}$$

Let us now see how we can perform Fisher's exact test using Python.

Imagine we are running a trial experiment on a new drug, and we want to know if the response has nothing to do with the gender or not. The first step is to import the libraries.

```
import numpy as np
import pandas as pd
importscipy
importscipy.special
```

Then, we simulate our dataset.

```
ar=np.array([[6.0, 3.0],[2.0, 12.0]])
df=pd.DataFrame(ar, columns=[“Responded”, “No Response”])
df.index=[“Males”, “Females”]
df.head()
```

	Responded	**No Response**
Males	6.0	3.0
Females	2.0	12.0

The first thing that might come to your mind is that there are indeed differences in the response to the treatment that is related to gender. However, we need to make sure of this using Fisher's exact test.

We add a new row that contains the sum of each column, and a new column that contains the sum of each row.

```
df2=df.copy()
df2.loc['Column_Total']= df2.sum(numeric_only=True, axis=0)
df2.loc[:,'Row_Total'] = df2.sum(numeric_only=True, axis=1)
df2.head()
```

	Responded	No Response	Row_Total
Males	6.0	3.0	9.0
Females	2.0	12.0	14.0
Column_Total	8.0	15.0	23.0

Before we perform Fisher's exact test, we might want to know the expected value for each cell. We do so by multiplying each cell total row with its total column and divide by the total.

```
total=df2.at["Column_Total", "Row_Total"]
expected_values=df2.copy()
for row in expected_values.index[0:-1]:
    for col in expected_values.columns[0:-1]:
current_expected= (((df2.at[row, "Row_Total"])*(df2.
at["Column_Total", col]))/total).round(2)
        expected_values.at[row,col]=float(current_expected)
expected_values.head()
```

	Responded	No Response	Row_Total
Males	3.13	5.87	9.0
Females	4.87	9.13	14.0
Column_Total	8.00	15.00	23.0

As we can see, more than 20 percent of the table has an expected value of less than 5. Therefore, we are entirely sure that Fisher's exact test is the most suitable significance test here.

Finally, we can perform the test with a single line of code.

```
oddsratio, pvalue = stats.fisher_exact(df)
oddsratio,pvalue
(12.0, 0.022760924631970555)
```

7

Confidence Interval

In this chapter, we will discuss what is meant by a confidence interval. Then, we will see how we would define the confidence interval for a population mean. Following that, we will do the same but for a population proportion. After that, we will understand the meaning of the confidence interval for the difference between two means and between two proportions. Finally, we will discuss the relation between t-distribution and the confidence intervals.

7.1. Confidence Interval Meaning

In the last chapter, we discussed the concept of statistical significance, and we mentioned the term *confidence interval* while doing so. We could define the confidence interval (CI) as being how much we are uncertain about a statistical test result on a sample if it was applied to the entire population. Another simpler definition is that we define the confidence interval as the range of values that we are sure about.

For example, we can select randomly 40 men and measure their heights. We found that the mean height is 175 cm, while we know that 20 cm is the standard deviation of men's heights. If we decided to go with a 95 percent confidence interval, we would have the result as 175 ± 6.2 cm, which means that 95 percent of the time, we will find people who are between 168.8 cm and 181.2 cm. We will see in the next section how we calculated this number.

7.2. Confidence Interval for a Population Mean

Following the example of the last section, we will now calculate the confidence interval range for the population mean by following these steps:

1. Choose a confidence interval you want. In our case, it is 95 percent.
2. Find the z-value from the z lookup table. It is 1.96 in our case.
3. Calculate the range using $\bar{X} \pm Z \frac{s}{\sqrt{n}}$ where $\bar{X}$ is the mean, z is our chosen z-value, s is the standard deviation, and n is the number of observations. Therefore, we would have $175 \pm \frac{20}{\sqrt{40}} = 175 \pm 6.2$

Now, let us see how to simulate a more complex exercise using Python.

We start by importing the needed libraries.

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

We then generate 100 thousand data points with a mean of 4 and a standard deviation of 2.82

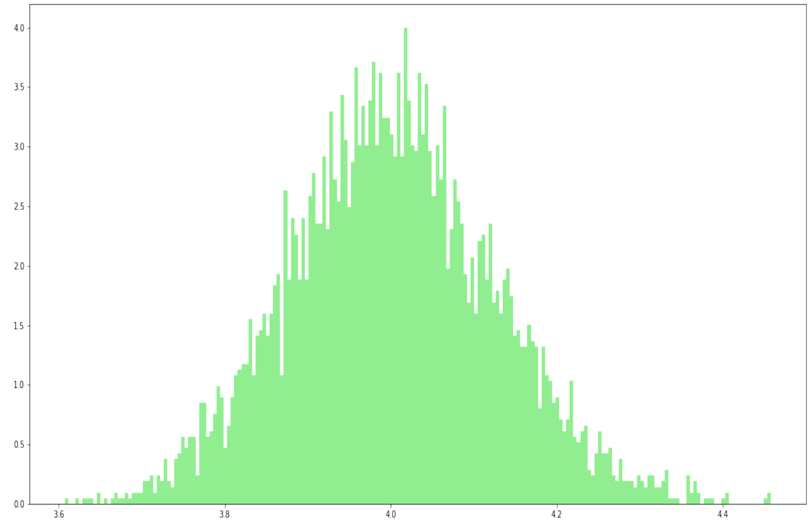
```
shape, scale = 2.0, 2.0
s = np.random.gamma(shape, scale, 100000)
mu = shape*scale
sigma = scale*np.sqrt(shape)
print(mu)
print(sigm)
4
2.82
```

After that, we create a mean sample array with a sample size of 500.

```
sample_mean = []
sample_size = 200
for j in range(0,5000):
random_sample = random.choices(s, k=sample_size)
sample_mean.append(sum(random_sample)/len(random_sample))
```

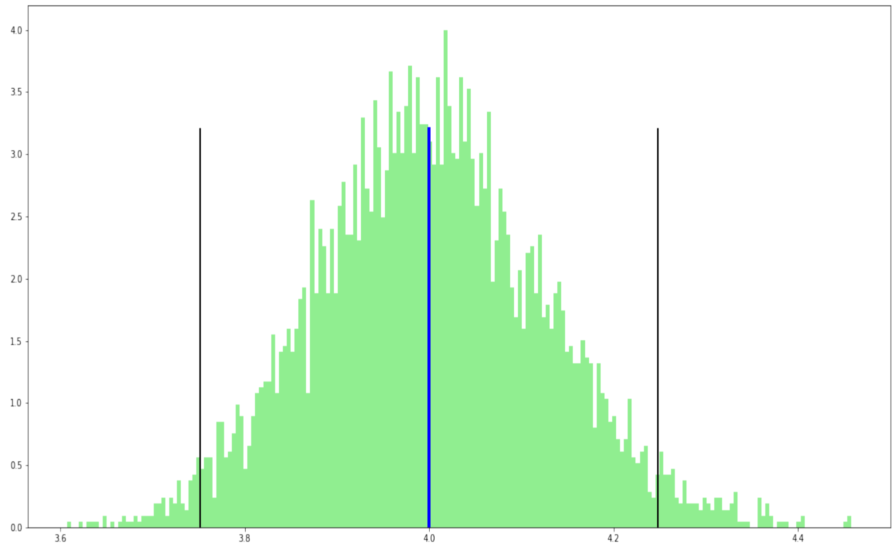
Then, we plot our data.

```
plt.figure(figsize=(20,10))
plt.hist(sample_mean, 200, density=True,
color='lightgreen')
plt.show()
```



Finally, we can find the confidence interval on the graph.

```
plt.figure(figsize=(20,10))
plt.hist(sample_mean, 200, density=True,
color='lightgreen')
plt.plot([mu,mu],[0, 3.2], 'k-', lw=4, color='blue')
plt.plot([mu-(1.96*sigma/np.sqrt(sample_size)),
mu-(1.96*sigma/np.sqrt(sample_size))],[0, 3.2], 'k-', lw=2,
color='black')
plt.plot([mu+(1.96*sigma/np.sqrt(sample_size)),
mu+(1.96*sigma/np.sqrt(sample_size))],[0, 3.2], 'k-', lw=2,
color='black')
plt.show()
```



As we see, the confidence interval is 4 ± 0.2479

```
patches, texts = plt.pie(sizes, colors=colors)
plt.legend(patches, labels, loc="best")
plt.axis('equal')
plt.tight_layout()
plt.show()
```

7.3. Confidence Interval for a Population Proportion

Now, let us discuss the confidence interval for a population proportion. But before that, we need to know what is meant by a population proportion first. It means the percentage of the population that has the same characteristic.

As an example, if we have 400 people in our population and 100 of them have brown eyes, then the percentage of the population with brown eyes is $p = 25\%$.

Going back again to the confidence interval definition, we can say it is calculated as follows:

$$Best\ Estimate \pm Margin\ of\ Error$$

Where the best estimate is the observed population, and the margin of error is the z-value for our chosen confidence interval.

We can also create a 95 percent confidence interval as follows:

$$Population\ Proportion \pm (z\_value * Standard\ Error)$$

Where the standard error can be calculated as:

$$Standard\ Error\ for\ population = \sqrt{\frac{population * (1 - population)}{number\ of\ observations}}$$

That is enough math. Let us implement it using Python.

The first step is to import our libraries.

```
import numpy as np
import statsmodels.stats.proportion as sm
```

Then, we will choose a confidence interval of 95 percent, so the z value is 1.96, according to the z-table. And we set our population proportion to be 90 percent, and we assume we have 600 observations.

```
z_value = 1.96
population_proportion = 0.9
mumber_of_observations = 600
```

Then, we calculate the standard error according to its equation.

```
standard_error = np.sqrt((population_proportion*(1-
population_proportion))/mumber_of_observations)
print(standard_error)
0.01224744871391589
```

Finally, we calculate the lower and the upper bound using the above equation.

```
lower_bound = population_proportion - (standard_error *z_
value)
upper_bound = population_proportion + (standard_error *z_
value)
lower_bound,upper_bound
(0.8759950005207249, 0.9240049994792752)
```

Let us repeat the same example but using StatsModels.

```
sm.proportion_confint(mumber_of_observations*
population_proportion,mumber_of_observations)
(0.8759954416182235, 0.9240045583817765)
```

7.4. Confidence Interval for the Difference Between Two Means

The confidence interval can also be defined for the difference between two means. If so, it indicates the range of values that difference between the means of our population is. Moreover, you can think about it as a test to know if two samples came from the same distribution or not. In the last chapter, when we applied the ANOVA one-way and two-way tests, this analysis was also conducted in StatsModels functions.

7.5. Confidence Interval for the Difference Between Two Proportions

Imagine you want to calculate the difference between the proportion of men and women who support the notion that the workweek should be only 3 days instead of 5.

What you want to estimate is the difference between the men's population and the women's population, $p_m - p_w$. You can do so by taking a random sample of each population and use the difference between the two sample proportions $\widehat{p_m} - \widehat{p_w}$ plus or minus the margin of error that we explained earlier.

The full equation is as follows:

$$\widehat{p_m} - \widehat{p_w} \pm z * \sqrt{\frac{\widehat{p_m}(1-\widehat{p_m})}{n_m} + \frac{\widehat{p_w}\,(1-\widehat{p_w})}{n_w}}$$

As we know from before, the z can be estimated from the z lookup table, which is equal to 1.96 in case of a 95 percent confidence interval.

Let us see how to apply this in Python.

We start with our libraries.

```
import numpy as np
```

We set our parameters.

```
p_m = 0.64
p_w = 0.43
n_m = 100
n_w = 120
z_value = 1.96
```

Then, we calculate the difference between the means.

```
difference = p_m - p_w
first_term = (p_m* (1-p_m))/n_m
second_term = (p_w*(1-p_w))/n_w
```

Finally, we calculate the lower and upper bounds using the given equation.

```
lower_bound = difference - (z_value*(np.sqrt(first_term +
second_term)))
upper_bound = difference + (z_value*(np.sqrt(first_term +
second_term)))
lower_bound,upper_bound
(0.0807811375998071, 0.33921886240019294)
```

We can interpret the results by saying with 95 percent confidence that a higher percentage of men than women are supporting the 3-day workweek, and the difference in their percentages is between 8 percent and 33.9 percent.

7.6. The t-Distribution and Confidence Intervals

Throughout this chapter, we have been assuming a normal distribution. But now, let us discuss another commonly found distribution called the t-distribution. While also known as the *Student's distribution*, the t-distribution is a probability distribution that has heavier tails than the normal distribution, which makes room for more extreme values to be found.

Throughout this chapter, we have been using the z-score and the z-table. However, you should know that to calculate the exact z-score, you need to know the *true* standard deviation, which is usually unknown. Therefore, we use the t-distribution as it can be defined as the continuous probability distribution of the z-score if we use the *estimated* standard deviation.

So, we just want to highlight that all the calculations we have done using StatsModels is based on the t-distribution as we used the estimated standard deviation all the time.

8

Regression Analysis

In this chapter, we will have a fundamental understanding of regression analysis, which is needed for any data scientist or machine learning engineer.

We will start the chapter by defining a regression problem and understand clearly what is meant by regression. Then, we will understand, in detail, linear regression analysis concepts and relate them to what we have learned so far. Following that, we will explore multiple regression analysis. After that, we will discuss the most common mistakes that people do while working on a regression problem. Finally, we will conclude the chapter with hands-on projects on both simple linear regression and multiple regression.

8.1. Regression Meaning

Imagine you want to know if the income of a person has anything to do with the area of his/her house. To test this hypothesis, you ask many of your friends about both their income and the area of their house, or you simply download a dataset from the internet that fits your use case. Then, you will simply try to plot the data and see if there is a clear linear relationship. This plotting is done using a mathematical equation or a model, which can then be used to predict the income of a person by just knowing the area of her/his house.

This equation is a simple one, which is as follows:

$$y = mx + b$$

In this equation, we can find the output *y* by multiplying the input *x* by the slope *m* and add this to the y-intercept *b*. The output is also called the response variable, as we are trying to evaluate or predict it, while the input can be called the predictor variables.

One goal for regression analysis is, as we have just seen, predict values—response variable. To achieve that, we need to learn how we can do so.

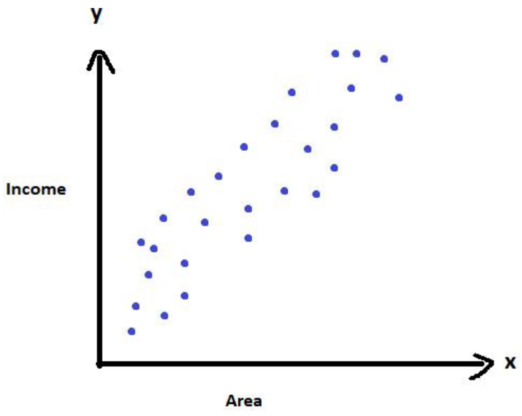
We start by having a dataset containing both the input and the output, and our goal is to find the slope *b*. Thus, when we have only input data, we can multiply them by this slope, and we get the output with great accuracy. This is a **learning** process.

8.2. Simple Linear Regression

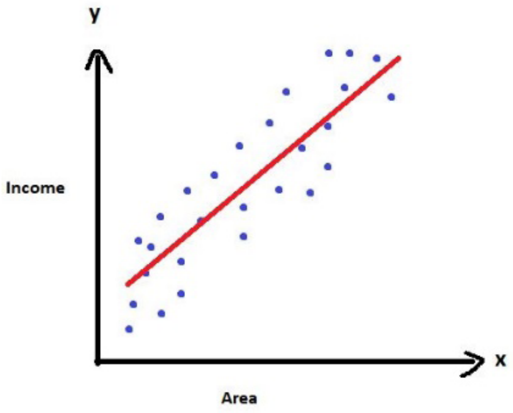
For simple linear regression analysis, we want to **predict** the output by **training** a machine learning model on both the input and the output so we can get the slope and use both the new input and the learned slope to predict the new output.

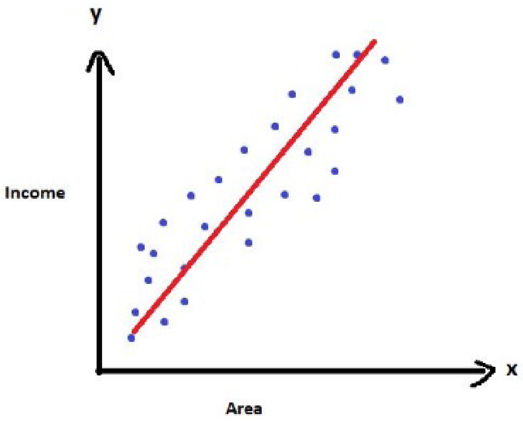
$$y = mx + b$$

So, our goal is to find *m* and *b,* which we will call the weights and the bias from now on. Let us take our example one step further and plot it.



We can fit our model with different slopes, as we change *b* and *m*.





To stick to the machine learning notation, let's rename *b* to be w_0 and *m* to be w_1. So now, we can rewrite the equation this way:

$$y = w_0 + w_1 * x$$

As we can see, there are infinite values for the weights, and we cannot tell, until now, which set of weights gives the best performance.

There are two main methods to determine these weights. Both these methods are based on minimizing the error. However, they differ in their approaches to doing so. The first method, as we will see, does this by getting a closed-form mathematical solution, while the second one is an iterative solution that tries to converge to the correct answer.

The first method is quite simple, as we say that the error is $\epsilon_i = y_i - \hat{y}_i$ where y_i is the true output for the example *i,* and $\hat{y}_i$ is the estimated output for the example *i*. So, the error, which is also called the residual, is the difference between them.

Our objective is to minimize the sum of the squared prediction errors. We use the square because we want to have all our errors to be positive values and eliminate any negative values.

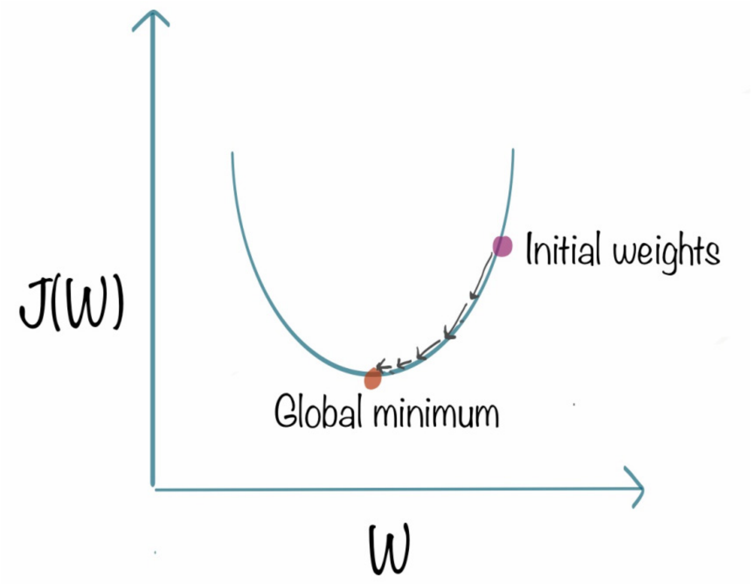
We could also minimize the sum of the absolute prediction errors, as this will also do the trick. However, using the squaring technique has some mathematical advantages over the absolute technique. Therefore, we will stick with the sum of the squared error technique.

So, we can write this mathematically as follows:

$$E = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - (w_0 + w_i x_i)^2$$

We can find *w* using some mathematical manipulation that we will not be concerned about right now, but it has a closed-form solution that is applied.

The second method is an iterative method called **gradient descent**. In this method, we have our cost function, which is the same as the sum of the squared errors. Our objective again is to find the weights that minimize the cost function as follows:



If you had studied pre-calculus in high school, you would know that by saying *minimize* or *maximize* for a function, we mean getting the first derivative of this function and making this derivative equal to zero. The symbol that we will use for the derivative of the cost function is ∇J. The most common optimization algorithm used in machine learning for minimization is called gradient descent.

The intuition behind the gradient descent is very simple. You start by choosing random weights. Then you calculate the first derivative of the cost function. After that, you move in the opposite direction of this value, with multiplying this number by a factor called the learning rate. Finally, we update the weights and repeat until convergence.

$$w = w - \alpha \nabla J(w)$$

So, you might have two questions. The first one is what should be the value of the learning rate. The answer to this question depends on the convergence rate. So, if we have an error and far from the right answer, then we will want a bigger learning rate. However, once we start converging, this big learning rate will make it difficult for us to reach the minimum value as it may overshoot.

Also, choosing a very small learning rate makes the model take too much time to converge, and it may also get stuck in a local minimum and does not reach the global minimum. Nonetheless, people tend to use the learning rate in the range of 10^{-2} – 10^{-5}. So, a good method to choose your learning rate is to start from 10^{-5} and increase it sharply if it gives you good results, then increase it carefully once you reach a critical value.

Note that the learning rate is not included in the trainable parameters of the model. Thus, we call it a *hyperparameter.*

The second question is, why do we take the negative of the gradient. The answer is that the derivative is the slope at this point, and the direction of that slope is in the opposite direction of the correct answer. Therefore, we use the negative sign in our calculation of the new weights.

It is also worth mentioning that it is preferred to use gradient-descent based learning than the closed-form solution when we have many features because it becomes computationally expensive to find a closed-form solution.

8.3. Multiple Regression

After going through all the details about simple linear regression, let us see now how we can expand this to multiple linear regression.

Let us continue from where we left when we discussed the income estimation problem. We assumed that we could estimate the income from just knowing the area of the house, but in fact, we need more information. This is because the more relevant information we have, the better we can estimate anything. So, for example, if we know the country that this person works in, the position that he/she holds, the field, and any more relevant information, we can define a better regression model.

Each one of these variables is called a feature. Thus, we have multiple features, and hence, multiple regression. However, if we have too many features, this might lead to worse estimation, as we will see in the common mistakes section.

The simple linear regression equation was as follows:

$$y = w_0 + w_1 * x$$

Now, we have multiple weights, one for each feature. Also, *x* is composed of different features, so we need a subscript for it too. Therefore, the generalized linear regression formula is as follows:

$$y = \sum_{i=0}^{n} w_i x_i = w^T x$$

The *T* superscript that we use for w is called the transpose, and this equation is the same as the sum equation. But it is mainly used when we convert our variables into vectors and matrices. By converting them, we can avoid using loops which takes too much time to finish if we have many inputs. Using vectors is always preferable as computers are optimized to perform matrix multiplication more than loops. We call this paradigm *vectorization*.

Also, note that this equation is also true for simple linear regression.

For optimizing a multiple linear regression model, the same concepts apply. This means we can use the same equation for gradient descent

$$w = w - \alpha \nabla J(w)$$

But now, we have multiple equations equal to the number of features that we have in our model.

Another key concept in multiple regression is the interaction effect, which happens in regression analysis when the effect of the predictive variable on the response variable changes, depending on the value of one or more predictive variables. This is represented as a product of two or more predictive variables. The following equation is for multiple regression without an interaction:

$$y = w_0 + w_1 x_1 + w_2 x_2$$

And the following equation is when we consider the interaction:

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_2 x_3$$

In the last equation, w_3 is a regression coefficient, while x_2x_3 is the interaction variable, which is also called a two-way interaction.

Now, to know if there is an interaction between two variables or not, we plot an interaction line graph. To do so, we follow these steps:

1. Plot the response variable on the *y*-axis and the predictive variable on the *x*-axis.
2. For each level of a potential interacting variable, plot the average score on the response variable separately.
3. Produce separate lines for each level of the interacting variable by connecting the average scores.

Then, to know if there is an interaction or not:

- If the lines are parallel, this means there is no interaction.
- If the lines are intersecting anywhere, this means that there is an interaction.

We will see in the hands-on project how we can do all this step-by-step.

Note also we can estimate the feature importance of the variables using different techniques. One technique is to fit the regression model using all predictive variables except one, then fit it again with all variables except another one. By keeping track of the loss, we can understand which variable is not that important in our estimation.

Another technique is to look at the correlation matrix and the p-value in the StatsModels summary results. If the correlation value between two variables is tending to 0 or/and the p-value is more than 0.05, then this means we can remove this feature without hurting the prediction power of our model.

8.4. Common Mistakes

In this section, we will discuss the common mistakes that people tend to do while working on a regression problem. We will go through the most three common mistakes.

8.5. Overfitting and Underfitting

Before we talk about what is meant by overfitting and underfitting, let us recall what we know so far about the main objective of regression analysis.

If you remember, the main objective is to recognize the pattern of the data, which can be measured by how well the algorithm performs on unseen data, not just the ones that the model was trained on.

This is called **generalization**, which is performing well on previously unseen input.

The problem in our discussion so far is that when we train our model, we calculate the training error. However, we need to know more about the testing error (generalization error).

Therefore, we need to split our dataset into two sub-datasets, one for training and one for testing. For traditional machine learning algorithms like linear regression with small datasets (less than 50,000 instances), we usually split the dataset into 70 percent for training and 30 percent for testing. If the dataset is large (more than 50,000 instances), we train on more than 70 percent and test on less than 30 percent.

Note that your model should not be exposed to the testing set throughout the training process. You might ask now, are there any guarantees that this splitting operation will make the two datasets have the same distribution?

This is hard to answer. But data science pioneers made all their algorithms based on the assumption that the data generation process is I.I.D., which means that the data are independent of each other and identically distributed.

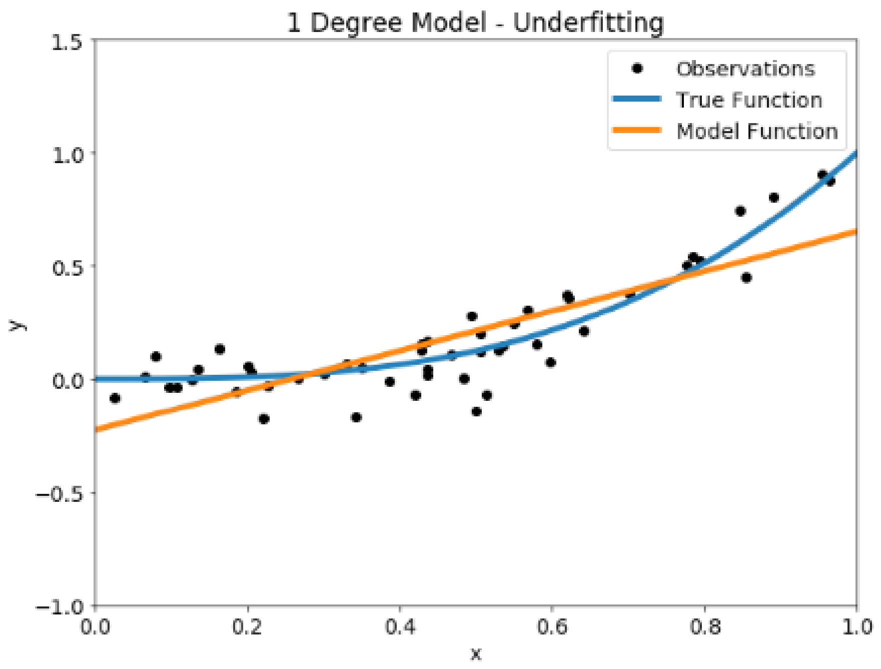
So, what are the factors that determine how well the linear regression model is doing?

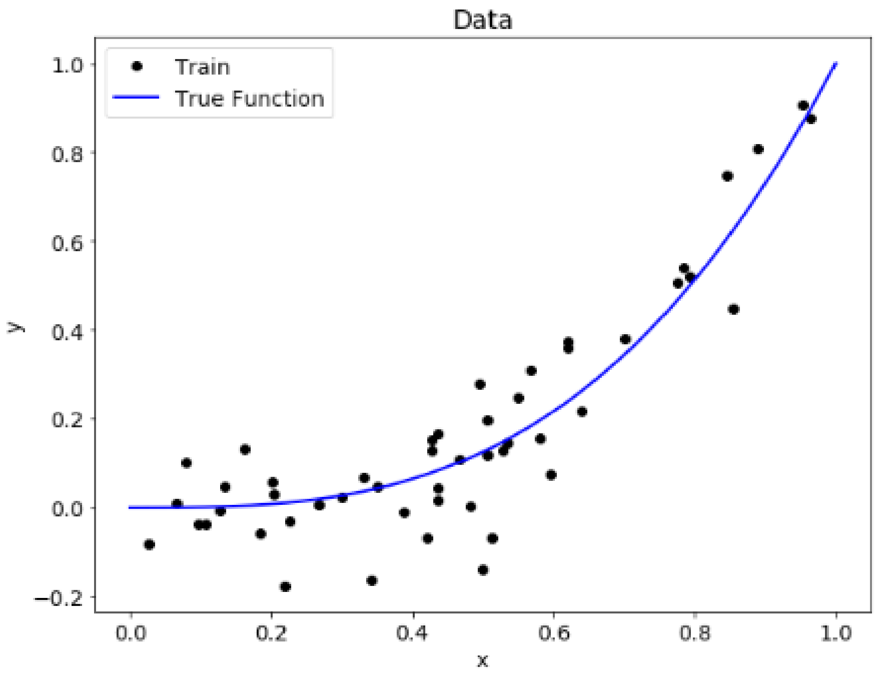
We can think of two main factors, which are making the training error small, and making the gap between the training error and testing error also small.

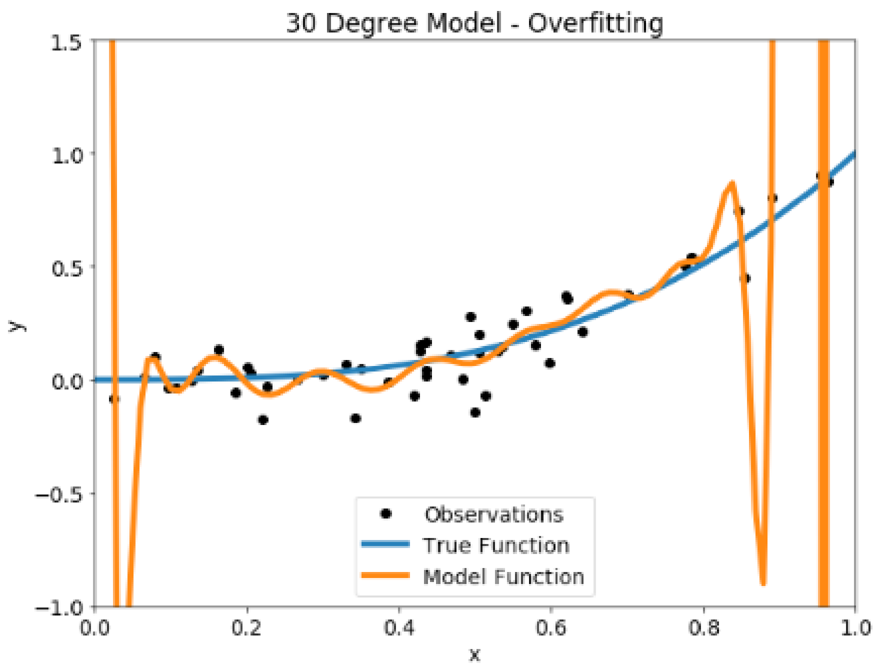
By defining these two factors, we can now introduce what is meant by **underfitting** and **overfitting**.

We say that the model is underfitting when the training error is large, as the model cannot capture the true complexity of the data.

We say that the model is overfitting when the gap between the training and testing errors is large, as the model is capturing even the noise among the data.


1 Degree Model - Underfitting
Observations
True Function
Model Function
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x


Data
Train
True Function
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
y
0.0
0.2
0.4
0.6
0.8
1.0
x



So, you might wonder, can we control this? The answer is yes. This can be achieved by changing the model **capacity**. Capacity is a term that is used in many fields. But in the context of machine learning and regression analysis, it is a measure of how complex a relationship that the model can describe. We say that a model that represents quadratic function—**polynomial of the second degree**—has more capacity than the model that can represent a linear function.

You can relate the capacity to overfitting and underfitting by thinking of a dataset that follows a quadratic pattern. If your model is a linear function, then it will underfit the data no matter what you do. If your model is a cubic function—**polynomial of third-degree**—then it will overfit the data.

Therefore, we can say that the model is performing well, if the capacity is appropriate for training data it is provided with, and the true complexity of the task it needs to perform. Given that knowledge, we can say with confidence that the model in the first figure is underfitting because it has low capacity, the model in the last figure is overfitting because it has a high capacity, and the model in the center figure is just right because it has the appropriate capacity.

The solution of underfitting is straightforward:

1. Increasing the size of the dataset
2. Increasing the complexity of the model
3. Training the model for more time until it fits.

Overfitting solution is a bit trickier because it needs more carefulness:

1. Gather more data, of course. However, this solution is not always feasible.
2. Use cross-validation. So, let us stop here and know what is meant by cross-validation.

So far, we split our dataset into training and testing. We said we train our model on the training set for the specified number of iterations, and after the training is finished, we test the model's performance by using the test set. But what if we need to test our model after each iteration so we can know if the model is converging or diverging? This is where the validation set comes to the rescue.

The validation dataset is simply another part of the dataset that is used for validating the performance of the model while it is still being trained. So, we split our dataset now into three datasets instead of two.

But the problem is, if the validation set is the same each time, we are back to square one again, which prevents us from using the testing set while training our model.

Therefore, to solve this problem, **K-fold cross-validation** was introduced. In this technique, the training dataset is split into k separate parts. The training process is repeated k times, with each time we randomly choose a subset that is held out for testing the model while the remaining subsets are used for training. The model overall error is the average error of all errors.

Leave-one-out cross-validation is a special type of k-fold cross-validation, where k is simply the number of instances in the dataset. So, each time we test only on one example and train on the rest. This method, of course, is not used because we cannot rely on one example.

Another reason it is not used is it is computationally expensive as we will need to train our model numbers of time equal to the size of the dataset to get the overall error.

After understanding what is meant by cross-validation, we can now understand why it is used for preventing overfitting:

1. Because now, we can monitor our model and stop the training whenever the gap between the training error and validation error is increasing. This is called early stopping.
2. Regularization, which is penalizing the model if it is getting too complex for the problem at hand.

8.6. High R^2 Score Overinterpreting

If you remember, we said that a high R2 score means that there is a high correlation between the two variables. However, you need to interpret it carefully, as it usually occurs also when the model is overfitting.

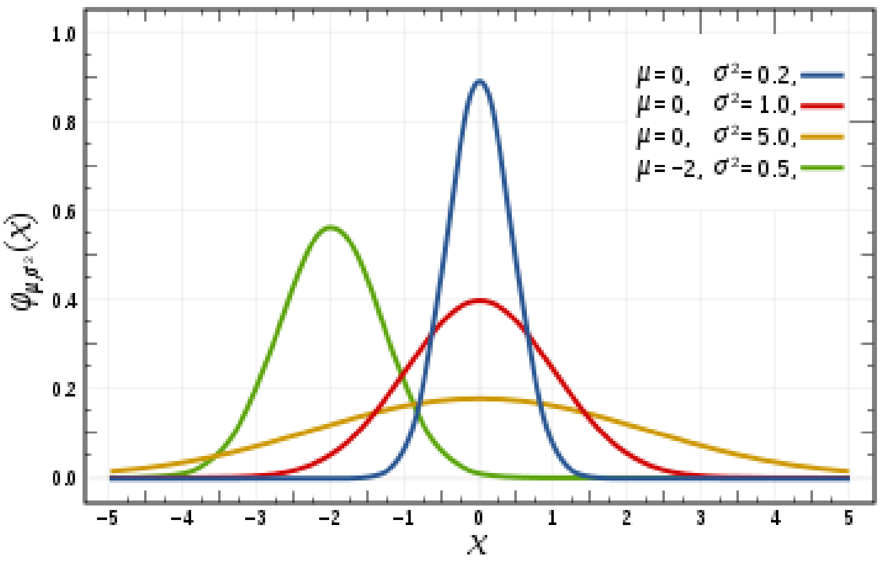
Moreover, you need to know what is "a high" R2 score, as it varies a lot from one domain to another. Therefore, you can get $R^2 = 0.6$ for a problem and $R^2 = 0.75$ for another problem, and the first one is considered high while the second one is not.

8.7. Insufficient Data Preprocessing

To have a promising regression model, 80 percent to 90 percent of the time should be devoted to data exploration, preparation, and preprocessing. Otherwise, all the tests and models will not be reliable.

One of the most commonly overlooked data preprocessing techniques is standardization, or mean removal and variance scaling. This is a crucial requirement for most of the machine learning algorithms, including regression analysis. This is because these algorithms assume that the features look like standard normally distributed data, which is a **Gaussian** with zero mean μ and unit variance σ^2.

We can see the plot of different Gaussian distributions in the following figure. We need to convert the distribution of the dataset's features to be as close as possible to the red graph.



To do so in Python, we transform the data by removing the mean value of each feature and then scale them by dividing non-constant features by their standard deviation.

Let us see how to do this in practice.

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])
X_scaled = preprocessing.scale(X_train)
print(X_scaled)
array([[ 0.  ..., -1.22...,  1.33...],
[ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

Let us see the mean and the variance of the data after standardization.

```
print(X_scaled.mean(axis=0))
print(X_scaled.std(axis=0))
array([0., 0., 0.])
array([1., 1., 1.])
```

Another technique to standardize the dataset is to compute the mean and the variance and apply them to any part of the data. Sklearn can help us to do so also using the following code.

We start by importing the needed libraries and loading the dataset. Then, we apply the standard scaler function found in Sklearn and fit it on the dataset. After that, we see the mean values of the four variables.

```
import numpy as np
import pandas as pd
import sklearn.preprocessing as preprocessing
fords=pd.read_csv('fords.csv')
fords = fords[['Year','Mileage','Price','Age']]
scaler = preprocessing.StandardScaler().fit(fords)
print(scaler.mean_)
[2.00488661e+03 5.60155714e+04 2.53135326e+04
4.28031496e+00]
```

We can also print the standard deviation of the dataset.

```
print(scaler.scale_)
[4.98562283e+00 3.41181011e+04 3.98060973e+05
3.28156481e+00]
```

Finally, we can apply this transformation on the dataset to standardize it.

```
print(scaler.transform(fords))
[[ 1.90775334e+01  2.66560640e+00 -5.60681255e-02
3.26663822e+00]
 [-1.78244815e+00  3.44797703e+00 -5.85702548e-02
2.65717288e+00]
 [ 6.24472796e-01 -1.46914892e+00 -1.71846352e-02
-9.99619129e-01]]
```

Another critical preprocessing technique is normalization, which is the process of scaling individual observations to have unit norm. It is very important in regression analysis to have the same weights for each feature in the dataset.

For example, suppose we want to predict the price of a house based on several features. One of these features can be the number of bedrooms, while another one can be the area of the house.

The problem now is that the range of values for the bedroom's variable is not on the same scale as the other variable. Thus, even though they can be equally important in our estimation of the price, the second variable mathematically will have a much bigger influence on the final estimation. That is why we need to normalize the dataset before we do any analysis.

Note the difference between standardization and normalization. The first one tries to distribute the data equally without changing the range of the values for each variable independently, while the second one tries to make all the values in the whole dataset on the same range.

There are a lot of mathematical tricks that can be used to convert to normalize the dataset. The most famous ones are the *l2 norm* and the *l1 norm*. We will not go into the mathematics behind them, but rather will see how to use them in Python.

```
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
X_normalized = preprocessing.normalize(X, norm='l2')
print(X_normalized)
array([[ 0.40..., -0.40...,  0.81...],
[ 1.  ...,  0.  ..., 0.  ...],
[ 0.  ...,  0.70..., -0.70...]])
```

As we can see, all the values are now in the range between 0 and 1.

8.8. Regression Hands-on Projects in Python

Now, let us work on two projects, one for simple linear regression, and the other one for multiple regression.

8.9. Simple Linear Regression Hands-on Projects in Python

Let us start with the simple linear regression project.

We start with the libraries.

```
import pandas as pd
import os
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
```

We load the data.

```
os.chdir('D:')
os.getcwd()
cars_df=pd.read_excel('cars.xls')
cars_df.head()
```

	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Year	Origin
0	chevroletchevellemalibu	18.0	8	307.0	130	3504	12.0	70	US
1	buick skylark 320	15.0	8	350.0	165	3693	11.5	70	US
2	plymouth satellite	18.0	8	318.0	150	3436	11.0	70	US
3	amc rebel sst	16.0	8	304.0	150	3433	12.0	70	US
4	ford torino	17.0	8	302.0	140	3449	10.5	70	US

We then choose the input and output variables and split the data.

```
y=cars_df.MPG
X=cars_df.Horsepower
X_train,X_test,y_train,y_test=train_test_split(
pd.DataFrame(X),y,test_size=0.3,random_state=42)
```

Then, we fit the model using the Sklearn linear regression module.

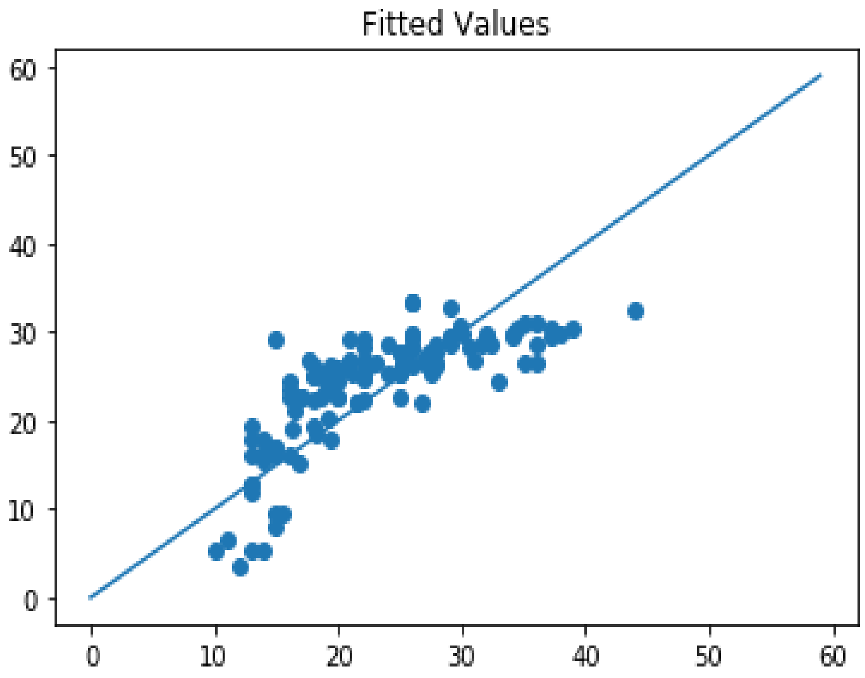
```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

After that, we test the model on the test data.

```
y_prediction = regressor.predict(X_test)
RMSE = sqrt(mean_squared_error(y_true = y_test, y_pred =
y_prediction))
print(RMSE)
4.955413560049774
```

Finally, let us plot the fitted slope to link everything together.

```
# add your actual vs. predicted points
plt.scatter(y_test, regressor.predict(X_test))
# add the line of perfect fit
straight_line = np.arange(0, 60)
plt.plot(straight_line, straight_line)
plt.title("Fitted Values")
plt.show()
```



8.10. Multiple Linear Regression Hands-on Projects in Python

Now, let us see how to use Sklearn to perform regression analysis and also understand interaction models.

We start, as always, by importing the libraries.

```
import pandas as pd
import os
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from math import sqrt
from sklearn.metrics import mean_squared_error
import statsmodels.api as sm
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import scipy as sp
import seaborn as sns
import statsmodels.formula.api as smf
```

Then we load the dataset.

```
os.chdir('D:')
os.getcwd()
cars_df=pd.read_excel('cars.xls')
cars_df.head()
```

	Model	**MPG**	**Cylinders**	**Displacement**	**Horsepower**	**Weight**	**Acceleration**	**Year**	**Origin**
0	chevroletchevellemalibu	18.0	8	307.0	130	3504	12.0	70	US
1	buick skylark 320	15.0	8	350.0	165	3693	11.5	70	US
2	plymouth satellite	18.0	8	318.0	150	3436	11.0	70	US
3	amc rebel sst	16.0	8	304.0	150	3433	12.0	70	US
4	ford torino	17.0	8	302.0	140	3449	10.5	70	US

After that, we split the data.

```
y=cars_df.MPG
X=cars_df[['Cylinders', 'Displacement', 'Horsepower', 
'Weight',
       'Acceleration', 'Year', 'Origin']]
```

As before, we create a dummy variable for the categorical feature.

```
X=pd.get_dummies(X,drop_first=True)
```

Then, we split the data into train and test.

```
X_train,X_test,y_train,y_test=train_test_split(X,y,
test_size=0.3,random_state=4)
```

Then, we perform feature scaling.

```
scaler=MinMaxScaler()
X_train_sc=pd.DataFrame(scaler.fit_transform(X_train),
columns=X_train.columns)
X_test_sc=pd.DataFrame(scaler.transform(X_test),
columns=X_test.columns)
```

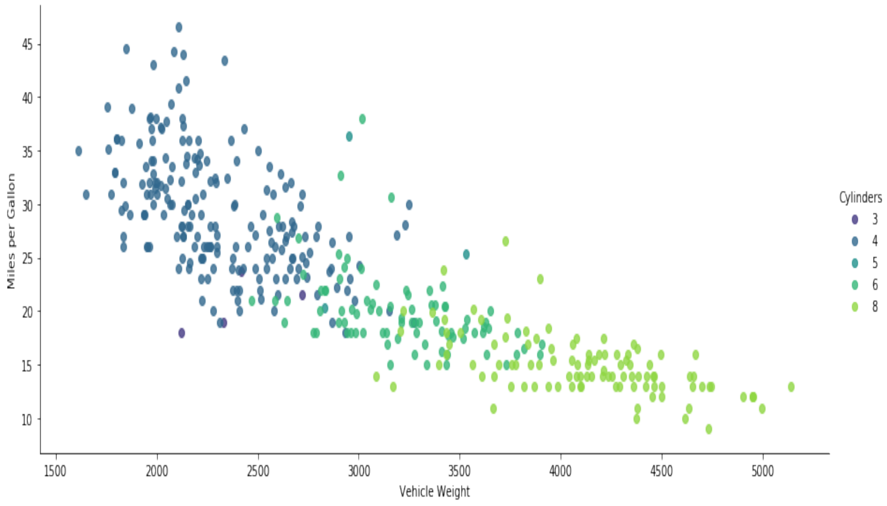
Finally, we train the linear regression model and test it on our test set.

```
regressor = LinearRegression()
regressor.fit(X_train_sc, y_train)
```

```
y_prediction = regressor.predict(X_test_sc)
RMSE = sqrt(mean_squared_error(y_true = y_test, y_pred =
y_prediction))
print(RMSE)
3.216235254988254
```

Now, let us understand the interaction models more. Suppose that your target variable is the *MPG,* and we want to know how the *Weight* variable affects the relationship between the *Cylinders* variable and the *MPG* variable. The first step is to plot the Miles per gallon on the y-axis and the Vehicle weight on the x-axis.

```
sns.lmplot(x='Weight', y='MPG', hue='Cylinders', data=cars_
df, fit_reg=False, palette='viridis', size=5, aspect=2.5)
plt.ylabel("Miles per Gallon")
plt.xlabel("Vehicle Weight");
```



Now, let us build a regression model which has the following equation:

$$MPG = w_0 + w_1 Weight + w_2 Cylinders$$

```
model = smf.ols(formula='MPG ~ Weight + Cylinders',
data=cars_df).fit()
summary = model.summary()
summary.tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	46.2923	0.794	58.305	0.000	44.731	47.853
Weight	-0.0063	0.001	-10.922	0.000	-0.007	-0.005
Cylinders	-0.7214	0.289	-2.493	0.013	-1.290	-0.152

From the coefficient column, we have the values of the weights, and we can write the equation again as follows:

$$MPG = 46.2923 - 0.0063 Weight - 0.7214 Cylinders$$

These coefficients can be interpreted as follows:

- For every unit increase in the weight variable, mpg decreases by 0.0063, assuming the cylinders variable is constant.
- For every unit increase in the cylinders variable, mpg decreases by 0.7214, assuming the weight variable is constant.

We can also notice that all p-values (the fourth column) are significant. We say that the p-value is significant if it is less than 0.05.

For any variable in the regression model, we must select one of two hypotheses:

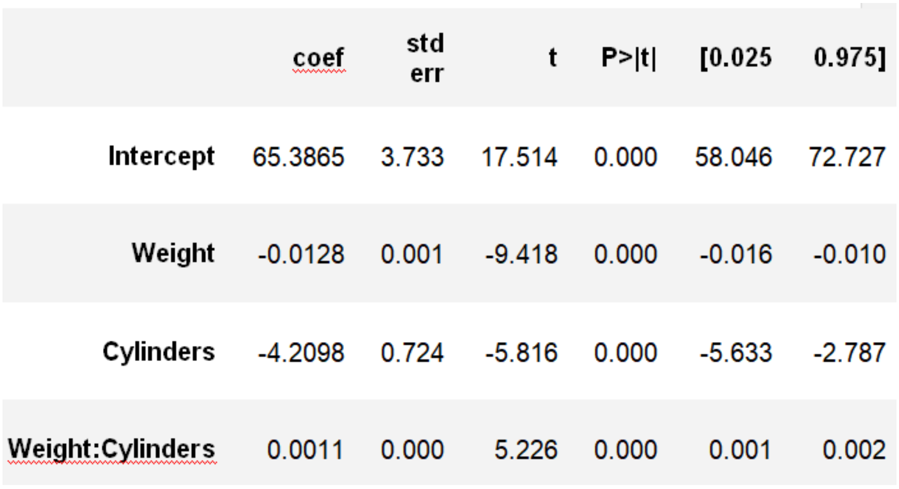
1. Null hypothesis: The coefficient for this variable is zero.
2. Alternative hypothesis: The coefficient for this variable is not zero.

If the p-value for a variable is significant—less than 0.05—, then we reject the null hypothesis. Therefore, we reject the null hypothesis for both variables.

Now, let us model the interaction between the weight variable and the cylinders variable.

We can do so using two ways. The first way is as follows.

```
model_interaction = smf.ols(formula='MPG ~ Weight +
Cylinders + Weight:Cylinders', data=cars_df).fit()
summary = model_interaction.summary()
summary.tables[1]
```



	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	65.3865	3.733	17.514	0.000	58.046	72.727
Weight	-0.0128	0.001	-9.418	0.000	-0.016	-0.010
Cylinders	-4.2098	0.724	-5.816	0.000	-5.633	-2.787
Weight:Cylinders	0.0011	0.000	5.226	0.000	0.001	0.002

We see that the equation now becomes,

$$MPG = 65.3865 - 0.0128Weight - 4.2098Cylinders \\ + 0.0011WeightCylinders$$

You can also notice that the p-value for the interaction variable is significant, confirming an interaction between the two variables.

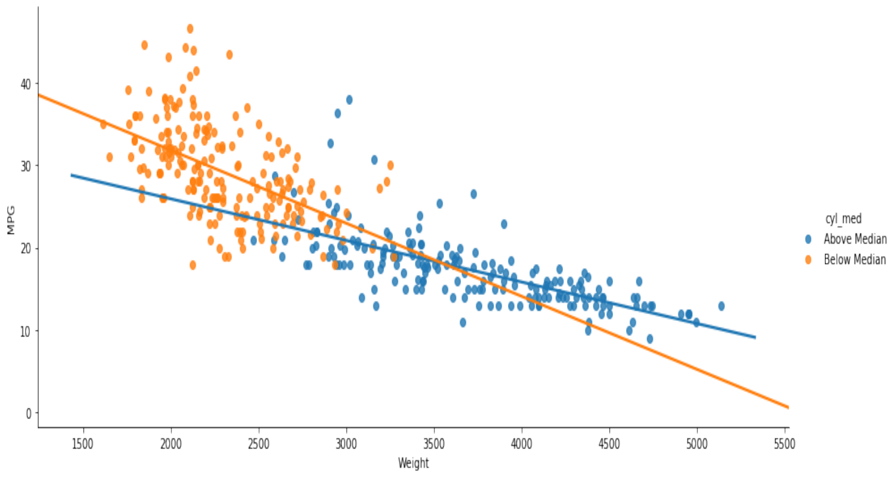
The second way to model the interaction is by adding another variable, which is the multiplication of the two variables.

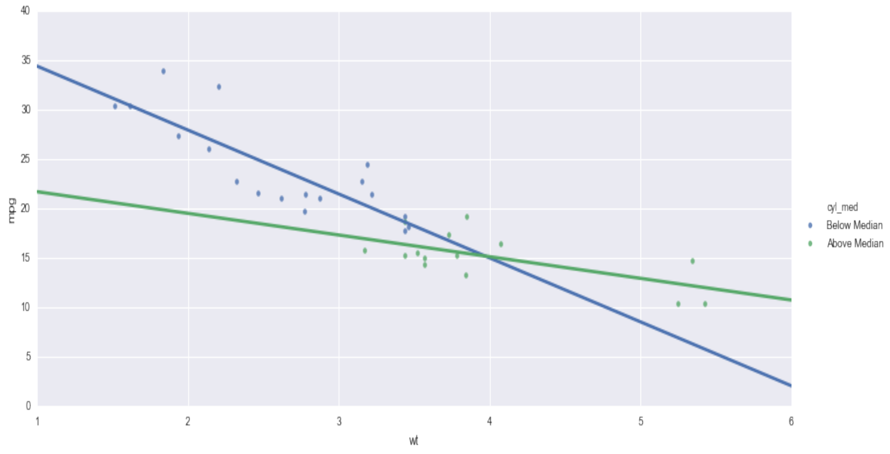
```
cars_df[‘wt_cyl’] = cars_df.Weight * cars_df.Cylinders

model_multiply = smf.ols(formula=’MPG ~ Weight + Cylinders
+ wt_cyl’, data=cars_df).fit()
summary = model_multiply.summary()
summary.tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	65.3865	3.733	17.514	0.000	58.046	72.727
Weight	-0.0128	0.001	-9.418	0.000	-0.016	-0.010
Cylinders	-4.2098	0.724	-5.816	0.000	-5.633	-2.787
Weight:Cylinders	0.0011	0.000	5.226	0.000	0.001	0.002

Now, let us plot the interaction graph using the steps that we explained earlier.

```
cars_df['cyl_med'] = cars_df.Cylinders>cars_df.Cylinders.
median()
cars_df['cyl_med'] = np.where(cars_df.cyl_med == False,
"Below Median", "Above Median")
sns.lmplot(x='Weight', y='MPG', hue='cyl_med', data=cars_
df, ci=None, size=5, aspect=2.5);
```





We can deduct from this graph that when the cylinder value is small—below median—the relationship between *MPG* and *Weight* is strongly negative. While on the other side, when the cylinder value is big—above median—the relationship between *MPG* and *Weight* is weaker.

Finally, we can deduce that the larger the differences in slopes, the larger the interaction effect.

9

Classification Analysis

In this chapter, we will explore in detail what is meant by classification analysis, as we will go through different classification algorithms and highlight the differences between regression and classification. We will start with the simplest classification algorithm called logistic regression. Then, we will discuss another more complicated one called Naïve Bayes. After that, we will understand a much more complex algorithm called Discriminant Analysis.

As with regression, we will have a dedicated section to highlight the common mistakes usually taking place in classification analysis. Finally, we will end the chapter with two complete hands-on projects for classification.

9.1. Logistic Regression

In linear regression, we saw how we could perform regression analysis using the linear regression equation with the gradient descent to update the weights. Now, we will do something very similar but for classification purposes.

The main difference between regression and classification is that in regression, we try to estimate a value in continuous space with no restriction, while in classification, our goal is to estimate also a value but within a discrete space with limited value.

For example, in house price estimation, the house price can be any value, while in dog breed classification, for example, we try to predict to which breed the current image of a dog belongs, so we know beforehand that it has to be one of 100 possible values if we assume that there are only 100 dog breeds in the world.

In order to simplify the classification problem and focus only on the algorithm of logistic regression, we will assume, for now, the ones that have only two classes. So, we can treat our problem as a binary classification problem. For example, we could predict if a student will be accepted in a specific university or not.

Therefore, we can formalize our output as a probability from [0,1], and if it is above a certain threshold, 0.5 for example, then this student will get accepted, and if it is less than the threshold, then he will get rejected.

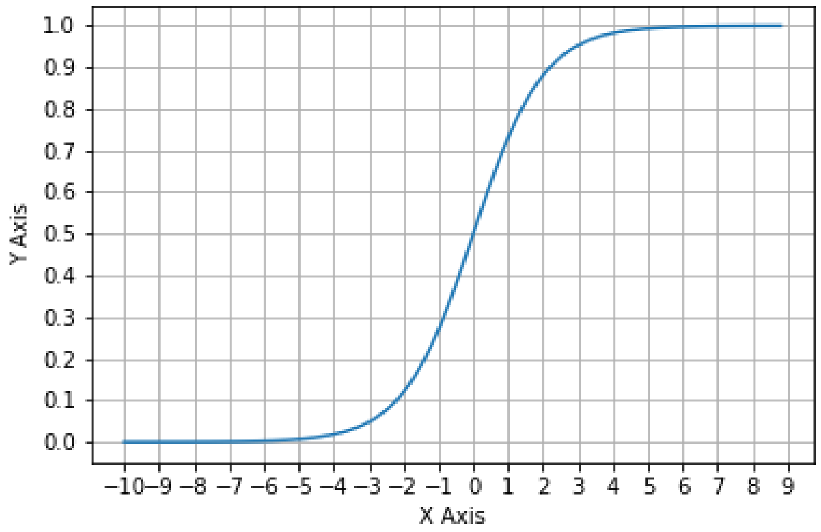
However, the equation that we used for linear regression is not limited by this constraint. So, we use a *logistic* function to transform our output to be in the range [0,1], so we can treat it as a probability. The most famous and currently used logistic function is the *sigmoid* which has the following equation:

$$y(z) = \frac{1}{1 + e^{-z}}$$

Where z is the linear equation that we used in linear regression,

$$z = \sum_{i=0}^{n} w_i x_i = w^T x$$

In order to understand how the sigmoid function squashes our input into [0,1], we can plot it using Python, and we will get the following curve.



We can generate this plot by simply writing the sigmoid as a Python function, and then call this function with different values of input.

As you can see, the output—Y-axis—can only take values in the range [0,1], and it reaches zero at negative infinity and reaches one and positive infinity. We can also see that the output is 0.5 when the input is zero. We can alter that by scaling the sigmoid function or changing the bias.

Moving to the loss function, we cannot use the same mean square error loss that we used for linear regression, as the numbers are all between 0 and 1, so the results will be signification. Thus, we need a loss function that is sensitive to small changes. To do so, we use the *negative log-likelihood* loss function, which is defined as follows:

$$J(w) = -\sum_i (y^i \log\left(h_w(x^i)\right) + (1-y^i)\log\left(1 - h_w(x^i)\right))$$

There is no closed-form solution to calculate the weights as in linear regression. Therefore, the only possible way to estimate the weights is to use an iterative solution such as Gradient Descent.

The mathematics behind the final output is complex. Hence, the only thing that you need to know is that gradient descent and other iterative optimization algorithms are the only way to update the weights in logistic regression, and thus, classify the output correctly.

By the end of this chapter, we will work on a hands-on project to implement logistic regression using Python.

9.2. Naïve Bayes

Another important classification algorithm is called Naïve Bayes, which is based on Bayes' theorem we discussed earlier in chapter 4.

So, before we start seeing the algorithm in action and how it can be used in Python, let us quickly revise the theory with an example.

First, let us write the formula of Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

If you remember, we said that the conditional probability $P(B|A)$ is called the *likelihood,* which is the probability of observing the new evidence, given our initial hypothesis. We also said that the marginal probability $P(A)$, which is called the *prior*, is the probability of our hypothesis without any additional prior information. Finally, we said that $P(B)$ is the *marginal probability*.

So, using Bayes' Rule, we can update our beliefs when new information or evidence is found.

You might be wondering how we can use Bayes' Rule classification analysis, and why the algorithm is called *Naive*. The answer to these questions can be obtained by looking at a classification problem and following the steps of the algorithm accordingly.

But before that, we should know that it is called *Naïve* mainly because it assumes that the features are independent, which means that the presence of one feature does not affect the others.

The best way to understand this algorithm is to see how it works using a numerical example. So, let us suppose that we want to decide if your friend is going to play golf or not tomorrow.

To do so, you observed and documented his decision in the last fourteen days while taking into consideration four attributes, which are the outlook, the temperature, the humidity, and the wind strength.

Finally, you come up with the following table:

Day	Outlook	Temperature	Humidity	Wind	Play Golf
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	Yes
D7	Overcast	Cool	Normal	Strong	No
D8	Sunny	Mild	High	Weak	Yes
D9	Sunny	Cool	Normal	Weak	No
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

We will assume that all the features are independent, which means that if the wind is weak, for example, then this does not imply anything about the outlook of this day. Another assumption is that all the features contribute equally to the prediction.

These assumptions are, of course, invalid in most cases. This is because the features by nature have some dependency on each other, while also some of the features are more important in predicting the output than the others. However, these two assumptions are crucial in order to derive the naïve Bayes' classifier, as we will see.

Let us rewrite the Bayes' Rule again.

$$P(Y|X) = \frac{P(X|Y) * P(Y)}{P(X)}$$

Where $X = (x_1, x_2, x_3, \dots, x_n)$, which represent the different features. If the features are independent, we can then write the Bayes' Rule again as follows:

$$P(Y|x_1, x_2, x_3, \dots, x_n) = \frac{P(x_1|Y) * P(x_2|Y) * \dots * P(x_n|Y) * P(Y)}{P(x_1) * P(x_2) * \dots * P(x_n)}$$

We can obtain all the values by looking at the dataset and substitute them into the equation. Let us do so for the outlook column, for example.

Outlook	Play Golf
Sunny	No
Sunny	No
Overcast	Yes
Rain	Yes
Rain	Yes
Rain	Yes
Overcast	No
Sunny	Yes
Sunny	No
Rain	Yes
Sunny	Yes
Overcast	Yes
Overcast	Yes
Rain	No

We can then get the frequency table as follows:

Weather	Yes	No
Overcast	4	-
Rainy	2	3
Sunny	3	2

Then, we can get the likelihood table as follows:

Weather	Probability
Overcast	0.29
Rainy	0.36
Sunny	0.36

Now, let us assume we want to know the probability that our friend will play if the weather is sunny. We can convert this to be:

$$P(Yes|Sunny) = \frac{P(Sunny|Yes) * P(Yes)}{P(Sunny)}$$

From our likelihood table, we got that $P(Sunny) = 0.36$, and from the frequency table, we got that $P(Yes) = \frac{9}{14}$. Also, we get that $P(Sunny|Yes) = \frac{3}{9}$ because we have 9 Yeses and only 3 of them were Sunny. Therefore, the number of times we get $(Yes|Sunny) = 0.6$.

We observe that the denominator does not change because all the features are independent. As a result, we can remove it and add a proportionality instead.

$$P(Y|x_1, x_2, x_3, \ldots, x_n) \propto P(Y) * \prod_{i=1}^{n} P(x_i|Y)$$

Where the Π represents the multiplication of the probability.

We can manipulate this even further by saying that we want to find the class *y*, which gives us the maximum probability. This was fairly easy in case of a binary classification problem like the golf problem because if we got $P(Yes|Sunny) = 0.6$, then $P(No|Sunny)$ will be 0.4, and we really do not need to calculate it. However, if the classification problem is multivariate, then we need a formula for that.

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

By getting this formula, we can classify the output, which is our ultimate goal.

As you can see, Naïve Bayes is a data-driven algorithm that does not require to calculate any weights or define any loss functions.

Finally, Naïve Bayes has only one hyperparameter, which is called alpha. Increasing the value of this hyperparameter smoothes the naïve Bayes' model, which makes it even more naïve, to a limit, of course. Decreasing it will make the model make fewer assumptions and may result in better accuracy. However, changing the value of this hyperparameter has a little influence on the overall performance of the algorithm.

There are three main variations of naïve Bayes that are used in practice, which are Multinomial Naïve Bayes, Complement Naïve Bayes, and Bernoulli Naïve Bayes.

Further Readings

If you want to know more about the different variations of Naïve Bayes, you can look here:

https://scikit-learn.org/stable/modules/naive_bayes.html

9.3. Linear Discriminant Analysis

The final classification algorithm that we are going to discuss is the Linear Discriminant Analysis, which is a dimensionality reduction technique that is usually used as a preprocessing step in the machine learning pipeline. By doing so, the algorithm can remove the redundant features and reduce both the complexity and the dimensionality of the problem.

The algorithm has three main steps, as follows:

1. Calculate the separability between different classes, which is also called the between-class variance. It is calculated using the following equation: $S_b = \sum_{i=1}^{g} N_i(\overline{x}_i - \bar{x})(\overline{x}_i - \bar{x})^T$, where N is the average, $\bar{x}$ is the sample, and g is the total number of classes.
2. Calculate the within-class variance, which is the distance between the mean and the samples of each class. It can be calculated as follows: $S_w = \sum_{i}^{g}(N_i - 1)S_i$
3. Construct the lower-dimensional space that minimizes the within-class variance and maximizes the between-class variance.

9.4. Common Mistakes

As we did in the last chapter when we discussed the most common mistakes in regression analysis, let us do the same but for classification analysis.

9.4.1. Overfitting

The most frequent mistake is, as with regression, **overfitting**. We can solve it in logistic regression, for example, by using regularization.

Regularization is a fancy word for the penalty as we penalize the model if it is getting more complex. We can understand this better by looking at how we update the weights when we introduce the regularization term.

$$w = w - \alpha \frac{\partial J(w)}{\partial x} - \lambda \alpha w$$

We say that α is the learning rate and λ is the penalization term. So, we see that the regularization term is added as a second term in the loss function. The purpose of this regularization term is to push the parameters toward smaller numbers, and thus, the model does not become more complex, and hence, it does not overfit.

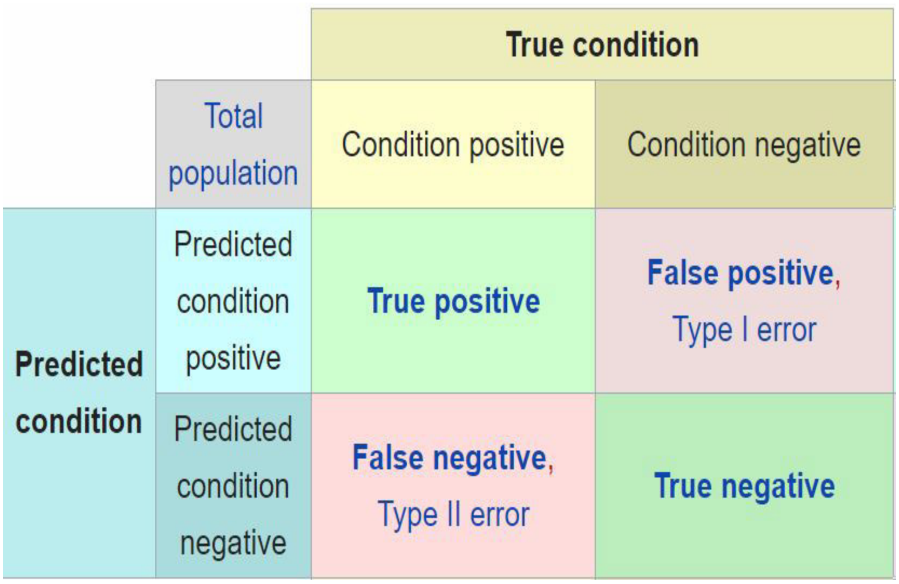
There are many different methods to implement the regularization term. However, the two most used ways are the Lasso method, also called *L1*, and the Ridge method, which is also called *L2*.

The main difference between the two methods is that the Lasso method tries to push all the parameters toward zero, while the Ridge method tries to push all the parameters toward very small numbers but not equal to zero. Both methods are used, and you need to experiment with both to know which one works best for each specific case.

9.4.2. The Right Metric

There are many more metrics that you can choose from when working on a classification problem than when working on a regression problem, and this can be quite confusing. We can state the most important ones as follows:

1. **Accuracy:** Here, we report the number of predicted outputs that match the true outputs.
2. **Confusion Matrix:** We can see the confusion matrix in the following figure.



		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	**True positive**	**False positive,** Type I error
	Predicted condition negative	**False negative,** Type II error	**True negative**

From the confusion matrix, we can get the accuracy which is,

$$Accuracy = \frac{TP + TN}{P + N}$$

Also, we can get another two metrics called the precision and the recall:

$$Precision = Positive\ Predictive\ Value = \frac{TP}{TP + FP}$$

$$Recall = Sensitivity = True\ Positive\ Rate = \frac{TN}{TP + FN}$$

And we can also get a combination of the precision, and the recall called the F-score as follows:

$$F = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$$

So, why do not we just get the accuracy? Why do we need precision and recall?

To answer this question, let us look at two different classification problems and see if the accuracy is the best evaluation metric to use.

Suppose that you have a spam classification problem where you classify emails to be either spam or ham. So, we have two different kinds of errors, which are the false positives and the false negatives. The false positives occur when we misclassify a ham email as a spam email, and the false negatives occur when we misclassify a spam email as a ham email. Which of these two errors are more critical? I think you agree with me that not putting an important email into the spam folder is more crucial than getting annoyed with a spam email into your main email folder. Of course, both are considered types of errors, but in our problem, we care more about having the minimum number of false positives. Thus, we use **precision** as our metric when evaluating the model.

For the second problem, suppose we have a cancer detection problem, wherewith we classify the patients to either having cancer or not. Again, we have two types of errors, which are predicting that a healthy patient has cancer and predicting that a sick patient does not have cancer. Here, in contrast to the first example, we care more about the false negatives because of the nature of the problem itself. We really do not want a cancer patient to be classified as healthy, but we can accept that some of our healthy patients are misclassified because then they will do more tests, and they will find themselves healthy afterward. Thus, we use **recall** as our evaluation metric.

If we want a **harmonic mean** between accuracy and recall, then we use the **F-score.**

3. **ROC curve:** This is short for Receiver Operating Characteristic. ROC curve is just a plot of the False Positive Rate against the True Positive Rate. It is used mainly to select the optimum model, which should have an area under the curve—AUC—equal to or near 1. This is because the True Positive Rate should be equal to or near 1, while the False Positive Rate should be equal to or near 0. Moreover, a random classifier is found to have an AUC of 0.5.

9.5. Classification of Hands-on Projects

Finally, we will conclude this chapter and this book by two main hands-on projects for logistic regression and Naïve Bayes.

9.5.1. Logistic Regression Hands-on Project

Let us start with the logistic regression one.

We start, as always, by importing the libraries.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import
StandardScaler,MinMaxScaler
from sklearn import metrics
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
```

Then, we load the dataset.

```
os.chdir('D:')
os.getcwd()
credit=pd.read_csv('german_credit.csv')
print(credit.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 22 columns):
Customer_ID                 1000 non-null int64
checking_account_status     1000 non-null object
loan_duration_mo            1000 non-null int64
credit_history              1000 non-null object
purpose                     1000 non-null object
loan_amount                 1000 non-null int64
savings_account_balance     1000 non-null object
time_employed_yrs           1000 non-null object
payment_pcnt_income         1000 non-null int64
gender_status               1000 non-null object
other_signators             1000 non-null object
time_in_residence           1000 non-null int64
property                    1000 non-null object
age_yrs                     1000 non-null int64
other_credit_outstanding    1000 non-null object
home_ownership              1000 non-null object
number_loans                1000 non-null int64
job_category                1000 non-null object
dependents                  1000 non-null int64
telephone                   1000 non-null object
foreign_worker              1000 non-null object
bad_credit                  1000 non-null int64
dtypes: int64(9),object(13)
```

Then, we choose the bad credit variable to be the output variable that we want to predict and classify.

```
Y=credit['bad_credit']
```

Then, we create a dummy variable for the credit categorical variable.

```
X=pd.get_dummies(credit)
```

After that, we split the dataset, scale it, and train it.

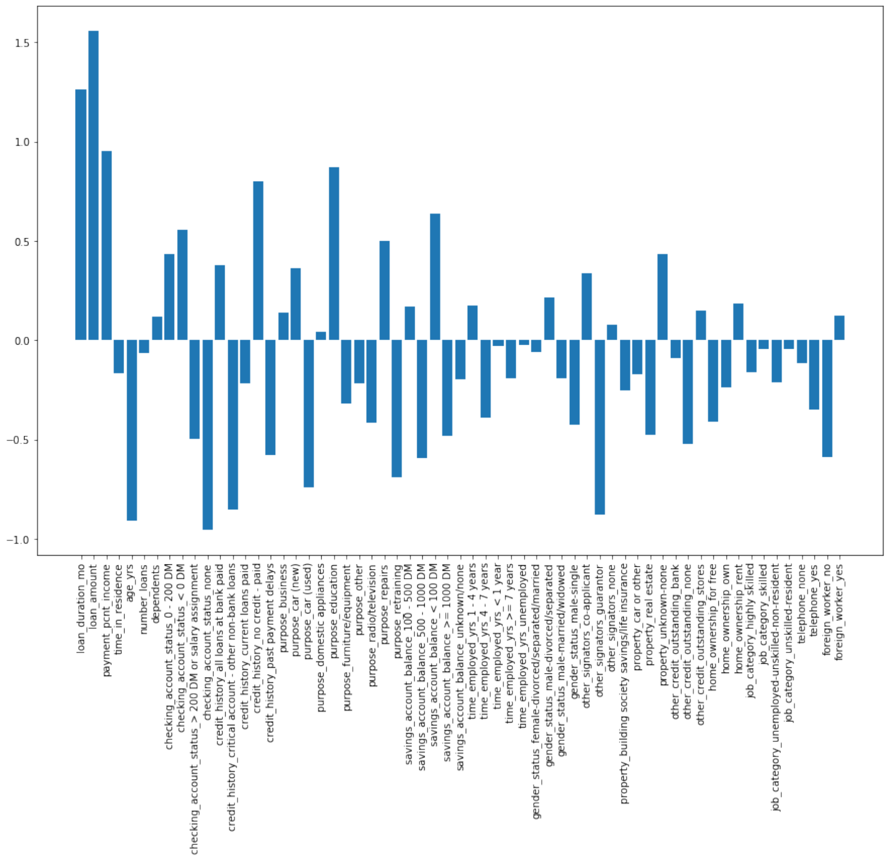
```
X_train,X_test,Y_train,Y_test=train_test_split(X,
Y,test_size=0.3,random_state=42)
scaler=MinMaxScaler()
X_train=pd.DataFrame(scaler.fit_transform(X_train),
columns=X_train.columns)
X_test=pd.DataFrame(scaler.transform(X_test),
columns=X_test.columns)
logreg=LogisticRegression()
mod1=logreg.fit(X_train,Y_train)
```

Now, let us test our model.

```
pred1=logreg.predict(X_test)
accuracy_score(y_true=Y_test, y_pred=pred1)
0.77
```

Let us also plot the weight for each feature to see how important it is.

```
plt.figure(figsize=(15,10))
plt.bar(X_train.columns.tolist(),logreg.coef_[0])
plt.xticks(rotation=90,size=10)
plt.show()
```



Now, let us see what the results would be with either L1 or L2 regularization.

```
logreg1=LogisticRegression(penalty='l1',C=1)
mod2=logreg1.fit(X_train,Y_train)
pred2=logreg1.predict(X_test)
accuracy_score(y_true=Y_test, y_pred=pred2)
0.7566666666666667
```

```
logreg2=LogisticRegression(penalty='l2',C=0.01)
mod3=logreg2.fit(X_train,Y_train)
pred3=logreg2.predict(X_test)
accuracy_score(y_true=Y_test, y_pred=pred3)
0.7
```

As we can see, the results became worse, which means that we need more data as the model is underfitting and not overfitting.

9.5.2. Naïve Bayes Hands-on Project

Now, let us walk through the Naïve Bayes one.

First, we import the needed libraries. We will use a real-world dataset from Sklearn called 20 newsgroups, which contains 18846 examples in text form belonging to 20 different classes.

Hands-on Time – Link to more dataset

You can check more about this interesting dataset by going to this link :

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html#sklearn.datasets.fetch_20newsgroups

We will also use a function called TfidfVectorizer, which is used to get something similar to the frequency table that we used in the golf example, but for text. We will also evaluate our model using the confusion matrix, which we mentioned in the common mistakes section.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.metrics import confusion_matrix
```

Then, we will load the dataset and split it into a training set and a test set.

```
data=fetch_20newsgroups()
```

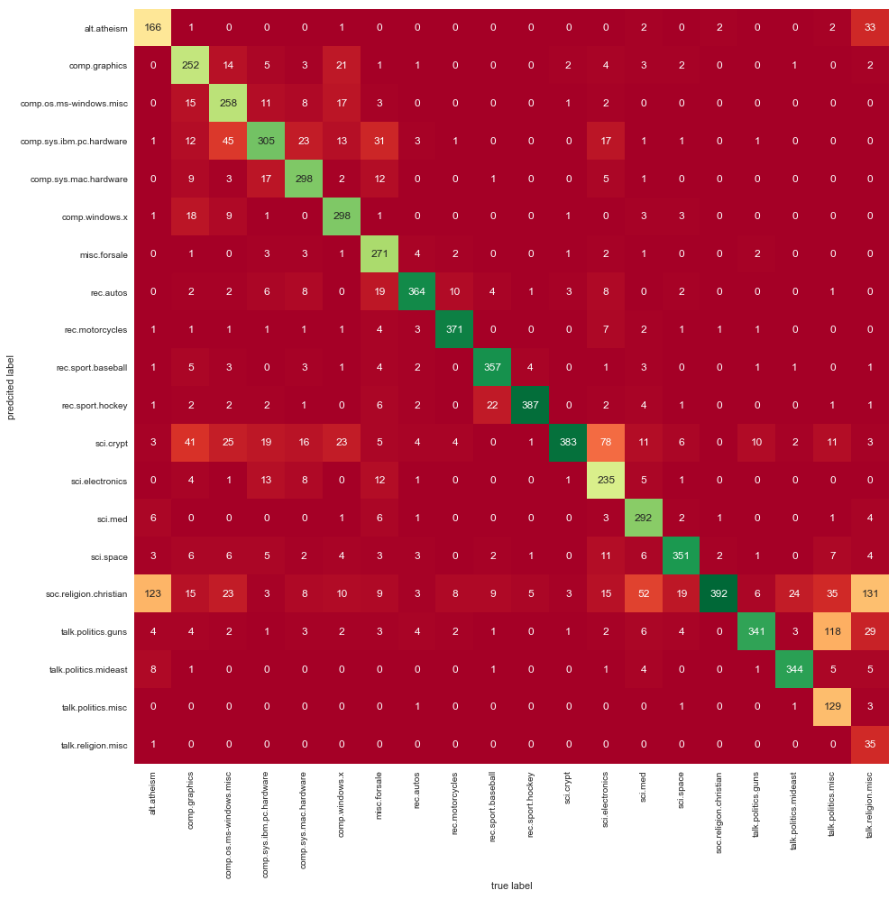
```
train=fetch_20newsgroups(subset='train')
test=fetch_20newsgroups(subset='test')
```

We then create our multinomial Naïve Bayes' classifier and train it. Then, we test it and store the predicted outputs.

```
model=make_pipeline(TfidfVectorizer(),MultinomialNB())
model.fit(train.data,train.target)
labels=model.predict(test.data)
```

Given that we have a multi-class classification problem, we need a more informative score than the accuracy. Thus, we use the confusion matrix.

```
mat=confusion_matrix(test.target,labels)
plt.figure(figsize=(15,30))
sns.heatmap(mat.T,square=True,annot=True,fmt='d',
            cbar=False,cmap='RdYlGn',
xticklabels=train.target_names,
yticklabels=train.target_names)
plt.xlabel('true label')
plt.ylabel('predcited label')
```



Finally, we can create a simple function that gives us the predicted class, and we use it to predict a given text.

```
def predict_category(s,train=train,model=model):
pred=model.predict([s])
    return train.target_names[pred[0]]
```

```
predict_category('Jesus Christ')
'soc.religion.christian'
```

```
predict_category('Bush')
'talk.politics.misc'
```

Thanks, Data Scientist and Python Programmer

Congrats on completing this book.
You now have an elementary understanding of the main concepts of Python for data analysis.

If you want to help us produce more material like this, then please leave an honest review online.
It really does make a difference.

If you have any feedback, kindly let us know by sending an email to contact@aispublishing.net.

Your feedback is highly valued. At AI Publishing, we are genuinely excited about hearing from you. It will be very helpful for us to improve the quality of our books.

Until next time, happy analyzing.

How to Contact Us

If you have any feedback, please let us know by sending an email to contact@aispublishing.net.

Your feedback is immensely valued, and we look forward to hearing from you. It will be beneficial for us to improve the quality of our books.