

AI PUBLISHING

Statistics Crash Course for Beginners

Theory and Applications of
Frequentist and Bayesian Statistics
Using Python

Our Books are designed
to teach beginners
Data Science and AI

© Copyright 2020 by AI Publishing
All rights reserved.
First Printing, 2020

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

ISBN-13: 978-1-7347901-6-0

The contents of this book may not be copied, reproduced, duplicated, or transmitted without the direct written permission of the author. Under no circumstances whatsoever will any legal liability or blame be held against the publisher for any compensation, damages, or monetary loss due to the information contained herein, either directly or indirectly.

Legal Notice:

You are not permitted to amend, use, distribute, sell, quote, or paraphrase any part of the content within this book without the specific consent of the author.

Disclaimer Notice:

Kindly note that the information contained within this document is solely for educational and entertainment purposes. No warranties of any kind are indicated or expressed. Readers accept that the author is not providing any legal, professional, financial, or medical advice. Kindly consult a licensed professional before trying out any techniques explained in this book.

By reading this document, the reader consents that under no circumstances is the author liable for any losses, direct or indirect, that are incurred as a consequence of the use of the information contained within this document, including, but not restricted to, errors, omissions, or inaccuracies.

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-sccb>

The order number is required.

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 science 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 science to advanced courses for machine learning, deep learning, computer vision, big data, and much more. The programming languages used include 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 science to the maximum extent.

Moreover, we offer specialized assistance in the form of our 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.

Our experts have cautiously developed our contents and kept them concise, short, and comprehensive so that you can understand everything clearly and effectively and start practicing the applications right away.

We also offer consultancy and corporate training in AI and data science for enterprises so that their staff can navigate through the workflow efficiently.

With AI Publishing, you can always stay closer to the innovative world of AI and data science.

If you are eager to learn the A to Z of AI and data science but have no clue where to start, AI Publishing is the finest place to go.

Please contact us by email at contact@aispublishing.net.

AI Publishing is Looking for Authors Like You

Interested in becoming an author for AI Publishing? Please contact us at author@aispublishing.net.

We are working with developers and AI tech professionals just like you, to help them share their insights with the global AI and Data Science lovers. You can share all your knowledge about hot topics in AI and Data Science.

Table of Contents

How to Contact Us	iii
About the Publisher	iv
Chapter 0: Preface	1
Why Learn Statistics?.....	1
The difference between Frequentist and Bayesian Statistics... <td>2</td>	2
What's in This Book?	3
Background for Reading the Book	4
How to Use This Book?.....	5
About the Author	7
Get in Touch With Us	8
Download the PDF version	9
Chapter 1: A quick Introduction to Python for Statistics 11	
1.1 Installation and Setup of Python Environment	11
1.1.1 Windows.....	12
1.1.2 Apple OS X	17
1.1.3 GNU/Linux	19
1.1.4 Creating and Using Notebooks	20

1.2	Mathematical Operators in Python.....	24
1.2.1	<i>Arithmetic Operators</i>	26
1.2.2	<i>Bitwise Operators</i>	27
1.2.3	<i>Assignment Operators</i>	29
1.2.4	<i>Logical Operators</i>	30
1.2.5	<i>Comparison Operators</i>	31
1.2.6	<i>Membership Operators</i>	32
1.3	String Operations	33
1.4	Conditional Statements and Iterations.....	39
1.4.1	<i>If, Elif and Else Statements</i>	40
1.4.2	<i>For Loop</i>	44
1.4.3	<i>While Loop</i>	47
1.5	Functions in Python.....	49
1.6	Data Structures.....	51
1.6.1	<i>Lists</i>	52
1.6.2	<i>Tuples</i>	54
1.6.3	<i>Sets</i>	55
1.6.4	<i>Dictionaries</i>	57
1.7	Python Libraries for Statistics.....	61
1.7.1	<i>NumPy for Mathematical Functions</i>	62
1.7.2	<i>Pandas for Data Processing</i>	64
1.7.3	<i>Statistics: Python's Built-in Module</i>	70
1.7.4	<i>Matplotlib for Visualization and Plotting</i>	72
1.7.5	<i>SciPy.stats Module for Statistical Functions</i>	73
1.7.6	<i>Statsmodels for Statistical models</i>	75
1.7.7	<i>PyMC for Bayesian Modeling</i>	77
1.8	Exercise Questions	79

Chapter 2: Starting with Probability83

2.1	Definition of Probability	83
2.2	Some Important Definitions	85
2.3	Samples Spaces and Events.....	86
2.4	Axioms of Probability	90
2.5	Calculating Probability by Counting	90
2.6	Combining Probabilities of More than One Events	94
2.7	Conditional Probability and Independent Events	94
2.8	Bayes' Theorem	98
2.9	Calculating Probability as Degree of Belief.....	100
2.10	Exercise Questions.....	103

Chapter 3: Random Variables & Probability Distributions.....107

3.1	Random Variables: Numerical Description of Uncertainty	107
3.2	Generation of Random Numbers and Random Variables.....	109
3.3	Probability Mass Function (PMF)	112
3.4	Probability Density Function (PDF)	117
3.5	Expectation of a Random Variable	120
3.6	Probability Distributions	121
3.6.1	<i>Bernoulli and Binomial Distribution</i>	122
3.6.2	<i>Uniform Distribution</i>	128
3.6.3	<i>Normal (Gaussian) Distribution</i>	132
3.6.4	<i>Poisson Distribution</i>	136
3.7	Exercise Questions.....	140

Chapter 4: Descriptive Statistics: Measure of Central Tendency and Spread.....145

4.1	Measuring the Central Tendency of Data	145
4.1.1	<i>The Mean</i>	146
4.1.2	<i>The Median</i>	146
4.1.3	<i>The Mode</i>	147
4.2	Measuring the Spread of Data.....	148
4.2.1	<i>The Range</i>	148
4.2.2	<i>The InterQuartile Range (IQR)</i>	148
4.2.3	<i>The Variance</i>	150
4.2.4	<i>The Standard Deviation</i>	153
4.3	Covariance and Correlation	155
4.4	Exercise Questions.....	158

Chapter 5: Exploratory Analysis: Data Visualization161

5.1	Introduction	161
5.2	Bar (Column) Charts.....	162
5.3	Pie Charts.....	166
5.4	Line Plots for Continuous Data	167
5.5	Scatter Plot	170
5.6	Histogram	172
5.7	Creating a Frequency Distribution.....	176
5.8	Relation between PMF, PDF, and Frequency Distribution	179
5.9	Cumulative Frequency Distribution and Cumulative Distribution Function (CDF)	180
5.10	The Quantile Function	186
5.11	The Empirical Distribution Function.....	191
5.12	Exercise Questions.....	194

Chapter 6: Statistical Inference..... 197

6.1	Basics of Statistical Inference and How It Works?.....	197
6.2	Statistical Models and Learning	198
6.3	Fundamentals Concepts in Inference	201
6.3.1	<i>Point Estimation</i>	201
6.3.2	<i>Interval Estimation</i>	203
6.4	Hypothesis Testing	204
6.4.1	<i>Null and Alternative Hypotheses</i>	205
6.4.2	<i>Procedure for Hypothesis Testing</i>	207
6.5	Important Terms used in Hypothesis Testing	207
6.5.1	<i>Sampling Distribution</i>	207
6.5.2	<i>Errors in Hypothesis Testing</i>	208
6.5.3	<i>Tests for Statistical Hypotheses</i>	209
6.5.4	<i>z-value (z-score)</i>	210
6.5.5	<i>p-value</i>	211
6.6	Exercise Questions.....	219

Chapter 7: Frequentist Inference 223

7.1	Parametric Inference	223
7.2	Confidence Intervals	230
7.3	Nonparametric Inference.....	233
7.4	Hypothesis Testing using z Tests	239
7.4.1	<i>One-tailed z Test</i>	240
7.4.2	<i>Two-tailed z Test</i>	242
7.5	Exercise Questions.....	245

Chapter 8: Bayesian Inference..... 247

8.1	Conditional Probability	248
8.2	Bayes' Theorem and the Bayesian Philosophy	250
8.3	Computations in Bayesian Inference.....	253

8.3.1	<i>Computing Evidence: Total Probability</i>	253
8.3.2	<i>Steps to Follow for Bayesian Inference</i>	255
8.4	Monte Carlo Methods.....	256
8.5	Maximum a Posteriori (MAP) Estimation	259
8.6	Credible Interval Estimation	262
8.6.1	<i>Beta Distribution as a Prior</i>	263
8.6.2	<i>Gamma Distribution as a Prior</i>	267
8.7	Naïve Bayes' Classification.....	275
8.8	Comparison of Frequentist and Bayesian Inferences	278
8.9	Exercise Questions.....	280
Chapter 9: Hands-on Projects	285
9.1	Project 1: A/B Testing Hypothesis - Frequentist Inference	286
9.2	Project 2: Linear Regression using Frequentist and Bayesian Approaches	298
9.2.1	<i>Frequentist Approach</i>	302
9.2.2	<i>Bayesian Approach</i>	305
Answers to Exercise Questions	313

Preface

§ Why Learn Statistics?

The fields of Artificial Intelligence (AI), Machine Learning (ML), and Data Science (DS) are prevailing in many real-world applications. A crucial part of these fields is to deal with a huge amount of data, which is produced at an unprecedented rate nowadays. This data is used to extract useful information for making future predictions on unseen but similar kinds of data.

Statistics is the field that lies at the core of Artificial Intelligence, Machine Learning, and Data Science. Statistics is concerned with collecting, analyzing, and understanding data. It aims to develop models that are able to make decisions in the presence of uncertainty. Numerous techniques of the aforementioned fields make use of statistics. Thus, it is essential to gain knowledge of statistics to be able to design intelligent systems.

§ The difference between Frequentist and Bayesian Statistics

This book is dedicated to the techniques for frequentist and Bayesian statistics. These two types of statistical techniques interpret the concept of **probability** in different ways.

According to the frequentist approach, the probability of an event is defined for the repeatable events whose outcomes are random. The statistical experiment is run again and again in a long run to get the probability of the event. Thus, the probability of an event equals the long-term **frequency** of occurrence of that event.

For example, rolling a six-sided dice can be considered a repeatable statistical experiment. The outcome of this experiment can be any number from 1 to 6. Since we do not know what will be the outcome in a particular rolling of the dice, we call it a random outcome. According to the frequentist approach, the chance of getting any particular number from 1 to 6 is equally likely. In other words, the probability of any number is $1/6$ or 1 out of 6.

As another example, in a pack of 52 cards, we randomly draw a card. We want to check the chance of getting a king. To find the probability of our defined event, i.e., getting a king, we count the number of favorable outcomes: 4 out of 52 cards. Thus, the probability of getting a king is obtained by dividing the number of favorable outcomes by the total number of possible outcomes: $4/52 = 1/13$.

The frequentist way of doing statistics makes use of the data from the current experiment. However, contrary to the frequentist approach, the **Bayesian** approach interprets probability as a **degree of belief**. For example, it is believed

from some previous experiments that a head is twice as likely to occur than a tail. Now, the probability of having a head would be $2/3$ as compared to the probability of getting a tail, i.e., $1/3$. This belief before running the experiment is our **prior** belief about the experiment of tossing a coin.

The belief can increase, decrease, or even remain the same if we run this experiment again and again. This example shows that the Bayesian interpretation of probability makes use of previous runs of the experiment to have a degree of belief about any particular experiment. We shall go into the details of these concepts in subsequent chapters of the book.

§ What's in This Book?

This book intends to teach beginners the concepts of statistics using the Python programming language. After completing the book, the readers will learn how to collect, sample, manipulate, and analyze data. They will also perform experiments to explore and visualize a given dataset. The book aims to introduce to the reader the techniques for estimation and inference of valuable parameters of the statistical models.

The book follows a very simple approach. It is divided into nine chapters.

Chapter 1 reviews the necessary concepts of Python to implement statistical techniques and carry out experiments with the data. It also highlights Python libraries that are helpful for statistical tasks.

Chapter 2 presents the basic concepts behind probability that is closely related to the frequency of occurrence of a certain event. Probability theory serves as a foundation for statistics

and data analysis. Moreover, Bayes' theorem that forms the basis for Bayesian statistics is discussed.

Chapter 3 covers the topics of random variables and probability distributions to describe statistical events. Several well-known probability distributions are presented in this chapter.

Chapter 4 provides a succinct introduction to the descriptive statistics which are applicable to both frequentist and Bayesian statistics.

Chapter 5 offers several techniques to explore and visualize discrete and continuous data. Exploratory analysis is performed to reveal features and patterns in the statistical data. The data visualization step is important in any statistical experiment, and it is almost always performed before any statistical estimation or inference technique can be applied to the data.

Chapter 6 introduces the techniques used for inferring or drawing conclusions from the statistical data on the basis of evidence.

Chapter 7 presents the main tasks performed using the frequentist or classical view of statistics. Mostly used statistical techniques are consistent with this view of probability.

Chapter 8 discusses topics in Bayesian statistics that interpret the concept of probability as a degree of belief.

Chapter 9 presents two hands-on projects for the understanding of practical tasks that use statistical data.

§ Background for Reading the Book

This book aims to describe statistical concepts to beginners using Python functions and libraries. It is intended for those

who do not have any previous knowledge of statistics and programming languages. Though programming knowledge is not a pre-requisite for this book, a basic background of programming languages—especially Python—would be helpful in a quick understanding of the ideas presented in this book. Chapter 1 of the book consists of a crash course in Python.

The reader requires a computer equipped with an internet connection to effectively learn the material of this book. Another requirement is to have an elementary knowledge of arithmetic operations such as addition, subtraction, multiplication, and division to understand calculations for numerous statistical techniques.

§ How to Use This Book?

This book presents a number of techniques to understand statistics and the difference between two major types of statistical techniques: frequentist and Bayesian. To facilitate the reading process, occasionally, the book presents three types of box-tags in different colors: **Requirements**, **Further Readings**, and **Hands-on Time**. Examples of these boxes are shown below.

Requirements

This box lists all requirements needed to be done before proceeding to the next topic. Generally, it works as a checklist to see if everything is ready before a tutorial.

Further Readings

Here, you will be pointed to some external reference or source that will serve as additional content about the specific **Topic** being studied. In general, it consists of packages, documentations, and cheat sheets.

Hands-on Time

Here, you will be pointed to an external file to train and test all the knowledge acquired about a **Tool** that has been studied. Generally, these files are Jupyter notebooks (.ipynb), Python (.py) files, or documents (.pdf).

The box-tag **Requirements** lists the steps required by the reader after reading one or more topics. **Further Readings** provides relevant references for specific topics to get to know the additional content of the topics. **Hands-on Time** points to practical tools to start working on the specified topics. Follow the instructions given in the box-tags to get a better understanding of the topics presented in this book.

In each chapter, several techniques have been explained theoretically as well as through practical examples. Each chapter contains exercise questions that can be used to evaluate the understanding of the concepts explained in the chapters. The Python Jupyter Notebooks and the datasets used in the book are provided in the resources.

It is of utmost importance to practice the statistical techniques using Python. To this end, the first chapter provides a crash course on Python. After you get sufficient theoretical background presented in each chapter, it is a good practice to write the code yourself instead of just running the source code provided with this book. The example code is surely helpful in case you are stuck. Furthermore, you are highly encouraged to complete the exercise questions given at the end of each chapter. The answers to these questions have been provided as well at the end of the book.

About the Author



M. Wasim Nawaz has a Ph.D. in Computer Engineering from the University of Wollongong, Australia. His main areas of research are Machine Learning, Data Science, Computer Vision, and Image Processing. Wasim has over eight years of teaching experience in Computer and Electrical Engineering. He has worked with both private and public sector organizations.

Get in Touch With Us

Feedback from our readers is always welcome.

For general feedback, please send us an email at contact@aispublishing.net and mention the book title in the subject line.

Although we have taken extraordinary care to ensure the accuracy of our content, errors do occur.

If you have found an error in this book, we would be grateful if you could report this to us as soon as you can.

If you are interested in becoming an AI Publishing author and if you have expertise in a topic and you are interested in either writing or contributing to a book, please send us an email at author@aispublishing.net.

Download the PDF version

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-sccb>

The order number is required.

1

A Quick Introduction to Python for Statistics

This chapter presents a crash course on Python to kick start the statistical techniques using Python. First, the complete installation of the Anaconda distribution of Python is described for Windows, Mac OS, and Linux operating systems. Second, mathematical operators, functions, control statements, and Python data structures are explained with practical examples. Finally, the most used Python libraries for statistics are presented.

1.1 Installation and Setup of Python Environment

This book utilizes Python 3, the latest release of Python. We may download and install Python from the official website of *Python Software Foundation*, <https://www.python.org/download>. However, we have to install libraries and packages separately when we follow the aforementioned installation.

One of the convenient ways to get started is to install Python using Anaconda distribution. Anaconda is a free and open-source distribution that comes with its own package manager.

Moreover, it includes multiple libraries for Windows, Linux, and Mac OS operating systems. Since libraries are included in this distribution, we do not need to install them separately.

Anaconda Installers		
Windows 	MacOS 	Linux 
Python 3.7	Python 3.7	Python 3.7
64-Bit Graphical Installer (466 MB)	64-Bit Graphical Installer (442 MB)	64-Bit (x86) Installer (522 MB)
32-Bit Graphical Installer (423 MB)	64-Bit Command Line Installer (430 MB)	64-Bit (Power8 and Power9) Installer (276 MB)
Python 2.7	Python 2.7	Python 2.7
64-Bit Graphical Installer (413 MB)	64-Bit Graphical Installer (637 MB)	64-Bit (x86) Installer (477 MB)
32-Bit Graphical Installer (356 MB)	64-Bit Command Line Installer (409 MB)	64-Bit (Power8 and Power9) Installer (295 MB)

Figure 1.1: Anaconda Installers for Windows, Mac OS, and Linux Individual Edition.

We download Anaconda Individual Edition from <https://www.anaconda.com/products/individual>, as shown in Figure 1.1. We select the proper operating system and its version, either 32-bit or 64-b, it from the aforementioned link. The following sections give a step-by-step guide for Windows, Mac OS, and Linux to install Anaconda and set up Python.

1.1.1 Windows

1. Download the graphical Windows installer from <https://www.anaconda.com/products/individual>.
2. Double-click the downloaded file. Next, click **Continue** to begin the installation.
3. On the subsequent screens, answer the *Introduction*, *Read Me*, and *License* prompts.

4. Then, click the **Install** button. Anaconda will install in a specified directory (C:\Anaconda3_Python) given in the installation.



Figure 1.2: Installing Anaconda on Windows.

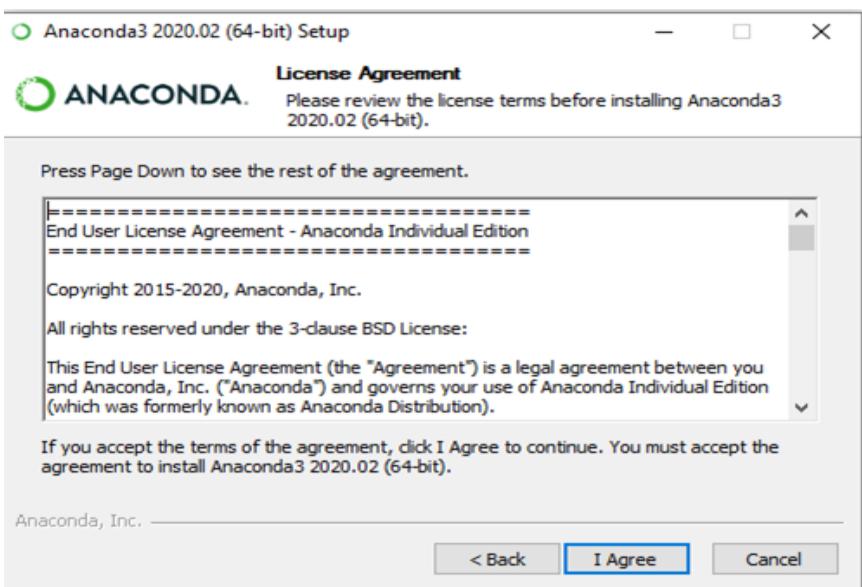


Figure 1.3: Installing Anaconda on Windows.

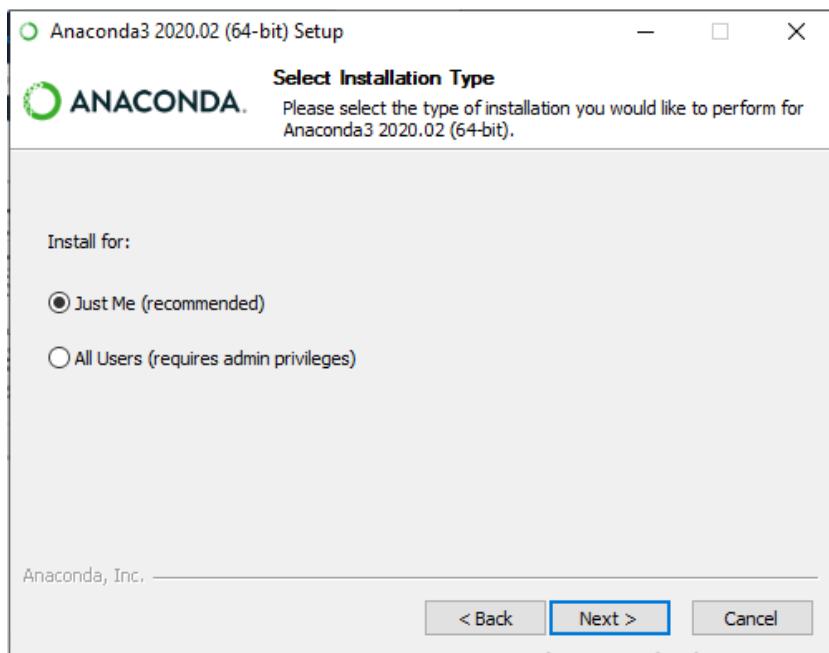


Figure 1.4: Installing Anaconda on Windows.

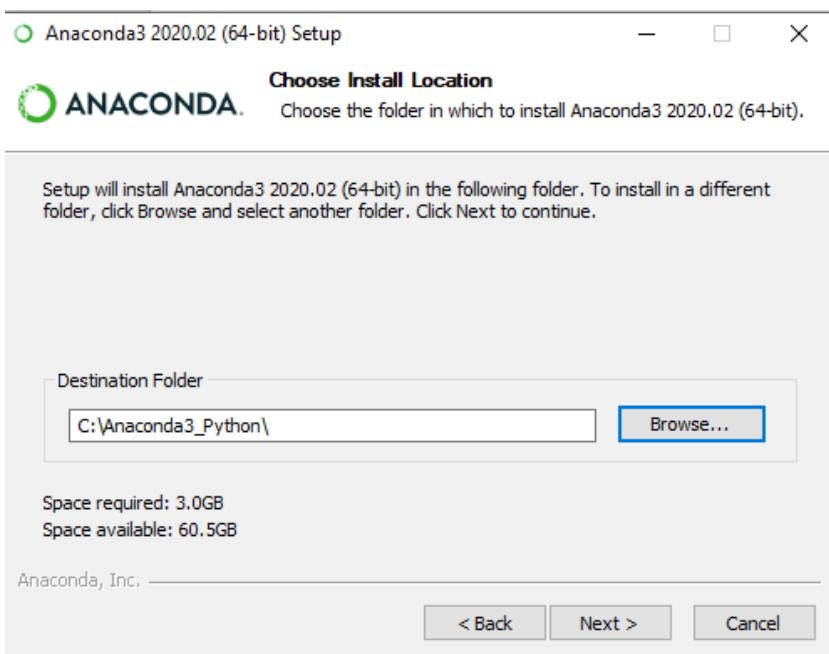


Figure 1.5: Installing Anaconda on Windows.

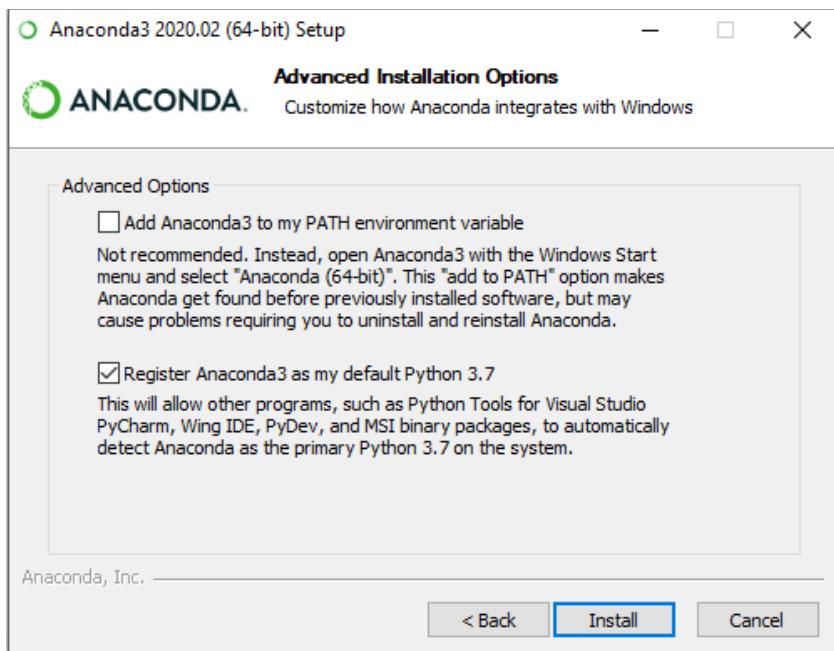


Figure 1.6: Installing Anaconda on Windows.

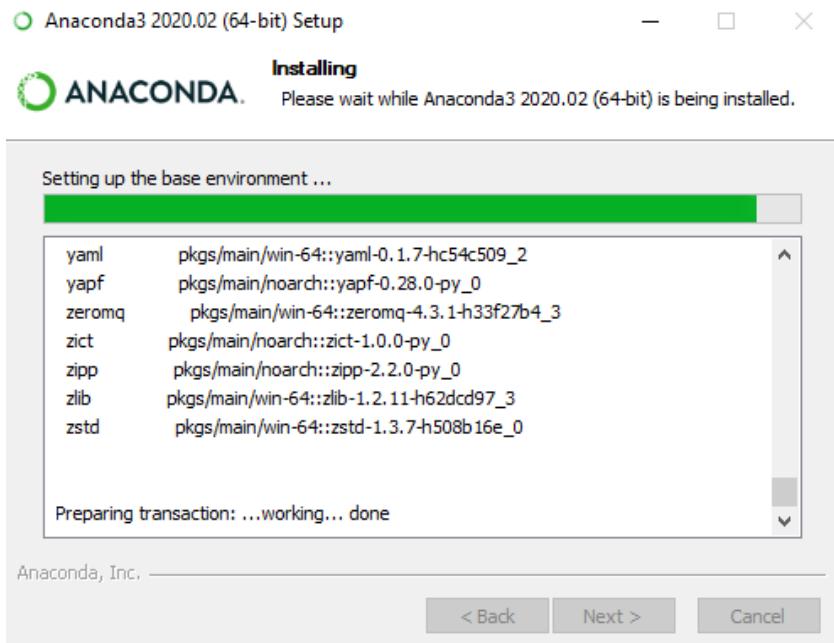


Figure 1.7: Installing Anaconda on Windows.

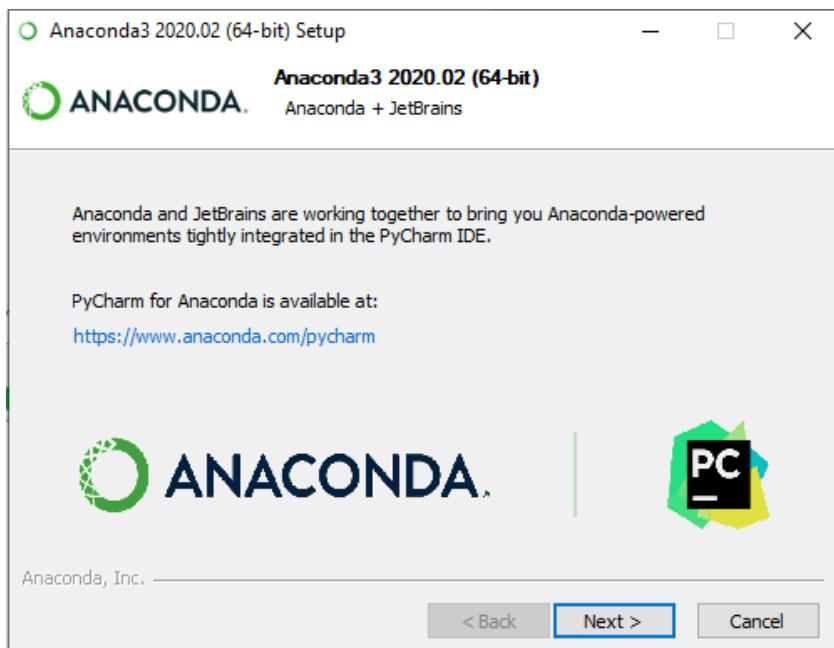


Figure 1.8: Installing Anaconda on Windows.

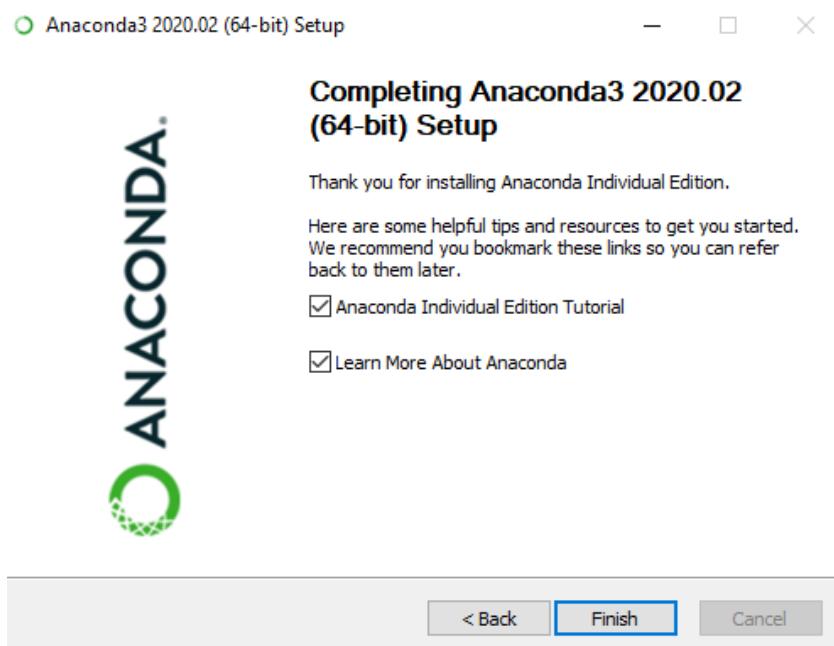


Figure 1.9: Installing Anaconda on Windows.

1.1.2 Apple OS X

1. Download the graphical MacOS installer from <https://www.anaconda.com/products/individual>.
2. Double-click the downloaded file. Next, click **Continue** to begin the installation.
3. Then, click the **Install** button. Anaconda will install in the specified directory.

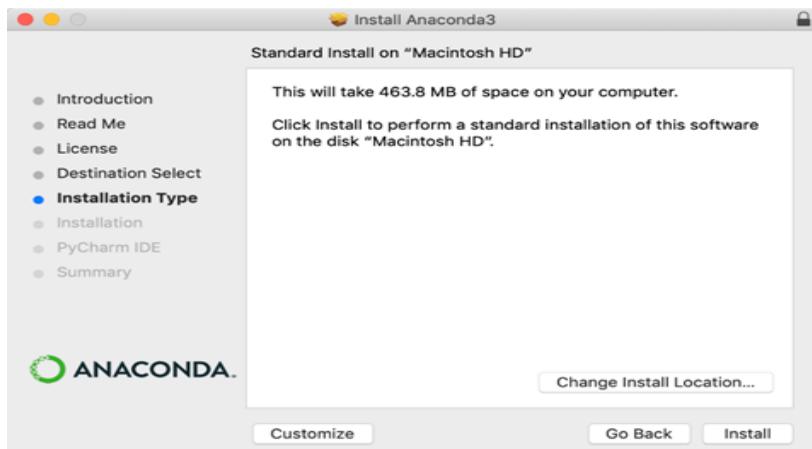


Figure 1.10: Installing Anaconda on Mac OS.



Figure 1.11: Installing Anaconda on Mac OS.

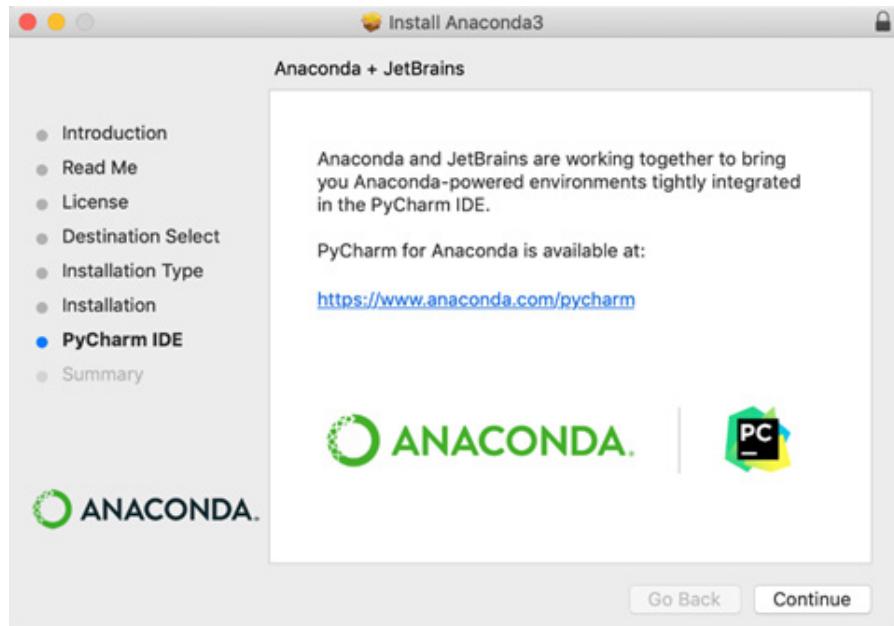


Figure 1.12: Installing Anaconda on Mac OS.

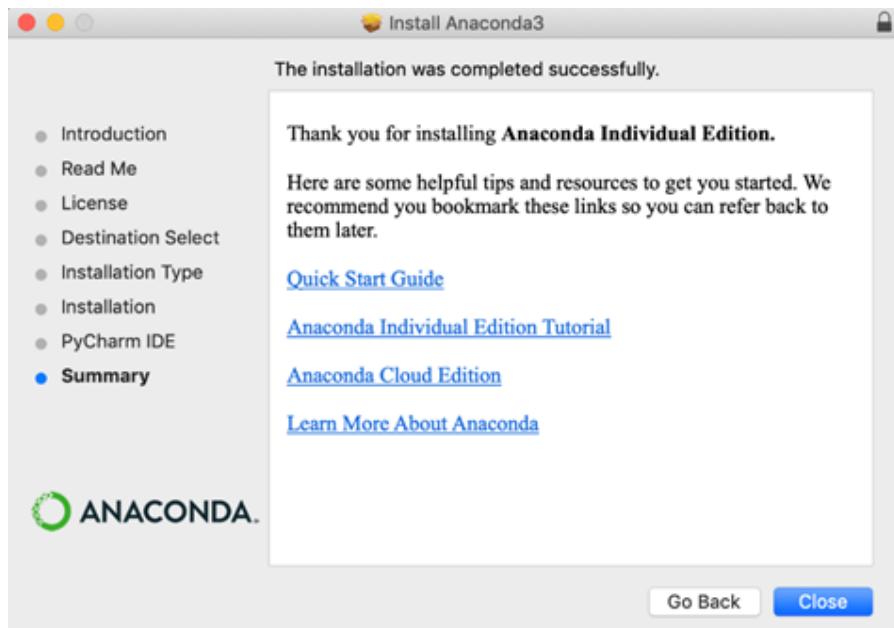


Figure 1.13: Installing Anaconda on Mac OS.

1.1.3 GNU/Linux

Since graphical installation is not available, we use the Linux command line to install Anaconda. A copy of the installation file is downloaded from <https://www.anaconda.com/products/individual>.

We follow the procedure mentioned below for Anaconda installation on a Linux system.

1. Open a copy of Terminal on Linux.
2. Change directories on the system to the downloaded copy of Anaconda.
3. The name of the file normally appears as *Anaconda-3.7.0-Linux-x86.sh* for 32-bit systems and *Anaconda-3.7.0-Linuxx86_64.sh* for 64-bit systems. The version number appears in the filename. In our case, the filename refers to version 3.7, which is the version used for this book.
4. Type `bash ~/Downloads/Anaconda3-2020.02-Linux-x86.sh` (for the 32-bit version) or `bash ~/Downloads/Anaconda3-2020.02-Linux-86_64.sh` (for the 64-bit version) and press **Enter**.
5. An installation wizard opens up and asks you to accept the licensing terms for using Anaconda.
6. Accept the terms using the method required for the version of Linux.
7. The wizard asks you to provide an installation location for Anaconda. Choose a location, and click **Next**.
8. The application extraction begins. A completion message pops up once the extraction is complete.

1.1.4 Creating and Using Notebooks

Once Python installation is complete, we start exploring its features and writing code to perform tasks. We launch Jupyter Notebook accompanied by Anaconda installation. The Jupyter Notebook, a web application, allows us to create Python code, plots, visualizations, and equations. It can be launched by

- the Anaconda Navigator by searching it in *Windows Search Box*. Open Jupyter Notebook, as shown in Figure 1.14.
- writing *Jupyter Notebook* in *Windows Search Box*, as shown in Figure 1.15.

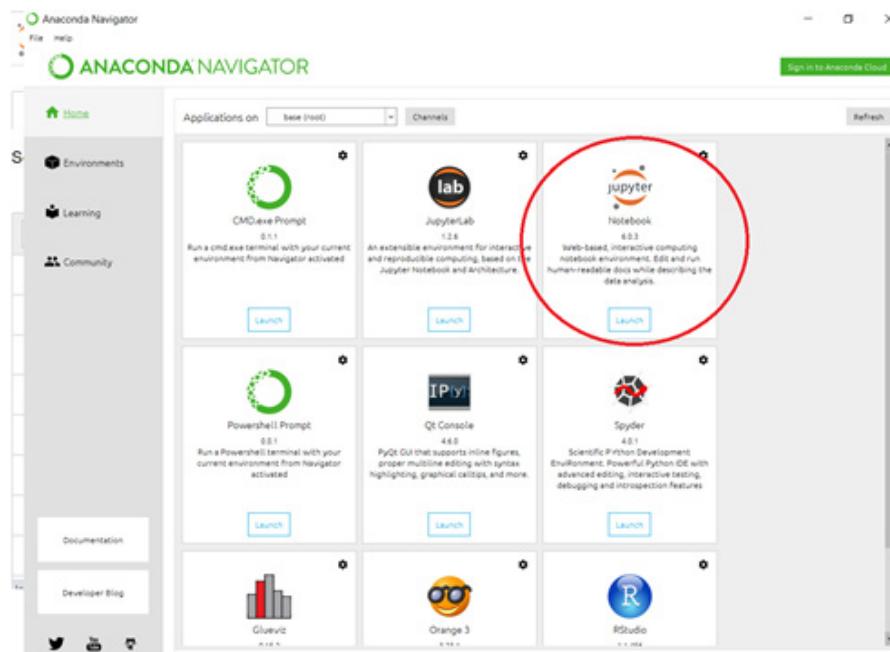


Figure 1.14: Launching the Jupyter Notebook using Anaconda Navigator.

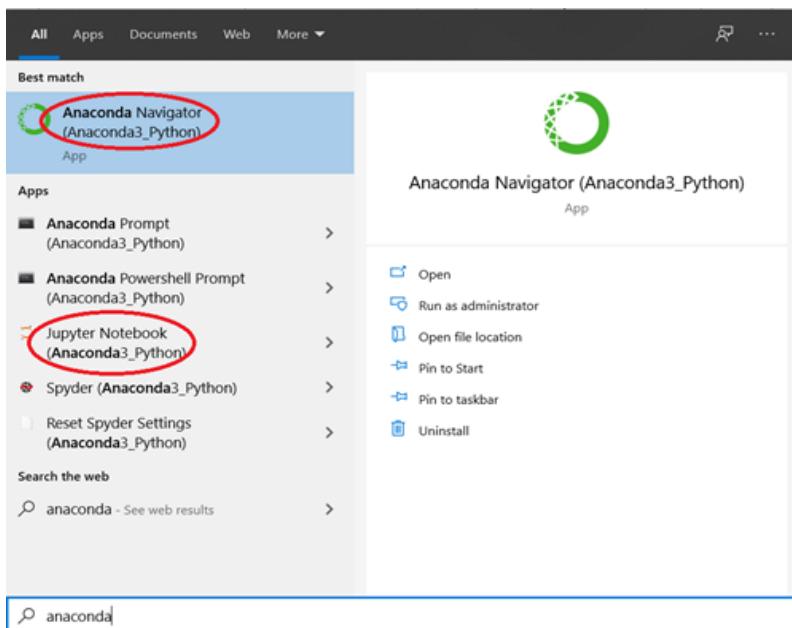


Figure 1.15: Launching Jupyter Notebook from the Windows search box.

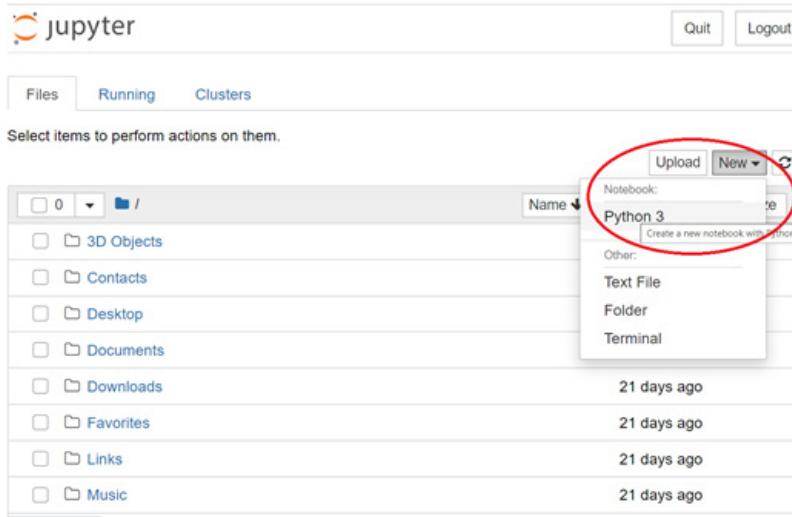


Figure 1.16: Creating a new Python 3 file in Jupyter Notebook.

Jupyter Notebook opens in a new browser page, as shown in Figure 1.16. To create a new notebook, go to **New** on the top right side of the page and select **Python 3**.

The box highlighted in the bottom of Figure 1.17 has In []: written next to it. This is the place where we can start typing our Python code. The Notebook can be given a meaningful name by clicking on **Untitled1** next to the Jupyter icon in the top left corner of the Notebook. Make sure the highlighted box in the middle of Figure 1.17 is selected to Code.

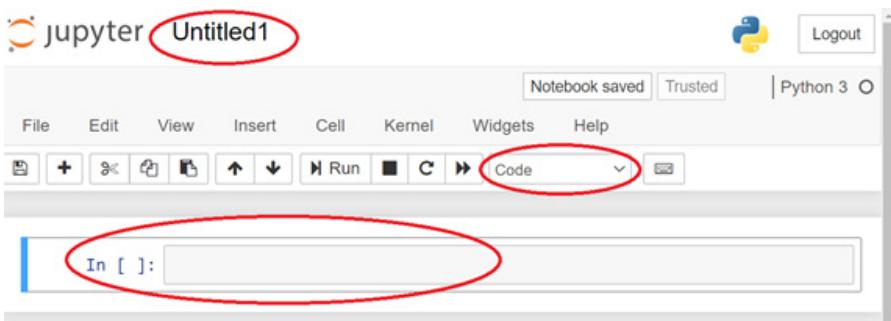


Figure 1.17: Start working with a Jupyter Notebook.

We are now ready to write our first program. Place the cursor in the cell. Type `print("Hello World")`, and click on the **Run** button on the toolbar. The output of this line of code appears on the next line, as shown in Figure 1.18.

```
In [1]: print("Hello World")
Hello World
```

Figure 1.18: Output of the print statement in Python.

The shortcut to run the code present in a cell is to hit **shift+enter** keys together. The Notebook shows the result of running the code right below the cell. A new cell is automatically created for the next commands / piece of code to be entered.

Besides code, Jupyter Notebook's cells can be used to enter text to elaborate the code. For this purpose, we use markdown cells. A markdown cell displays text and mathematical

equations formatted using markdown language. To convert to a markdown cell, we click on the cell menu and select *Markdown*, as shown in Figure 1.19. The In [] prompt will disappear to signify that the text written in the cell is not an executable Python code.

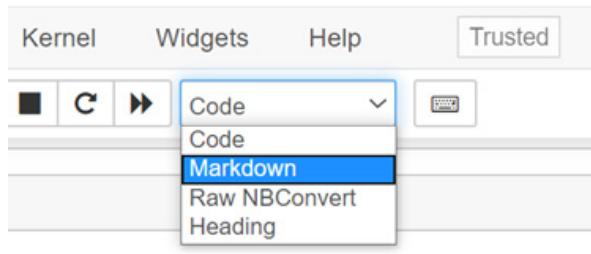


Figure 1.19: A markdown cell

Python is an interpretable language: Python code is executed on a Python virtual machine, one line at a time. The interpreter acts as a calculator. We can type an expression in the cell, and it will return us the output value. The significance of an interpreted language such as Python is that there is an abstraction between the code and the platform. This makes Python code portable across different platforms. For example, the same code running on Windows can also run on Linux.

We can download and save a Notebook by clicking on the *File* dropdown menu, selecting *Download as*, and clicking on Notebook (.ipynb), as shown in Figure 1.20. The downloaded files can be used in the future by going to the *File* dropdown menu and clicking open.

The code we generate for the tasks presented in this book resides in a repository on the hard drive of the computer system. A repository is a kind of filing cabinet or folder where we save our code. We can modify and run individual pieces of code within the folder and add new code to the existing one.

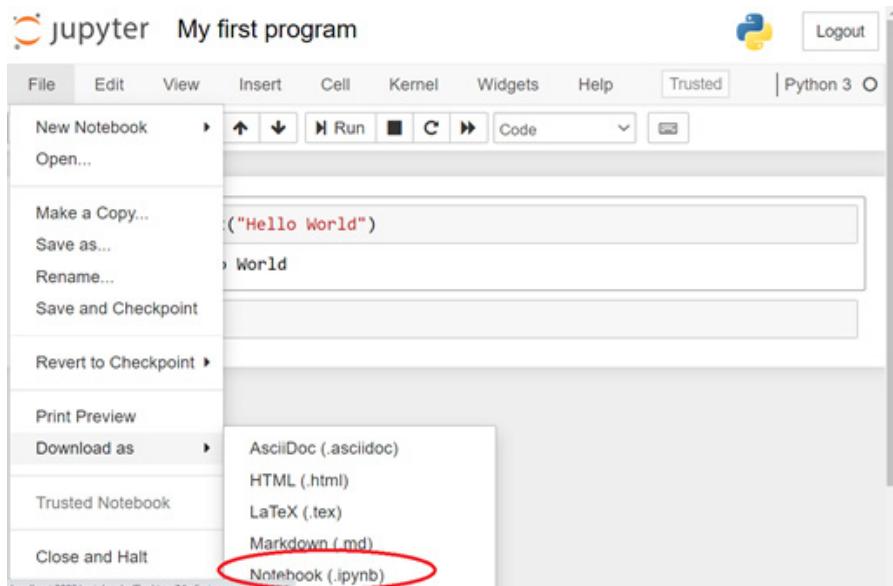


Figure 1.20: Saving a Jupyter Notebook for future use.

1.2 Mathematical Operators in Python

Data comes in numerous forms, such as numbers, logical values, special symbols, and text. Numbers can be integers or floating-points, whereas the logical values can be true or false that can be used to make decisions. For example, to find out whether one quantity is greater than another one, we may use logical values.

The arithmetic operators in Python are symbols which are used to perform arithmetic operations, for instance, addition, subtraction, multiplication, and division. Similarly, logical operators are used to perform logical operations. First, we discuss the data types used by Python to represent numbers and logical values. Next, we discuss arithmetic and logical operators.

Python offers numerous data types to represent numbers and logic values. These are given below.

Integer: An integer is a whole number without any fractional part. For instance, 5 is a whole number; it is an integer. Conversely, 5.0 is not an integer because it has a decimal part. Integers are represented by the data type **int** in Python.

Float: A floating-point number contains a decimal part. For example, 3.5 is a floating-point number. Python stores floating-point values in the data type **float**.

Complex: A complex number comprises of paired numbers: a real number and an imaginary number. The imaginary part of a complex number always appears with a *j*. If we create a complex number with 2 as the real part and 8 as the imaginary part, we make an assignment like this:

```
1. cmplx_no = 2 + 8j
```

Bool: Logical arguments require Boolean values represented by data type **Bool** in Python. A variable of type Bool can either be **True** or **False**. The first letter of both keywords is capital. We can assign a Boolean value to any variable using the keywords True or False as

```
1. x = True  
2. type(x) # it returns the type of x as bool.
```

Alternatively, to define a Bool variable, we create an expression that defines a logical idea. As an example,

```
1. bool_variable = 8 < 4
```

returns False because 8 is not smaller than 4.

1.2.1 Arithmetic Operators

To perform addition, subtraction, multiplication, and division, we use arithmetic operators +, -, *, and /, respectively. Arithmetic operations can be combined using parentheses (). To understand the working of these operators, type the following code.

```
1. 3+10-6
Output:
7
1. 30-5*8
Output:
-10
1. (30-5*8)/5
Output:
-2
1. 20 / 8 # division returns a floating-point number
Output:
2.5
```

The integer numbers, e.g., the number 20 has type int, whereas 2.5 has type float. To get an integer result from division by discarding the fractional part, we use the // operator.

```
1. 20 // 6
Output:
3
```

To calculate the remainder, we use the % operator.

```
1. 20%
Output:
2
```

To calculate powers in Python, we use the ** operator.

```
1. 3**4
Output:
81
```

Operations involving int and float type operands return the output having the data type float.

```
1. type(5 * 4.6)
```

Output:

float

Python supports complex numbers. It uses the suffix j or J to indicate the imaginary part of a complex number. To create a complex number, we type the following command.

```
1. 2+7j
```

Output:

(2+7j)

1.2.2 Bitwise Operators

To comprehend bitwise operations, we have to understand how data is stored and processed in computers as binary numbers. A bit is a binary digit 0 or 1. A computer represents every number as a series of 0s and 1s in its memory. For instance, the decimal number 5 equals 0000 0101 in binary when we use 8 bits to represent a binary number in a computer.

Negative numbers are represented in computers with a leading 1 on the left side instead of a leading 0. This procedure has two steps:

1. Invert individual bits of the number (this is known as taking 1's complement of the number). Operator ~ is used to take 1's complement.
2. Adding a 1 to 1's complement (this is known as taking 2's complement of the number).

For example, decimal -4 can be converted to binary by first taking 1's complement of 4 (0000 0100) that results in 1111 1011 in binary. Now adding a 1 to 1111 1011 results in 1111 1100 that is a

binary representation of -4. To take the negative of number 4 in Python, we may type:

```
1. # ~4 calculates 1's complement, 1 is added to it to get  
2. ~4+1  
Output:  
-4
```

Bitwise operators operate on individual bits of the operands. Let the operands be

- $x = 3$ (0000 0011 in binary) and
- $y = 9$ (0000 1001 in binary).

We discuss bitwise operators and apply them to the aforementioned operands.

Bitwise AND(&): It returns 1 if corresponding bits of x and y are 1; otherwise, it returns 0.

```
1. x & y  
Output: 1
```

Bitwise OR(|): It returns 1 if any of the corresponding bit of x or y is 1; otherwise, it returns 0.

```
1. x | y  
Output:  
11
```

Bitwise NOT(~): It returns $-(x+1)$ for a variable x that is 1's complement of x. In other words, it inverts all the bits of x.

```
1. ~x  
Output:  
-4
```

Bitwise XOR(^): It returns 1 if only one of the corresponding bits of x or y is 1; otherwise, it returns 0.

```
1. x ^ y
```

Output:

```
2
```

Bitwise right shift(>>): It shifts the bits of the operand toward the right by the amount specified in the integers on the right side of the operator. Specifying a float on the right side of the operator >>, for example, $y>>3.5$, gives an error message.

```
1. y >> 2
```

Output:

```
2
```

Bitwise left shift(<<): It shifts the bits of the operand toward left by the amount specified in the integers on the left side of the operator. Specifying a float on the right side of the operator <<, for example, $y<<3.5$, gives an error message.

```
1. x << 2
```

Output:

```
12
```

1.2.3 Assignment Operators

A *variable* is used to store the operands and results of the operations performed on these operands. It can be considered as a container that holds our information. We store our results in a Python Notebook using variables. We make an assignment to a variable to store our data. Various assignment operators supported by Python are given below. The equal sign (=) is used to assign a value to a variable.

```
1. marks1 = 75
```

```
2. marks2 = 85
```

```
3. marks1 + marks2
```

Output:

```
160
```

Variable names in Python are case-sensitive. It means that variable name *height* and *Height* are not the same.

If a variable is not assigned any value (not defined), we get an error when we try to use it:

```
1. Marks1
```

Output:

```
NameError: name 'Marks1' is not defined
```

The following code assigns value 10 to `my_var`, then adds value 20 to it. Thus, the result stored in `my_var` becomes 30.

```
1. my_var = 10
```

```
2. my_var+=20
```

```
3. my_var
```

Output:

```
30
```

In the aforementioned code, we have used `+=` as an assignment operator in `my_var+=20` that is equivalent to `my_var=my_var+20`. Other arithmetic operators, such as `-`, `*`, `/` etc., can be used along with `=` as assignment operators. For example, try the following code to observe the output.

```
1. my_var1 = 10
```

```
2. my_var2 = 4
```

```
3. my_var1**= my_var2
```

```
4. my_var1
```

Output:

```
10000
```

1.2.4 Logical Operators

Logical operators work on logical operands that can assume one of two values: True and False. The logical operators used in Python are given below.

Logicaland: It returns True only when both the operands are true.

1. `x=True`
2. `y=False`
3. `x and y`

Output:

`False`

Logicalor: It returns True if either of the operands is true.

1. `x=True`
2. `y=False`
3. `x or y`

Output:

`True`

Logicalnot: It complements the operand.

1. `x=True`
2. `not x`

Output:

`False`

1.2.5 Comparison Operators

Comparison operators are used to find the relationship between two operands if they are equal, or one of the operands is greater than another operand, or vice versa. These operators return either True or False, and they are extensively employed to make decisions in programming. The details of comparison operators are given below.

Let `x = 6` and `y = -8` be the operands.

Equal==: It checks if both operands `x` and `y` are equal.

1. `x==y`

Output:

`False`

Not equal!=: It checks if both operands x and y are not equal.

```
1. x!=y
```

Output:

True

Greater than>: It checks if one operand is greater than the other operand.

```
1. x > y
```

Output:

True

Less than<: It checks if one operand is smaller than the other operand.

```
1. x < y
```

Output:

False

Greater than or equal to>=: It checks if one operand is greater than or equal to the other operand.

```
1. x >= y
```

Output:

True

Less than or equal to<=: It checks whether one operand is less than or equal to the other operand.

```
1. x <= y
```

Output:

False

1.2.6 Membership Operators

These operators are used to find whether a particular item or set of items are present in a collection or not. The membership operators are **in** and **not in**.

They are used to test whether a value or variable is found in a sequence such as string, list, tuple, set, and dictionary. These operators are used in the following example.

```
1. x = 'Hello world' # x is a string of characters
2. print('H' in x)   # returns True if 'H' is in x
3. print('Hello' not in x) # returns True if 'Hello' is not
   present in x
4.  
Output:
True
False
```

1.3 String Operations

Text data includes numbers, alphabets, spaces, and special characters such as comma, full stop, and colon. Python represents text using a *string* data type. It provides us with rich resources to manipulate strings.

Since computers work on binary numbers, they convert string characters into numbers using a standard called *Unicode*. This number is then stored in memory. For instance, the letter A is converted to its Unicode 65. Type `ord("A")` in the Jupyter cell and press **Enter** to observe 65 as an output. If a decimal number is represented as text, its Unicode value will be different from that number. For example, Unicode for number 9 is 57. In Python, strings are specified either by:

- Enclosing characters in single quotes ('...') or
- Enclosing characters in double quotes ("...")

In Python, a string is an array of bytes representing Unicode characters. A single character is a string with a length of 1. Type the following code to observe the output.

```
'An example string in single quotes.'
```

Output:

```
'An example string in single quotes.'
```

```
"A string in double quotes."
```

Output:

```
' A string in double quotes.'
```

Multiple strings can be specified and printed using the print statement. A comma is used to separate different strings. Try the following code to print multiple strings.

```
print('Mango', 'and apple', 'are fruits')
```

Output:

```
Mango and apple are fruits
```

We get an error message when we try to print a single quote in a string enclosed in '...'. The same error message is displayed if we try to print a double quote in a string enclosed in "...".

```
print('Why don't we visit the museum?')
```

Output:

```
File "<ipython-input-96-868ddf679a10>", line 1
    print(' Why don't we visit the museum?')
    ^
SyntaxError: invalid syntax
```

To print such characters, we use backslash \, called the escape character.

```
1. print('Why don\'t we visit the museum?.')
```

Output:

```
Why don't we visit the museum?
```

The print() function omits the enclosing quotes in the output. Special tasks can be performed using escape characters. For example, \t generates a tab, and \n produces a new line within the print statement.

```
print('Why don\'t we visit \t the museum?')
```

Output:

Why don't we visit the museum?

```
print(' Why don\'t we visit \n the museum?')
```

Output:

Why don't we visit
the museum?

Suppose we want to display the path of a directory such as C:\new_directory using a print statement. We will get a new line because \n in the directory name produces a new line. To ignore the new line, we use the letter r at the start of the directory name as follows:

```
print(r'c:\new_directory')
```

Output:

c:\new_directory

The operator + can be used to concatenate strings together.

1. mystr1 = 'Statistics'
2. mystr2 = 'using'
3. mystr2 = 'Python'
4. mystr1 + mystr2+ mystr3

Output:

'Statistics using Python'

§ String Indexing

The string elements, individual characters, or a substring can be accessed in Python using positive as well as negative indices. Indexing used by Python is shown in Figure 1.21.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	E	I	I	o		w	o	r	I	d
0	1	2	3	4	5	6	7	8	9	10

Figure 1.21: Access elements of a string using positive and negative indices.

Square brackets are used to access string elements. For instance:

```
1. x = "This is a test string"  
2. print(x[0])
```

Output:

```
T
```

The indices start from 0 in Python. To access the last character of the string, we may write:

```
print(x[-1])
```

Output:

```
g
```

A substring can be extracted from a string using indexing. This process is known as **string slicing**. If x is a string, an expression of the form x[i1:i2] returns the x from position i1 to just before i2. A range of indices is specified to access multiple characters of a string.

```
1. print(x[0:7])
```

Output:

```
This is
```

The characters of x from index 0 to 6 are returned by the aforementioned statement. The index 7 is not included. To get

string characters from a specific index to the last index, we may type the following statement.

```
print(x[8:len(x)])
```

Output:

```
a test string
```

The string slice begins from the start of the string even if we omit the first index: `x[:i2]` and `x[0:i2]` gives the same result.

If we want to get every kth character in the string, we specify a **stride** or a **step** in the string indexing. A stride or a step specifies the size of the step between string characters. A stride is set if we add an additional colon in string indexing. For example, to retrieve every second character of the string `x` starting from index 1 to index 11, we specify a step of 2 as follows:

```
1. x = 'This is a test string'
2. x[1:12:2]
```

Output:

```
'hsi e'
```

If we use a negative stride, we step backward through the string. A negative stride requires the first index to be greater than the second one. To retrieve string characters from the last index to the index1, we may type the following.

```
1. x = 'This is a test string'
2. x[:0:-2]
```

Output:

```
'git stas i'
```

Note that the first character of string `x`, T, is not displayed because we have specified a range of indices from end to 1 (index 0 is not included). We get a different output when we write the following script.

```
1. x = 'This is a test string'  
2. x[::-2]  
Output:  
'git stas iT'
```

To replicate a string multiple times, we use the operator `*`. To generate five copies of the string ‘Hello,’ type:

```
1. str = 'Hello ' * 5  
2. str  
Output:  
'Hello Hello Hello Hello Hello '
```

In Python, strings can be manipulated using numerous built-in methods (functions). The method **strip()** eradicates any whitespace from the beginning or the end of a string.

```
1. str1 = " This is a text string "  
2. print(str1.strip())  
Output:  
This is a text string
```

To convert a string to lower case, we use the method **lower()**.

```
1. str2 = "This is a text string"  
2. print(str2.lower())  
Output:  
this is a text string
```

To convert a string to upper case, we use the method **upper()**.

```
1. str3 = "This is a text string"  
2. print(str3.upper())  
Output:  
THIS IS A TEXT STRING
```

To replace the characters of a string, we use the method **replace()**.

```
1. str4 = "I ate a banana"  
2. print(str4.replace("banana", "mango"))  
Output:  
I ate a mango
```

To break a string into smaller strings, we use the method **split()** along with a specified separator.

```
1. str5 = 'Hi mate, how do you do?'  
2. print(str5.split(','))  
Output:  
['Hi mate', 'how do you do?']
```

In the aforementioned code, we have used a comma as a separator. Other characters can also be specified to serve as separators.

1.4 Conditional Statements and Iterations

Conditional statements are used when we need to make decisions in our program. A block of code is executed only if a certain condition is satisfied, and a different piece of code is run otherwise. Conditional statements provided by Python are:

1. if,
2. elif, and
3. else.

If statements can be used standalone, but **elif** and **else** statements are always accompanied by a corresponding **if** statement. We give the details and usage of these statements in the following sections.

1.4.1 If, Elif, and Else Statements

If statement is the simplest conditional statement. The syntax, proper arrangement of symbols, and keywords to constitute a correct programming command of an if statement is:

S if(condition):

 Statement1

Note that the line following the if(condition): is indented. We may specify more indented statements after the Statement1. Statements indented the same amount after the colon (:) are part of the if statement. These statements run when the condition of the if statement is True. In case this condition is False, Statement1 and other indented statements will not be executed. Figure 1.22 shows the flowchart of an if statement.

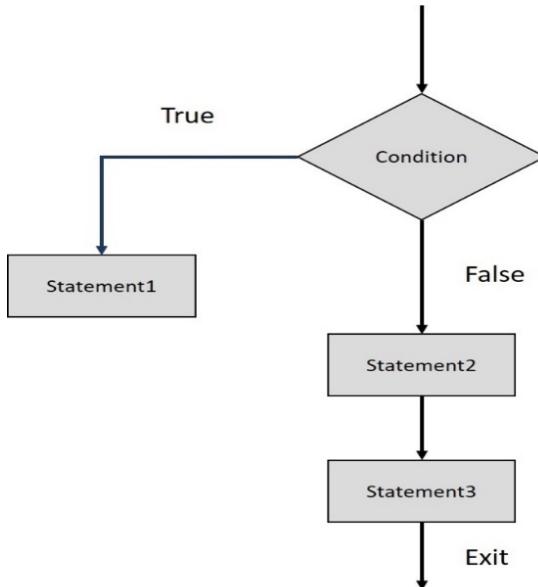


Figure 1.22: Flow chart of an if statement.

For instance, we want to record and display the marks a student obtained in a particular subject. The total marks obtained

should not exceed 100. The program should display an error or warning when the marks are greater than 100. The Python script incorporating an if statement would be as follows.

```
1. student_marks = 90
2. if(student_marks > 100):
3.     print("Marks exceed 100.")
```

The colon (:) after if(student_marks>100) is important because it separates the condition from the statements. The condition inside if(condition) is evaluated. It returns True if the condition is fulfilled and False otherwise. Arithmetic, logical, and conditional operators can be employed to design a condition.

An input from a user can also be used to form a condition. For instance, a user of the program can be asked to enter student_marks that can be used inside an if statement.

```
1. print('Enter marks obtained by a student')
2. student_marks = int(input())
3. if(student_marks > 100):
4.     print("Marks exceed 100.")
```

Note that the input() function gets the input from the user; it saves the input as a string. We use int(input()) to get an integer from the string. If we execute the aforementioned program and input marks are greater than 100, we get a warning “Marks exceed 100”. If the entered marks are 100 or less, no warning message is displayed.

S else Statement

The statement **else** is always accompanied by an accompanying if statement, i.e., **if-else**. The syntax of if-else is given below.

if(condition):

 Indented statement(s) when the condition is True

else:

Indented statement(s) when the condition is False

In our previous examples on student marks, let us display the message “Outstanding” if student_marks are 80 or greater. A message “Not outstanding” is displayed otherwise.

```
1. print('Input marks of a student')
2. student_marks = int(input())
3. if(student_marks >= 80):
4.     print("Outstanding")
5. else:
6.     print("Not outstanding ")
```

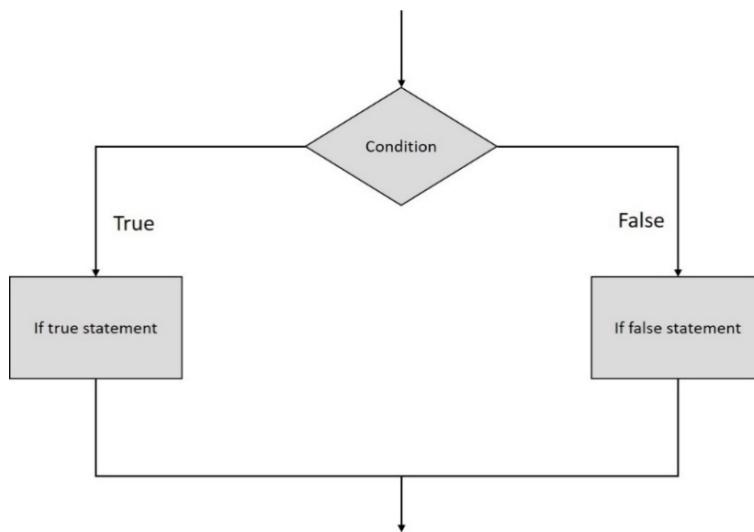


Figure 1.23: Flow chart of an if-else statement.

Figure 1.23 presents the flowchart of an if-else statement. It can be observed that one of two blocks of code will be executed based upon the condition to be evaluated.

§ Nested Decisions

To make decisions under multiple conditions, Python allows us to perform nested decisions. The **if-elif-else** statements

are employed to accomplish nested decisions. Continuing with the example of student marks, if the marks of a student are greater than 100 or less than 0, a warning should be displayed. Additionally, if the obtained marks are 80 or greater, *Outstanding* should be displayed. Otherwise, *Not outstanding* should be displayed.

```

1. print('Enter marks of a student')
2. student_marks = int(input())
3. if(student_marks > 100 or student_marks < 0):
4.     print("Invalid marks.")
5. elif(student_marks >= 80):
6.     print("Outstanding")
7. else:
8.     print("Not outstanding")

```

Note that we have used a logical operator or to combine two conditions together inside the if statement.

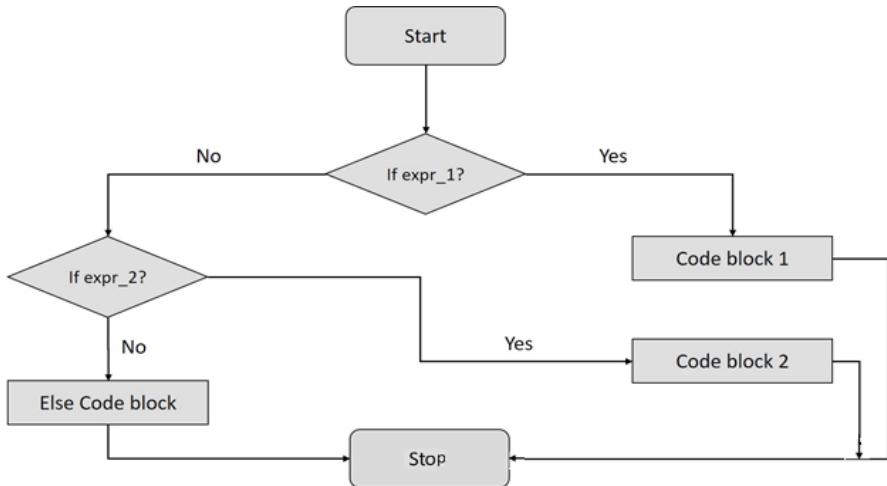


Figure 1.24: Flow chart of an if-elif-else statement.

Figure 1.24 shows the flowchart of an if-elif-else statement. It can be observed that one block of code will be executed based on the condition to be evaluated.

Further Readings

More information about conditional statements can be found at <https://bit.ly/38exHbQ>

1.4.2 For Loop

Iteration statements provided to us by Python allow us to perform a task more than once. A for loop is used to iterate a task a fixed number of times. For loop has a definite beginning and end. We provide a sequence or a variable to the loop as an input that causes the loop to execute a fixed number of times. The syntax of a for loop is given as.

for loop_variable in sequence:

Statement(s) to be executed in the for loop

The flow chart of a for loop is given in Figure 1.25.

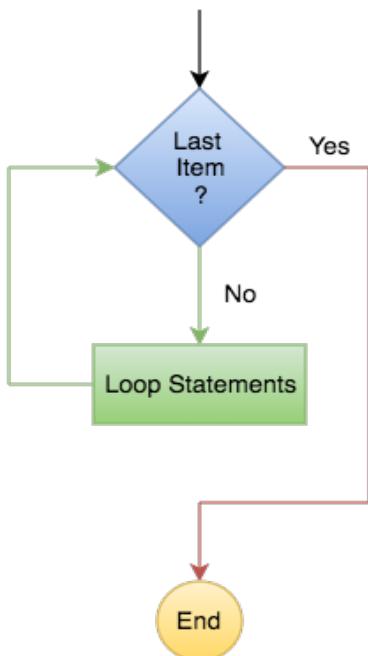


Figure 1.25: Flow chart of a for loop.

Note the Statement(s) to be executed in the body of the for loop are indented. The loop_variable is used inside the for loop, and the number of times the for loop runs depends upon the length of the sequence. To implement the for loop, type the following code.

```
1. subjects = ["Probability", "Statistics", "Machine
   Learning", "Data Science", "Artificial Intelligence"]
2. for k in subjects:
3.     print(k)
Output:
Probability
Statistics
Machine Learning
Data Science
Artificial Intelligence
```

In this example, “subjects” is a variable containing five items. This is used to decide the number of iterations of a for loop. The loop runs five times because the number of items in the variable subjects is five.

The function range() is normally used in a for loop to generate a sequence of numbers. For example, range(5) generates numbers from 0 to 4 (five numbers). The following code generates the first five numbers.

```
1. for k in range(5):
2.     print(k)
Output:
0
123
4
```

We can also specify a step size other than 1 within the range () function as follows.

```
1. for x in range(3, 12, 3):
2.     print(x)
```

Output:

```
369
```

In `range(3, 12, 3)`, 3 is the step size. The statements **break** and **continue** are sometimes used inside the loops. The break statement discontinues the execution of the loop. The continue statement skips all the statements of the for loop following the continue statement. The usage of both statements is illustrated in the following example.

```
1. students = ["Adam", "Alice", "Bob", "Emma", "Julia"]
2.
3. for k in students:
4.     if k == "Bob":
5.         continue
6.     print(k)
7.     if k == "Emma":
8.         break
```

Output:

```
Adam
```

```
Alice
```

```
Emma
```

The name Bob is not printed in the output because the continue statement is executed when the value of k equals Bob. Note that `print(k)` statement is not indented with the if statement. Thus, it is not part of the if statement. Moreover, the code breaks right after it has printed the name, Emma.

1.4.3 While Loop

The while loop iteratively runs certain statements until its condition is fulfilled. The syntax of the while loop is given below.

while (condition):

Statement(s) to be executed in the while loop

For instance, to add natural numbers up to the number input by a user, we use a while loop as follows:

```
1. # This program finds the sum of first n natural numbers,  
  where the value of n is input by the user.  
2.  
3. n = int(input("Input an integer = "))  
4.  
5. # initialize variables sum and j (counter)  
6. sum = 0  
7. j = 1  
8. while j <= n:  
9.     sum = sum + k  
10.    j = j+1    # update the counter variable  
11. print("First", n, "natural numbers add up to ", sum)
```

The user of this program is asked to input a natural number upon execution of this program. A sum of 10 is obtained if the number 4 is entered, and a sum of 55 is returned upon entering the number 10. The while loop allows us to use the break, continue, and else statements inside a while loop like a for loop. Figure 1.26 presents the flow chart of a while loop.

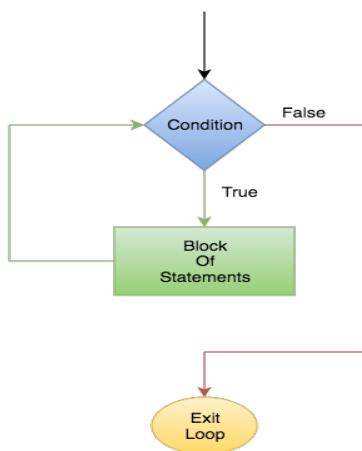


Figure 1.26: Flow chart of a while loop.

§ Nested Loops

A loop, either for or while can be used inside another loop. This is known as nested loops that can be used when we work with the data in two-dimensions. The following program uses two for loops, one nested inside another, to print all the combinations of two variables.

```
1. attributes = ["heavy", "wooden", "fast"]
2. objects = ["chair", "table", "computer"]
3. for j in attributes:
4.     for k in objects:
5.         print(j, k)
```

Output:

```
heavy chair
heavy table
heavy computer
wooden chair
wooden table
wooden computer
fast chair
fast table
fast computer
```

Further Readings

More information about iteration statements can be found at

<https://bit.ly/2TQeW6j>

<https://bit.ly/389E68j>

1.5 Functions in Python

A function is defined as a piece of code that is used to perform a specific task. Functions execute when they are called by their names. Python permits us to define/create functions using the keyword def. For instance, to define a function named my_function1, we write the following code:

```
1. def my_function1():
2.     print("This is a test function")
Output:
This is a test function.
```

To run this function, we type its name.

```
1. my_function1()
Output:
This is a test function.
```

A function can accept one or more inputs by passing parameters or arguments inside the braces (). For instance, the following function accepts a string as its input.

```
1. def my_function2(str_in):
2.     print("This function prints its input that is " + str_
    in)
3.
4. my_function2("computer")
5. my_function2("table")
6. my_function2("chair")
```

Output:

This function prints its input that is computer.
This function prints its input that is table.
This function prints its input that is chair.

If a * is added before the parameter name while defining the function, an input of variable size can be passed as input / argument to the function. Thus, the function is able to receive multiple inputs. The following function elaborates on this idea.

```
1. def my_function3(*myfruits):
2.     print(myfruits[2], "is my favorite fruit.")
3. my_function3("Mango", "Apple", "Orange") # 3 inputs.
```

Output:

Orange is my favorite fruit.

```
1. my_function3("Mango", "Orange", "Apricot", "Apple") # 4
   inputs.
```

Output:

Apricot is my favorite fruit.

Data can be returned from a function when we use the return statement as follows.

```
1. def mult_by_10(k):
2.     return 10 * k
3. print(mult_by_10(2))
4. print(mult_by_10(-8))
5. print(mult_by_10(100))
```

Output:

20
-80
1000

Python allows us to define a function without any name. This is called a **lambda function** (anonymous function), which can receive multiple input arguments but has only one statement. To define a lambda function, we use

lambda arguments : statement

To illustrate the creation and use of a lambda function, we may type the following code:

```
1. x = lambda input_number : input_number - 10
2. print(x(10))
3. print(x(-10))
4. print(x(20))
```

Output:

```
0
-20
10
```

The aforementioned lambda function subtracts 10 from each input number.

1.6 Data Structures

In addition to commonly used data types int, float, and str, Python offers data types to store a collection of multiple entries from numbers, alphabets, alphanumeric, strings, and special characters. The four commonly used collection data types provided by Python are as follows.

- **List** is a mutable and ordered collection that allows duplicate entries.
- **Tuple** is an immutable and ordered collection that also allows duplicate entries.
- **Set** is an unordered and unindexed collection that, like

real sets, prohibits duplicate entries.

- **Dictionary** is a mutable, unordered, and indexed collection of entries that prohibits duplicate entries.

The following sections discuss these data types and their usages.

1.6.1 Lists

A list is a mutable and ordered collection of elements. Lists are specified using square brackets in Python. For instance, to create a list named item_list, type the following code:

```
1. item_list = ["backpack", "laptop", "ballpoint",
   "sunglasses"]
2. print(item_list)
```

Output:

```
[‘backpack’, ‘laptop’, ‘ballpoint’, ‘sunglasses’]
```

To access any particular element/item of a list, we refer its index number. For instance, to print the third item of the list, we type the following code:

```
1. print(item_list[2])
```

Output:

```
ballpoint
```

Since Python permits negative indexing, we can use them to access elements of a list. As an example, the following piece of code prints the third last item of the list.

```
1. item_list = ["backpack", "laptop", "ballpoint",
   "sunglasses"]
2. print(item_list[-3])
```

Output:

```
laptop
```

To return the second and the third list items, we type the following code.

```
print(item_list[1:3])    # Elements at index 1 and 2 but not
the one at 3.
```

Output:

```
['laptop', 'ballpoint']
```

Try the following, and observe the output:

1. `print(item_list[:3]) # returns list elements from the start to “ball point”`
- 2.
3. `print(item_list[2:]) # returns elements from “ball point” to the last element`
- 4.
5. `print(item_list[-3:-1]) # returns elements from index -3 to -1`

The index of a specific element can be used to change the value of that element. For instance, to change the second element of the item_list, type the following:

1. `item_list[1] = “computer”`
2. `print(item_list)`

Output:

```
['backpack', 'computer', 'ballpoint', 'sunglasses']
```

Conditional statements can be used with a list. The elements of a list can be checked using an if statement and the keyword **in** as follows:

1. `if “sunglasses” in item_list:`
2. `print(“sunglasses is present in the list”)`
3. `else:`
4. `print(“sunglasses is not present in the list”)`

Output:

```
sunglasses is present in the list
```

1.6.2 Tuples

A tuple is immutable and ordered collection of items. In Python, tuples are specified using round brackets (). The following code creates a tuple:

```
1. py_stat = ("Python", "for", "statistics")
2. print(py_stat)
Output:
('Python', 'for', 'statistics')
```

Elements of a tuple can be accessed using []. For instance:

```
1. print(py_stat[2])
Output:
statistics
```

Similar to lists, we can use negative indexing and a range of indexing. Since tuples are immutable, the values present in a tuple are not changed once it is created. However, we can use loops to go through the elements of a tuple. The keyword **in** can be used to determine if a specified element is present in a tuple. The method **len()** finds out the number of items present in a tuple. The + operator can be employed to join two or more tuples together. Finally, to delete a tuple, use the statement **del py_stat**.

The tuple method **count()** returns the number of times a particular value appears in a tuple.

```
1. py_stat2 = ('Python', 'has support for', 'statistics')
2. print(py_stat2.count('stat'))
3. print(py_stat2.count('statistics'))

Output:
0
1
```

The tuple method **index()** is used to search the tuple for a particular value. It returns the position where the specified value is found. For example, type the following code:

```
1. print(py_stat2.index('Python'))
2. print(py_stat2.index('probability'))
Output:
2
ValueError: tuple.index(x): x not in tuple
```

A ValueError is printed if the value to be indexed for does not exist in the tuple.

1.6.3 Sets

A set is an unindexed and unordered collection of items. Python specifies sets using curly brackets { }. For example, type the following code.

```
1. my_animals = {"cat", "dog", "tiger", "fox"}
2. print(my_animals)
Output:
{'dog', 'tiger', 'cat', 'fox'}
```

No index is linked to set items because sets are unordered. Nevertheless, a loop can be used to go through the set elements.

```
3. my_animals = {"cat", "dog", "tiger", "fox"}
4. for x in my_animals:
5.     print(x)
Output:
dog
cat
tiger
fox
```

The printed output follows no order. The keyword **in** is used to check if a particular value is present in a set.

```
1. print("tiger" in my_animals)
2. print("lion" in my_animals)
```

Output:

True

False

Once a set is created, its elements cannot be changed. But new elements can be added. The method **add()** adds a single element to a set.

```
1. my_animals = {"cat", "dog", "tiger", "fox"}
2. my_animals.add("lion")
3. print(my_animals)
```

Output:

```
{'lion', 'cat', 'tiger', 'dog', 'fox'}
```

The output of the aforementioned program displays items without any order. The method **update()** adds multiple elements to a set.

```
1. my_animals = {"cat", "dog", "tiger", "fox"}
2. my_animals.update(["sheep", "goat", "sheep"])
3. print(my_animals)
```

Output:

```
{'cat', 'tiger', 'dog', 'goat', 'sheep', 'fox'}
```

The item ‘sheep’ appears once in the output because sets do not allow duplicate entries. To find the number of elements of a set, we use the method **len(my_animals)**.

To remove an element in a set, we use either the method **remove()** or the method **discard()**. For instance, we remove “tiger” as follows:

```
1. my_animals = {"cat", "dog", "tiger", "fox"}
2. my_animals.remove("tiger")
3. print(my_animals)
```

Output:

```
{'dog', 'cat', 'fox'}
```

Two or more sets can be joined together using the method **union()**. Alternatively, the method **update()** can be used to insert elements from one set into another. Let us try the following code:

```
1. myset1 = {"X", "Y", "Z"}  
2. myset2 = {4, 5, 6}  
3. myset3 = myset1.union(myset2)  
4. print(myset3)
```

Output:

```
{4, 5, 6, 'Y', 'X', 'Z'}
```

Moreover, the method **pop()** removes the last item from a set. The method **clear()** empties the set, and the keyword **del** before the name of the set deletes the set completely.

1.6.4 Dictionaries

A dictionary is a mutable, unordered, and indexed collection of items. A dictionary in Python has a key:value pair for each of its elements. Dictionaries retrieve values when the keys are known. To create a dictionary, we use curly braces { }, and put key: value elements inside these braces where each pair is separated from others by commas. For example, the following piece of code creates a dictionary named py_stat_dict.

```
1. py_stat_dict = {  
2.     "name": "Python",  
3.     "purpose": "Statistics",  
4.     "year": 2020  
5. }  
6. print(py_stat_dict)
```

Output:

```
{'name': 'Python', 'purpose': Statistics', 'year': 2020}
```

The dictionaries require the keys to be unique and immutable string, number, or tuple. The values, on the other hand, can be

of any data type and can repeat. Square brackets are used to refer to a key name to access a specified value of a dictionary.

```
1. print(py_stat_dict['name'])      # accesses value for key  
   'name'  
2. print(py_stat_dict.get('purpose')) # accesses value for  
   key 'purpose'
```

Output:

Python

Statistics

A message: *None* is displayed if we try to access a non-existent key.

```
print(py_stat_dict.get('address'))
```

Output:

None

We get the error: **KeyError: 'address'** to indicate that the key 'address' does not exist, when we run `print(py_stat_dict['address'])`. The value of an element can be changed by referring to its key, as shown below.

```
1. py_stat_dict["year"] = 2019  
2. py_stat_dict  
Output:  
{'name': 'Python', 'purpose': 'Statistics', 'year': 2019}
```

A for loop can be used to go through a dictionary; it returns keys of the dictionary.

```
1. for k in py_stat_dict:  
2.     print(k)  
Output:  
name  
purpose  
year
```

The values can be returned as well.

```
1. for k in py_stat_dict:
2.     print(py_stat_dict[k])
```

Output:

```
Python
Statistics
2019
```

The same output is obtained when we use the following:

```
1. for k in py_stat_dict.values():
2.     print(k)
```

To access both keys and values together, we use the method **items()** as follows.

```
1. for x, y in py_stat_dict.items():
2.     print(x, y)
```

Output:

```
name Python
purpose Statistics
year 2019
```

To check if a key exists within a dictionary, we employ a conditional if statement.

```
1. if "year" in py_stat_dict:
2.     print("year' is one of the valid keys")
```

Output:

```
'year' is one of the valid keys
```

We add a new element to a dictionary. For this, a new key is used, and a value is assigned to this key, as given below.

```
1. py_stat_dict["pages"] = 300
2. print(py_stat_dict)
```

Output:

```
{'name': 'Python', 'purpose': 'Statistics', 'year': 2019,
 'pages': 300}
```

The method **pop()** can be used to remove a specific element.

```
1. py_stat_dict.pop("year") # del py_stat_dict["year"] does  
    the same job.  
2. print(py_stat_dict)  
Output:  
{'name': 'Python', 'purpose': 'Data science', 'pages': 300}
```

The keyword **del** removes the whole dictionary when we use **del py_stat_dict**. The method **clear()** deletes all the elements of a dictionary.

```
1. py_stat_dict.clear()  
2. py_stat_dict  
Output:  
{ }
```

The method **len(dictionary_name)** is used to print the number of key:value pairs present in the dictionary.

```
1. py_stat_dict = {  
2.     "name": "Python",  
3.     "purpose": "Statistics",  
4.     "year": 2020  
5. }  
6. print(len(py_stat_dict))  
Output:  
3
```

A dictionary cannot be copied by a simple assignment such as **py_stat_dict2 = py_stat_dict**. It is because `py_stat_dict2` is just a reference to the original dictionary `py_stat_dict`. Whatever changes are made to `py_stat_dict` are automatically made to `py_stat_dict2`, as well. To copy the elements of a dictionary, we use the method **copy()** as follows.

```
1. py_stat_dict2 = py_stat_dict.copy()  
2. print(py_stat_dict)  
3. print(py_stat_dict2)  
Output:  
{'name': 'Python', 'purpose': 'Data science', 'year': 2020}  
{'name': 'Python', 'purpose': 'Data science', 'year': 2020}
```

Further Reading

A detailed tutorial on Python is given in

<https://bit.ly/34WfGwY>

<https://bit.ly/389E68j>

1.7 Python Libraries for Statistics

The Anaconda distribution of Python includes suitable packages and libraries that can readily be used for statistical tasks. The following libraries will be used right through this book:

1. NumPy for mathematical functions,
2. Pandas for data processing,
3. Statistics: Python's built-in module for Statistical functions,
4. Matplotlib for visualization and plotting,
5. SciPy.stats module for basic and advanced statistical functions,
6. Statsmodels for statistical models,
7. PyMC for Bayesian modeling and inference.

All except the PyMC library are pre-installed in Anaconda distribution. We shall discuss the installation of the PyMC in Section 1.7.7. The details of these libraries are given in the following sections.

1.7.1 NumPy for Mathematical Functions

NumPy, or Numerical Python, is a Python library that supports multi-dimensional arrays and matrices. It provides a sizable collection of fast numeric functions to work with these arrays and to perform operations of linear algebra on them.

In most programming languages, including Python, *arrays* are used to store multiple values in a variable. An array is a variable that is able to hold several values. Arrays are commonly used to store statistical data. In standard Python, lists are used as arrays. However, lists are slow to work with. In case speed is an important factor, we use NumPy's array object, also called **ndarray**, that is significantly faster than a list.

To create and use nd arrays, we first need to import the NumPy library. Often, we use an alias to refer to the name of different libraries. NumPy is usually replaced with its defined alias np. To create NumPy arrays, we type the following code.

```
1. import numpy as np  
2. my_arr0 = np.array(20)  
3. my_arr1 = np.array([10, 20, 30, 40, 50])  
4. print(my_arr0)  
5. print(my_arr1)
```

Output:

```
[20]  
[10 20 30 40 50]
```

The scalar values are considered as 0-dimensional arrays. An array of scalar or 0-dimensional values is called a 1-dimensional array. The variable `my_arr0` in the aforementioned program is a 0-dimensional array, whereas variable `my_arr1` is a 1-dimensional array.

We can create a 2-dimensional array by placing 1-dimensional arrays as elements of another array. A 2-dimensional array is used to store and process matrices.

```
1. import numpy as np
2. my_arr2 = np.array([[10, 20, 30, 40], [90, 80, 70, 60]])

3. print(my_arr2)
Output:
[[10 20 30 40]
 [90 80 70 60]]
```

In the above-mentioned example, a matrix of two rows and four columns is created. We can also create three or higher dimensional NumPy arrays as follows:

```
1. import numpy as np
2. my_arr3 = np.array([[[10, 20, 30], [40, 50, 60]], [[70,
   80, 90], [100, 110, 120]]])

3. print(my_arr3)
Output:
[[[ 10  20  30]
  [ 40  50  60]]

 [[ 70  80  90]
  [100 110 120]]]
```

This code generates two matrices of 2 rows and 3 columns each. The following program uses a few of the NumPy mathematical functions on arrays:

```
1. import numpy as np
2.
3. a = np.array([1,2,3])
4. b= np.array([4,5,6])
5.
6. print("Addition of a and b : ",np.add(a,b))
7.
8. print("Multiplication of a and b : ",np.multiply(a,b))
```

```
9.  
10. print("Subtraction of a and b :",np.subtract(a,b))  
11.  
12. print("division of a and b :",np.divide(a,b))  
13.  
14. print("a raised to b is:",np.power(a,b))  
15.  
16. print("mod of a and b :",np.mod(a,b))  
17.  
18. print("remainders when a/b :",np.remainder(a,b))  
19.  
20. a = np.array([3.33,4.55,5.25])  
21.  
22. rounded_a = np.round_(a,2)  
23. print("Rounded array is: ",rounded_a)  
24.  
25. floor_a = np.floor(a)  
26. print("Floor of the array is: ",floor_a)  
27.  
28. print("Square root of the array is: ",np.sqrt(a))
```

Output:

```
Addition of a and b : [5 7 9]  
Multiplication of a and b : [ 4 10 18]  
Subtraction of a and b : [-3 -3 -3]  
division of a and b : [0.25 0.4 0.5 ]  
a raised to b is: [ 1 32 729]  
mod of a and b : [1 2 3]  
remainders when a/b : [1 2 3]  
Rounded array is: [2.33 4.15 5.85]  
Floor of the array is: [2. 4. 5.]  
Square root of the array is: [1.52643375 2.03715488  
 2.41867732]
```

1.7.2 Pandas for Data Processing

Statistical data acquired from different sources is in a raw form that has to be cleaned and prepared for further analysis. Pandas is a library for data preparation and cleaning. Pandas uses mainly two data structures:

1. **Series**, which is like a list of items and
2. **DataFrames**, which acts like a table or a matrix with multiple columns.

Pandas allows data cleaning features such as replacing missing values, joining, and merging data from different sources. We import Pandas and use pd as its alias.

```
import pandas as pd
```

Series: Similar to a 1-dimensional NumPy array, the Series data structure is able to handle 1-dimensional data. However, unlike NumPy arrays, it offers extra features to pre-process data. The constructor Series () is used to create a series object.

```
1. myseries = pd.Series([-2, -1, 0, 1 2]) # note capital S in Series
```

```
2. myseries
```

Output:

```
0   -2  
1   -1  
2    0  
3    1  
4    2  
dtype: int64
```

The last line of the output, dtype: int64, indicates that the type of values of my series is an integer of 64 bits.

A series object contains two arrays: index and value that are linked to each other. The first array on the left of the output of the previous program saves the index of the data, while the second array on the right contains the actual values of the series.

Series objects can be generated using NumPy arrays. Instead of default numeric indices, descriptive indices can be assigned

to the values. The following program shows how to use a NumPy array to generate a series object.

```
1. import pandas as pd
2. import numpy as np
3.
4. series_list = pd.Series([1,2,3,4,5,6,7,8,9])
5. series_np = pd.Series(np.
   array([10,20,30,40,50,60,70,80,90]))
6.
7. print("A Pandas series list:\n",series_list)
8. print("\nA Pandas series using numpy array:\n",series_np)
9.
10. series_index = pd.Series(
11.         np.array([10,20,30,40,50]),
12.         index=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
13. )
14.
15. print("\nA Pandas series with indexing in letters:\n",
   series_index)
```

Output:

A Pandas series list:

```
0    1
1    2
2    3
3    4
4    5
5    6
6    7
7    8
8    9
dtype: int64
```

A Pandas series using a NumPy array:

```
0    10
1    20
2    30
3    40
4    50
5    60
```

```

6    70
7    80
8    90
dtype: int32

A Pandas series with indexing in letters:
a    10
b    20
c    30
d    40
e    50
dtype: int32

```

Line 12 uses the index option in pd.Series () constructor to assign letters as indices to the values.

DataFrame: The second data structure used by Pandas, the DataFrame, is similar to a 2-dimensional NumPy array. The DataFrame contains an ordered group of columns. Every column contains values of numeric, string, or Boolean, etc., types.

To create a DataFrame, we pass a dictionary to the constructor DataFrame(). This dictionary comprises keys with corresponding values. In the Python script given below, a dictionary, namely d, is created. This dictionary is used as an input to the DataFrame () constructor to create a dataframe.

```

1. d = {'one': pd.Series([1.2, 2.3, 3.4], index=
   ['a', 'b', 'c']),
2.      'two': pd.Series([1.5, 2.4, 3.2, 4.1], index=
   ['a', 'b', 'c', 'd'])}
3.
4. df = pd.DataFrame(d)
5. print("Dataframe from a dict of series is:\n",df)
6.
7. df1 = pd.DataFrame(d, index=['d', 'b', 'a'])
8. print("Dataframe from a dict of series with custom indexes
   is:\n",df1)

```

Output:

```
Dataframe from a dict of series is:  
    one  two  
a  1.2  1.5  
b  2.3  2.4  
c  3.4  3.2  
d  NaN  4.1  
Dataframe from a dict of series with custom indexes is:  
    one  two  
d  NaN  4.1  
b  2.3  2.4  
a  1.2  1.5
```

The following Python script makes use of a dictionary object to create a dataframe, namely df. The program also shows how to get data from the dataframe.

```
1. my_dict = {  
2.     'name' : ["a", "b", "c", "c", "e"],  
3.     'age' : [10,20, 30, 40, 50]  
4. }  
5.  
6. df = pd.DataFrame( my_dict,  
7. index = [  
8.     "First",  
9.     "Second",  
10.    "Third",  
11.    "Fourth",  
12.    "Fifth"])  
13.  
14. print("\nThe dataframe is:\n",df)  
15.  
16. series_name = df.name  
17. print("\nThe name series is:\n",series_name)  
18.  
19.  
20. series_age = df.age  
21. #Getting the mean of a Series
```

```
22. print("\nGetting mean value from a series: ",series_age.  
       mean())  
23.  
24. # Getting the size of the Series  
25. print("\nGetting size of a series: ",series_age.size)  
26.  
27. # Getting all unique items in a series  
28. print("\nGetting unique values of a series: ",series_name.  
       unique())  
29.  
30. # Getting a python list out of a Series  
31. print("\nGetting a list of a series: ",series_name.  
       tolist())  
Output:  
The dataframe is:  
      name  age  
First    a   10  
Second   b   20  
Third    c   30  
Fourth   c   40  
Fifth    e   50  
  
The name series is:  
First    a  
Second   b  
Third    c  
Fourth   c  
Fifth    e  
Name: name, dtype: object  
  
Getting the mean value from a series:  30.0  
  
Getting the size of a series:  5  
  
Getting the unique values of a series:  ['a' 'b' 'c' 'e']  
  
Getting a list of a series:  ['a', 'b', 'c', 'c', 'e']
```

1.7.3 Statistics: Python's Built-in Module

Python provides a built-in module with a rich variety of functions to calculate statistics of real-valued data. A brief list of functions provided by this module is given in Table 1.1.

Table 1.1: Statistical functions provided by the Statistics module.

Function Name	Description
mean()	It calculates the average or arithmetic mean of the data.
median()	It is used to calculate median or middle value of the data.
mode()	It calculates the single mode (most common value) of discrete or nominal data.
quantiles()	It divides data into intervals with equal probability.
pstdev()	It calculates the standard deviation of the whole population.
pvariance()	It calculates the variance of the whole population. The variance is the square root of the standard deviation.
stdev()	It calculates the standard deviation of a sample taken from the population.
variance()	It calculates the variance of a sample taken from the population.

There is a slight difference between the standard deviation / variance of a sample and that of the whole population. In the calculation of statistics for a sample, we divide by $(N-1)$ where N represents the number of data points. However, in the case of population variance and standard deviation, we divide by (N) . Therefore, we get slightly different results. This

is illustrated along with other statistics in the following Python script:

```
1. import numpy as np
2. import statistics as st
3.
4. raindata = [2, 5, 4, 4, 0, 2, 7, 8, 8, 8, 1, 3]
5.
6. # Mean
7. print("mean = ", st.mean(raindata))
8.
9. # Median
10. print("median = ", st.median(raindata))
11.
12. # Mode
13. print("mode = ", st.mode(raindata))
14.
15. # population variance
16. print("pvariance = ", st.pvariance(raindata))
17.
18. # population standard deviation
19. print("pstdev = ", st.pstdev(raindata))
20.
21. # variance
22. print("variance = ", st.variance(raindata))
23.
24. # standard deviation
25. print("stdev = ", st.stdev(raindata))

Output:
mean = 4.333333333333333
median = 4.0
mode = 8
pvariance = 7.555555555555555
pstdev = 2.748737083745107
variance = 8.242424242424242
stdev = 2.8709622502610936
```

Note that we get `pstdev = 2.75` and `stdev = 2.87`. The function `st.mode()` gives the data point that occurs the most in the

dataset. This function gives an error in case there is no unique mode.

1.7.4 Matplotlib for Visualization and Plotting

It is often desirable to plot the datasets and its important statistics to get a better understanding of the distribution of the data. Matplotlib is the Python library that provides us with a variety of interactive tools to plot the data. The following example plots a given dataset along with its mean, median, and mode:

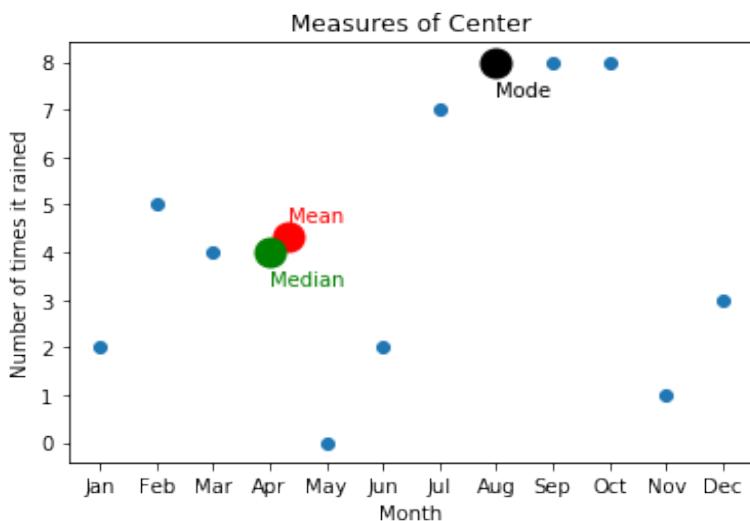
```
1. import matplotlib.pyplot as plt
2. import statistics as st
3.
4. month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
   'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
5. months = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6.
7. fig, ax = plt.subplots(nrows=1, ncols =1)
8.
9. ax.set_title("Measures of Center")
10. ax.set_xlabel("Month")
11. ax.set_ylabel("Number of times it rained")
12.
13. ax.scatter([1,2,3,4,5,6,7,8,9,10,11,12],raindata)
14.
15. plt.xticks(np.arange(12)+1, month_names, color = 'black')
16.
17. # draw points for mean, median, mode
18. ax.plot([st.mean(raindata)], [st.mean(raindata)],
   color='r', marker="o", markersize=15)
19. ax.plot([st.median(raindata)], [st.median(raindata)],
   color='g', marker="o", markersize=15)
20. ax.plot([st.mode(raindata)], [st.mode(raindata)],
   color='k', marker="o", markersize=15)
21.
22. # Annotation
```

```

23. plt.annotate("Mean", (st.mean(raindata),
   st.mean(raindata)+0.3), color="r")
24. plt.annotate("Median", (st.median(raindata),
   st.median(raindata)-0.7), color="g")
25. plt.annotate("Mode", (st.mode(raindata), st.mode(raindata)-
   0.7), color="k")
26.
27. plt.show()

```

Output:



1.7.5 SciPy.stats Module for Statistical Functions

SciPy.stats is a Python module that contains several probability distributions in addition to a large number of statistical functions. To generate summary statistics, we first import `scipy.stats`. The following script prints numerous statistics:

```

1. import scipy.stats as stats
2. import numpy as np
3.
4. # Statistical functions applied on an array of data
5.
6. c = np.array([5, 2, 5, 6, 3, 4, 2, 8, 7])

```

```
7.  
8. all = stats.describe(c)  
9. gmean = stats.gmean(c)  
10. hmean = stats.hmean(c)  
11. mode = stats.mode(c)  
12. skewness = stats.skew(c)  
13. iqr = stats.iqr(c)  
14. z-score = stats.z-score(c)  
15.  
16. print("\nDescribe\n",all)  
17. print("\nGeometric mean\n",gmean)  
18. print("\nharmonic mean\n",hmean)  
19. print("\nMode\n",mode)  
20. print("\nSkewness\n",skewness)  
21. print("\nInter Quantile Range\n",iqr)  
22. print("\nZ Score\n",z-score)
```

Output:**Describe**

```
DescribeResult(nobs=9, minmax=(2, 8), mean=4.666666666666667,  
variance=4.5, skewness=0.12962962962962918,  
kurtosis=-1.1574074074074074)
```

Geometric mean

```
4.196001296532889
```

harmonic mean

```
3.722304283604136
```

Mode

```
ModeResult(mode=array([2]), count=array([2]))
```

Skewness

```
0.12962962962962918
```

Inter Quantile Range

```
3.0
```

Z Score

```
[ 0.16666667 -1.33333333  0.16666667  0.66666667 -0.83333333  
 -0.33333333 -1.33333333  1.66666667  1.16666667]
```

Scipy.stats provides us a large number of random variables and discrete as well as continuous probability distributions.

```

1. import scipy.stats as stats
2. from scipy.stats import norm
3. from scipy.stats import uniform
4.
5. # Generation of random variables
6. a = norm.rvs(size = 5)
7. print (a)
8. b = uniform.cdf([0, 1, 2, 3, 4, 5], loc = 1, scale = 4)
9. print (b)
Output:
[-0.58508956  1.96115496  1.04615577 -0.0938734   0.91089162]
[0.    0.    0.25 0.5  0.75 1.  ]

```

We shall go into the detail of random variables and probability distributions in Chapter 3.

1.7.6 Statsmodels for Statistical models

The Python module statsmodels offers numerous statistical functions to create statistical models and assess their performances. This module also provides a range of statistical tests to explore a given dataset.

In statsmodels, we can specify the details of the statistical model using formulas or array objects. The following is an example to generate a regression model using ordinary least squares (OLS):

```

1. import numpy as np
2. import statsmodels.api as sm
3.
4. # Generate artificial data (2 regressors + constant)
5. nobs = 100
6.
7. X = np.random.random((nobs, 2))

```

```
8. X = sm.add_constant(X)
9.
10. beta = [1, .1, .5]
11.
12. e = np.random.random(nobs)
13.
14. # Dot product of two arrays
15. y = np.dot(X, beta) + e
16.
17. # Fit regression model
18. results = sm.OLS(y, X).fit()
19.
20. # Inspect the results
21. print(results.summary())
```

Output:

OLS Regression Results

```
=====
Dep. Variable:                      y      R-squared:                 0.079
Model:                            OLS      Adj. R-squared:            0.060
Method:                           Least Squares      F-statistic:             4.155
Date:                     Fri, 04 Sep 2020      Prob (F-statistic):       0.0186
Time:                         22:18:38      Log-Likelihood:          -25.991
No. Observations:                  100      AIC:                   57.98
Df Residuals:                      97      BIC:                   65.80
Df Model:                           2
Covariance Type:                nonrobust
=====
            coef    std err        t     P>|t|      [0.025    0.975]
-----
const    1.6532    0.088     18.781     0.000      1.478    1.828
x1      0.0062    0.120      0.052     0.959     -0.233    0.245
x2      0.3266    0.114      2.856     0.005      0.100    0.553
=====
Omnibus:                 71.909      Durbin-Watson:           2.124
Prob(Omnibus):            0.000      Jarque-Bera (JB):        7.872
Skew:                    -0.133      Prob(JB):               0.0195
Kurtosis:                  1.651      Cond. No.                 5.72
=====
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
    specified.
```

The output of the code describes a number of statistics related to the regression model. The reader may not be familiar with a

lot of terms in the output. However, at this stage, it is not very important to understand the minute details of the statistical models. We shall get back to these concepts in the later chapters of the book. However, it is important to realize the powerful features of the statsmodels library.

1.7.7 PyMC for Bayesian Modeling

PyMC3 is a Python package specifically designed for Bayesian statistical modeling and machine learning. This package makes use of Markov chain Monte Carlo (MCMC) and variational algorithms to create / train the statistical models which are hard to create otherwise.

To perform differentiation as frequently required in Bayesian statistics and for computational optimization, PyMC3 depends upon Theano, another Python library.

PyMC is not part of Anaconda distribution. Thus, we have to install it manually. To install, PyMC to Anaconda distribution, we open the Anaconda prompt from the Windows start menu. In the prompt, we type the following command.

```
conda install m2w64-toolchain
```

Once the toolchain is installed, we get a message “done.” Next, we type the following command to install the pymc3, the latest release of the package.

```
conda install -c conda-forge pymc3
```

We also have to install another Python library, Theano, which has dependencies on PYMC3.

```
conda install theano pygpu
```

Depending upon the internet speed, it will take a while to install this package.

If the installed PyMC package does not work, we may need to install the xarray dependency by typing the following command:

```
pip install xarray==0.16.0
```

An essential step to build Bayesian models is the description of a probability model. In this process, we assign statistical distributions that can be described by some parameters to unknown quantities in the model. Therefore, PyMC3 provides wide-ranging pre-defined probability distributions, which are used to build Bayesian models. For instance, to define a variable having Normal or Gaussian probability, we specify it as an instance of the Normal.

```
1. with pm.Model():
2.
3.     x = pm.Normal('x', mu=0, sigma=1)
```

Details of PyMC3 functionality is beyond the scope of this chapter. We go into the details of Bayesian modeling in Chapter 8.

Further Reading

More information about Python and its commonly used functions can be found at
<https://bit.ly/3oQ6LFC>

Hands-on Time

To test your understanding of the concepts presented in this chapter, complete the following exercise.

1.8 Exercise Questions

Question 1:

Which statement is usually used when we have to make a decision based upon only one condition?

- A. If Statement
- B. else Statement
- C. For Loop
- D. Both A and B

Question 2:

Which statement is usually used when we need to iteratively execute a code fixed number of times?

- A. If Statement
- B. else Statement
- C. For Loop
- D. Both A and B

Question 3:

What will be the output if we type `19 / 3`?

- A. 6
- B. 6.33333333333333
- C. 1
- D. Not given

Question 4:

What will be the output if we type `17 // 4`?

- A. 4
- B. 1
- C. 4.25
- D. 68

Question 5:

What will be the output if we type `45 % 7`?

- A. 6
- B. 3
- C. 6.428571428571429
- D. Not given

Question 6:

What will be the output if we type the following code?

```
word = 'Python'
```

```
word[1]
```

- A. 'P'
- B. 'p'
- C. 'y'
- D. 'Y'

Question 7:

What will be the output if we type the following code?

word = 'Python'

word[-2]

- A. 'n'
- B. 'o'
- C. 'h'
- D. 'P'

Question 8:

What will be the output if we enter 80 as student marks when we run the following code?

```
1. print('Input marks of a student')
2. student_marks = int(input())
3. if(student_marks > 100 or student_marks < 0):
4.     print("Marks exceed 100.")
5. elif(student_marks >= 80):
6.     print("Excellent")
7. else:
8.     print("Not excellent")
```

- A. Nothing will be printed
- B. Not excellent
- C. Excellent
- D. Marks exceed 100.

Question 9:

Suppose we have run the following piece of code.

```
1. mybirds = ["Parrot", "Sparrow", "Crow", "Eagle"]
2. mybirds.insert(1,'Crow')
3. mybirds
```

What would be the result?

- A. ['Parrot', 'Sparrow', 'Crow', 'Eagle']
- B. ['Parrot', 'Sparrow', 'Crow', 'Crow', 'Eagle']
- C. ['Parrot', 'Crow', 'Sparrow', 'Crow', 'Eagle']
- D. ['Crow', 'Parrot', 'Sparrow', 'Crow', 'Eagle']

Question 10:

What would be the result if we have run the following piece of code?

```
1. mybirds = ["Parrot", "Sparrow", "Crow", "Eagle"]
2. mybirds.remove("Pigeon")
3. mybirds
```

- A. ['Parrot', 'Sparrow', 'Crow', 'Eagle']
- B. ['Parrot', 'Sparrow', 'Crow', 'Eagle', 'Pigeon']
- C. An error message will be displayed
- D. Not given

2

Starting with Probability

2.1 Definition of Probability

Statistical processes and measurements are usually uncertain because the outcomes cannot be precisely predicted. In the process of collecting statistical data, we generally assume that there is a true value, which lies within the measurements we make. This means there is uncertainty in collecting the data. Generally, our goal is to find the best estimate of our measured data under uncertainty.

To measure the uncertainty of statistical processes and events, we resort to the probability theory that is often used to describe uncertain quantities. We explain the concept of probability in the subsequent sections of this chapter.

In numerous situations, we have to describe an event in terms of its chances of occurrence. For example, weather forecasts tell us about the chances of rain on any particular day. In sports analysis, the probability of a team winning is given before or during the actual game.

The definition of probability is a subject of philosophical debate. Probability is a single number that varies from 0

to 1. A high value of the probability of an event indicates more likeliness that the event will occur.

In repeatable experiments whose outputs are random such as tossing a coin or drawing a card from a deck, the probability of a desirable event happening can be defined as the number of desired outcomes divided by the total number of outcomes from all the repetitions of the experiment.

For instance, when we toss an unbiased coin, either of the two outcomes—heads or tails—can occur. The chances of getting heads are equal to the chances of getting tails. We say that the probability of heads and tails is 0.5 each.

In another experiment where a coin is tossed thrice, we may get one out of eight possible outcomes: HHH, HHT, HTH, HTT, THH, THT, TTH, or TTT. Here, H and T represent heads and tails, respectively. The probability of getting a particular outcome, for example, all heads HHH, is 1 out of 8 outcomes. Numerically, we report the probability as $1/8$ or 0.125. When such an experiment is performed, the chances of occurrence of a particular outcome may not be as per the calculations.

There are two opposing factions that interpret the concept of probability differently. One of the groups is known as **frequentist** that interprets the probability as the **relative frequency** of the happening of a certain outcome, provided the experiment is repeated again and again. This is somewhat an objective view of probability because the fundamental assumption is the repetition of the experiment. The computation of the probability in the aforementioned experiments follows the frequentist approach.

The second major group, also known as **Bayesian**, interprets probability as a **degree of belief** in an event. This degree of

belief is usually based on expert knowledge of the event, which is obtained from the results of previous experiments and is represented by a **prior** probability distribution. The combination of the prior with a **likelihood** function results in a **posterior** probability distribution that incorporates all the information to get the probability of an event as a degree of belief. This understanding of probability is rather subjective, and it is different from the frequentist interpretation. We shall go into the details of the Bayesian interpretation when we discuss Bayes' theorem in the same chapter.

2.2 Some Important Definitions

In statistical studies or experiments, the quantities to be measured are named **random variables**. An observation is a specific outcome of the experiment. Numerous observations collected from the study constitute the **data**, and an assortment of all possible outcomes of the experiment is called the **population**.

Practically, we cannot observe the whole **population**. As an alternate, we take a **sample**, that is, a small portion of the population. We try to get the sample from the population in such a way that it should represent the whole population. Thus, we take a random sample in which all members of the population are equally likely to be included in the sample.

For instance, if we are going to conduct a survey on the amount of time people spend online, we shall select and observe some individuals. This subset of people might provide us a biased sample of the population, and the results obtained from such a survey would not generalize to the whole population. Thus,

it is important to collect a sample of a considerable size that is representative of the whole population.

2.3 Samples Spaces and Events

The outcomes of statistical experiments are generally random. The probability theory is employed to study the random behavior of statistical experiments. The set of all possible outcomes of an experiment forms the **sample space** specific to the experiment. We usually denote the sample space by S . An element that is an outcome of S is represented by s . The sample space is selected to get one outcome from one run of the experiment. The sample space can be finite or infinite.

For example, if we roll a dice, we get one of the numbers from 1 to 6. The sample space of this experiment is given as,

$$S = \{1, 2, 3, 4, 5, 6\}.$$

As another example, tossing a coin twice will constitute the following sample space:

$$S = \{\text{HH}, \text{HT}, \text{TH}, \text{TT}\}.$$

We can view the aforementioned experiment in a slightly different way. For example, when we are interested in the number of heads, the sample space would be given as,

$$S = \{0, 1, 2\}.$$

All of the abovementioned sample spaces are **countably finite**.

The time from a specified point to the occurrence of a particular event is known as survival time. For example, the survival time, in hours, from the birth of a honeybee has the following sample space,

$$S = \{0, 1, 2, 3, 4, \dots\}$$

Practically, there is an upper limit to this survival time, but we do not know it in advance. Therefore, we include all non-negative integers in the sample space. This is an example of a **countably infinite** sample space. Finally, the lifetime of a light bulb can be any positive real number with the sample space,

$$S = [0, \infty),$$

where [includes 0 in the set, and) excludes the infinity ∞ from the set. This is an example of an **uncountably infinite** sample space because the time a light bulb may survive can take on any positive real number.

An **event** is a subset of a sample space. An empty subset of the sample space is known as the null event. The whole sample space is another type of event. An event that contains a single element of the sample space is called an elementary event, whereas an event that is constituted of multiple outcomes of the sample space is called a compound event.

In the example of tossing a coin twice, we define an event that no heads occurs: $E = \{O\}$.

This forms an elementary event. On the other hand, defining an event that at most one heads occurs constitutes a compound event with the following sample space:

$$E = \{O, 1\}.$$

The events can be combined together the way we combine different sets. For example, in rolling dice, event 1, E_1 , can be defined as getting an odd number, and event 2, E_2 , can be defined as getting a number less than 4.

$$E_1 = \{1, 3, 5\}.$$

$$E_2 = \{1, 2, 3\}.$$

The **intersection** of events E_1 and E_2 is another event E_3 that would be an odd number less than 4.

$$E_3 = E_1 \cap E_2 = \{1, 3\}$$

Moreover, the **union** of events can also be defined. For instance, we define E_4 as an event that the outcome is either odd or less than 4.

$$E_4 = E_1 \cup E_2 = \{1, 2, 3, 5\}.$$

The laws applicable to set theory are also equally applicable to events. These laws include commutative, associative, and distributive laws.

The relationship between different events is usually depicted graphically as a logic or set diagram known as a **Venn diagram**. The events are shown as regions represented by closed curves such as circles.

The points inside a circle signify the elements of the event, whereas the points outside the circle denote the elements of the sample space not present in the event. For instance, the set of all elements that are members of both events E_1 and E_2 , i.e., the intersection of E_1 and E_2 , is shown as the overlapped area in the Venn diagram. Figure 2.1 shows the intersection of two events as the area highlighted in brown color.

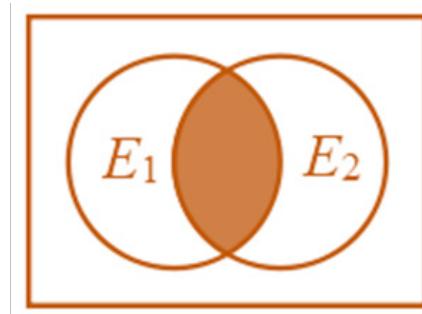


Figure 2.1: A Venn diagram showing the intersection of two events E_1, E_2 .

Figure 2.2 depicts the union of two events as the highlighted area.

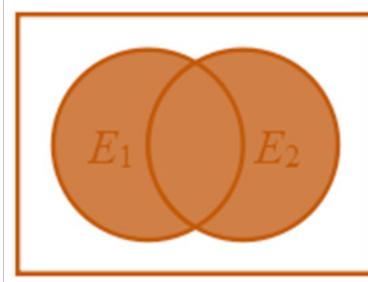


Figure 2.2: A Venn diagram showing the union of two events E_1 , E_2 .

In general, when there is an overlap between two events, we calculate the probability of the union of both events as,

$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2).$$

The term $P(E_1 \cap E_2)$ or the probability of the intersection is subtracted from $P(E_1) + P(E_2)$ because the sum of $P(E_1)$ and $P(E_2)$ includes the common area twice.

The disjoint or mutually exclusive events do not have any common elements from the sample space. Figure 2.3 shows two disjoint events as a Venn diagram.

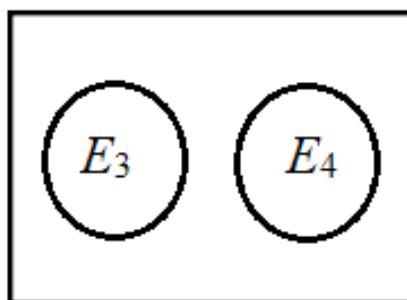


Figure 2.3: A Venn diagram showing two disjoint or mutually exclusive events that do not have any common elements of the sample space.

2.4 Axioms of Probability

Generally, we use sets to understand the events drawn from a sample space. The set theory can also be used to understand the probability theory. Based on the set theory, some propositions about probability can be deduced. These propositions are known as probability axioms, where an axiom is a proposition whose truth is self-evident. These axioms are given as:

1. $P(S) = 1.$

The probability that at least one of all the possible outcomes of the sample space S will occur is 1. Alternatively, when an experiment is performed, some events of the sample space of this experiment will always occur.

2. $P(E) \geq 0$

If E is an event (a subset of S), its probability is equal to or greater than zero.

3. $P(E_1 \cup E_2) = P(E_1) + P(E_2)$ for mutually exclusive or disjoint events E_1 and E_2 .

The symbol \cup stands for the set union. We can redefine this as: If E_1 and E_2 have nothing in common, i.e., these are mutually exclusive, the probability of either E_1 or E_2 equals the probability of occurrence of E_1 plus the probability of occurrence of E_2 .

2.5 Calculating Probability by Counting

Statistical experiments and the data used by them usually have an element of randomness. We employ probability theory as a foundation to understand this random behavior.

The frequentist statistics interprets the probability as relative frequency. In this interpretation, the probability is defined for

those events from experiments that can be repeated again and again under similar conditions.

The frequentist approach explains the probability of an event as the ratio of the number of times the event occurs to the total number of times the experiment is run, provided the experiment is run numerous times under identical conditions. The probability can also be defined as a long-run proportion, that is, the ratio of the occurrence of the event to the total large number of runs of the experiment.

In this case, we count the occurrence of the event whose probability is to be found. Mathematically, the probability p of the event E is defined as,

$$p = \lim_{n \rightarrow \infty} \frac{m}{n}$$

where m is the number of times the event E occurs in n repeated trials of the experiment. The frequentist interpretation of the probability is generally well-accepted by most statisticians. However, the problem with this interpretation is that it assumes the experiment is repeatable. In many real-world natural events that happen only once, such as the occurrence of rain in a particular geographic area the next day, this interpretation fails.

To calculate the probability by counting, let us consider an experiment. Suppose that a fair coin is thrown twice. The sample space of this experiment is,

$$S = \{HH, HT, TH, TT\}$$

Each outcome has a probability of $\frac{1}{4}$ under the assumption of equally likely outcomes. We define event A as getting both tails and event B as getting heads on the second toss as follows:

$$A = \{\text{TT}\} \text{ and}$$

$$B = \{\text{HH}, \text{TH}\}.$$

The probability of both events can be given as:

$$P(A) = \text{Number of times } A \text{ occurs} / \text{total number of outcomes}$$

$$P(A) = 1/4$$

$$P(B) = \text{Number of times } B \text{ occurs} / \text{total number of outcomes}$$

$$P(B) = 2/4$$

In another example, a statistical experiment is conducted numerous times. The data obtained from the experiment are summarized in the table given below. From this collected sample data, what is a reasonable estimate of the probability that the outcome of the next run of the experiment will be 5?

Outcome	Frequency	Relative Frequency
1	200	0.0513
2	100	0.0256
3	900	0.2307
4	500	0.1282
5	1200	0.3077
6	500	0.1282
7	200	0.0513
8	300	0.0769

The frequentist approach calculates the probability of an event by computing the relative frequency, which is obtained when we divide the frequency of occurrence of an event by the total number of runs of the experiment.

Though the data given in the abovementioned table is limited, we can still estimate the probability of occurrence of 5. The relative frequency of outcome 5 is the number 1,200 divided by the total number of runs of the experiment, i.e., 3,900. Thus, our estimate of the probability for outcome 5 is approximately $1200/3900 \approx 0.30$. In Python, we type the following code.

```
1. ## Calculating probability of events
2.
3. # Sample Space
4. num_cards = 52
5.
6. # Favorable Outcomes
7. num_aces = 4
8.
9. num_hearts = 13
10. num_diamonds = 13
11.
12. # Divide favorable outcomes by the total number of
     elements in the sample space
13. prob_ace = num_aces / num_cards
14.
15. prob_red_card = (num_hearts+ num_diamonds) / num_cards
16.
17.
18. # Print probability rounded to two decimal places
19. print('The probability of getting an ace =', round(prob_
     ace, 3))
20. print('The probability of getting a red card =',
     round(prob_red_card, 3))
21.
22. # to print the probability in percentage
23. prob_ace_percent = prob_ace * 100
```

```
24. prob_red_percent = prob_red_card * 100
25.
26. print('\nThe probability of getting an ace in percentage
      =', str(round(prob_ace_percent, 1)) + '%')
27. print('The probability of getting a red card in percentage
      =', str(round(prob_red_percent, 1)) + '%')

Output:
The probability of getting an ace = 0.077
The probability of getting a red card = 0.5

The probability of getting an ace in percentage = 7.7%
The probability of getting a red card in percentage = 50.0%
```

2.6 Combining Probabilities of More than One Events

Let us define events A and B:

- **Event A:** It is raining outside. Let there be a 40 percent chance of raining today. The probability of event A is $P(A) = 0.4$.
- **Event B:** A person needs to go outside. Let there be a 30 percent chance of the person going outside. The probability of B would be $P(B) = 0.3$.

Joint probability: Let the probability that both events happen simultaneously is 0.2 or 20 percent. It is written as $P(A \text{ and } B)$ or $P(A \cap B)$ and is known as the joint probability of A and B. The symbol \cap is for the intersection of the events.

2.7 Conditional Probability and Independent Events

Suppose we are interested to know the probability or the chances of occurrence of rain given the person has gone

outside. The probability of rain given the person has gone outside is the **conditional probability** denoted as $P(A|B)$.

The conditional probability can be understood by thinking of a reduced sample space when one of the events, i.e., B in our case, has already occurred. Since we are certain that B has occurred, we do not need to look for the whole sample space for calculating the conditional probability $P(A|B)$. Thus, we find the probability of the intersection of A and B under the condition that B has occurred. Mathematically, it can be given as,

$$\begin{aligned} P(A|B) &= P(A \cap B) / P(B) \\ &= 0.2/0.3 = 0.66 = 66.6\%. \end{aligned}$$

Some events depend upon other events. For example, let event A be “buying a lottery ticket,” and event B be “winning the lottery.” Event B is dependent upon event A because we cannot win without buying a ticket.

Conversely, two events are said to be **independent** if the occurrence of one does not affect the occurrence of the other event. For instance, rolling dice and tossing a coin are two independent events since these events can occur in any order, and the occurrence of one does not affect the occurrence of the other event.

Therefore, for any two independent events A and B , the probability of event A given the event B has occurred before is given as $P(A|B)$:

$$P(A|B) = P(A)$$

The probability of event B given the event A has occurred before is given as $P(B|A)$:

$$P(B|A) = P(B)$$

$$P(A \cap B) = P(A|B) P(B) = P(A) P(B)$$

From this last expression, it is obvious that for independent events, the joint probability of both events equals the product of individual probabilities of the events.

To find out if the events are dependent or independent, we perform certain simple tests.

- If the order of happening of two events matter, one of the events is dependent upon the other event. For instance, parking a car and getting the parking ticket are dependent events.
- If the order of events does not matter, we check if the occurrence of one event impacts the outcome of the second event. If yes, the events are dependent; otherwise, the events are independent.

Drawing a card from a deck of 52 cards, replacing it to the deck, and then drawing another card from the same deck constitutes two independent events as two draws from the deck. Since we replace the card, we do not change anything. The probability of drawing the first card is 1/52, and the probability of drawing the second card is also 1/52.

However, if we draw a card, then draw another card **without replacing** the first card, we have a 1/52 probability of drawing the first card and 1/51 probability of drawing the second card because the second draw is from 51 cards instead of 52 cards. This is an example of dependent events.

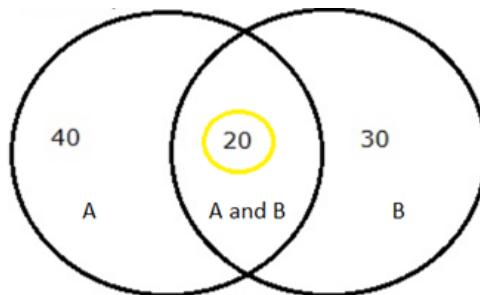


Figure 2.4: Explanation of the joint and conditional probability.

We implement these concepts in Python as follows.

```

1. ## Probability of Independent and dependent Events
2.
3.
4. # Sample Space
5. total_cards = 52
6. cards_drawn = 1
7. # remaining cards when the card drawn in the first place is
   not replaced
8. rem_cards = total_cards - cards_drawn
9.
10. # Calculate the joint probability of drawing a king after
    drawing a queen in the first draw with replacement
11. num_queens = 4
12. num_kings = 4
13. prob_king1 = num_kings / total_cards
14. prob_queen1 = num_queens / total_cards
15.
16. # probability of intersection of events
17. prob_king_and_queen1 = prob_king1 * prob_queen1
18.
19. # Determine the joint probability of drawing a king after
    drawing a queen in the first draw without replacement
20.
21. prob_king2 = num_kings / rem_cards
22. prob_queen2 = num_queens / total_cards
23.
24. # probability of intersection of events
25. prob_king_and_queen2 = prob_king2 * prob_queen2

```

```
26.
```

```
27.
```

```
28. # Print each probability
```

```
29. print('The product of probability of getting a king and  
the probability of drawing a queen = ',round(prob_king1 *  
prob_queen1,5))
```

```
30. print('The probability of getting a king after drawing a  
queen in the first draw with replacement = ',round(prob_  
king_and_queen1,5))
```

```
31. print('The probability of getting a king after drawing a  
queen in the first draw without replacement = ',round(prob_  
king_and_queen2,5))
```

Output:

```
The product of the probability of getting a king and the  
probability of drawing a queen =  0.00592
```

```
The probability of getting a king after drawing a queen in the  
first draw with replacement =  0.00592
```

```
The probability of getting a king after drawing a queen in the  
first draw without replacement =  0.00603
```

The aforementioned example shows that if the first card drawn is replaced, and another card is drawn from the deck, both events become independent of each other. The product of both probabilities equals the probability of getting a king after drawing a queen in the first draw with replacement. The last line of the output shows a different value of the probability because the first card is not replaced.

2.8 Bayes' Theorem

Besides $P(A|B)$, there is another conditional probability related to the event: the probability of occurrence of event B given A has already occurred, i.e., $P(B|A)$. **Bayes' theorem** converts one conditional probability to the other conditional probability. Since

$$P(A \cap B) = P(A|B) \cdot P(B)$$

and

$$P(A \cap B) = P(B|A) \cdot P(A).$$

Equating these two equations, we get

$$P(B|A) = [P(A|B) \cdot P(B)] / (P(A)).$$

For the events A and B defined in Section 2.7,

$$\begin{aligned} P(B|A) &= (0.66)(0.3)/0.4 \\ &= 0.495 = 49.5\%. \end{aligned}$$

Recall that we have calculated $P(A|B) = 0.66 = 66.6\%$ in Section 2.8. It is evident from this example that generally, $P(A|B)$ is not equal to $P(B|A)$.

The Bayes' theorem is usually used when we have access to the data, and we want to find out some unknown parameters from the data. Suppose event A represents the data, and event B represents some unknown parameter to be estimated from the data. We can interpret the probabilities used in the Bayes' theorem as follows.

- **P(B):** the probability of event B (sometimes referred to as an unknown parameter or a hypothesis) regardless of the data. This is known as the **prior** probability of B or the unconditional probability.
- **P(A):** the probability of the data, regardless of the event B. This is known as **evidence**.
- **P(A|B):** the probability of data A given that the hypothesis or the assumed parameter B is true. This is known as the **likelihood** of data A conditional on hypothesis B.

- **P(B|A):** the probability of event B given the data A . This is known as the **posterior** probability. Usually, we are interested to find this probability.

It is important to realize that if one of the conditional probabilities is used as a likelihood function, the other conditional probability will be the posterior. Using $P(B|A)$ as the likelihood will make $P(A|B)$ a posterior.

2.9 Calculating Probability as Degree of Belief

The probability is calculated as a degree of belief in the Bayesian interpretation of the probability. Suppose, in a rainy season, it rains most of the days of the month in some geographical area. The natives of that area believe that the chances of having a rainy day are 80 percent or 0.8. This becomes the prior probability that is based on the degree of belief of the natives. We write,

$$P(\text{rainy day} = \text{true}) = 0.8,$$

where a degree of belief of 80 percent that a randomly chosen day receives rain is the prior probability of having a rainy day in the absence of any other evidence or the data.

It is important to point out that the degree of belief denotes the probability of a particular event happening before we make an actual observation of the event. Obviously, the priors of the events can change when we observe the actual outcomes or events. In other words, the presence of evidence may cause our degree of belief in the event to change. It means that the calculation of posterior probability from prior, likelihood, and

evidence will change our belief, which may be used in later experiments of a similar type.

As a practical example, we might want to calculate the probability that a patient has heart disease, given they are obese.

- We define event A as “patient has a heart disease.” From previous experience and the data collected from different hospitals, it is believed that 15 percent of patients have heart disease, i.e., $P(A) = 0.15$.
- We define event B as “patient is obese.” From the past collected data, we believe that 10 percent of the patients are obese, i.e., $P(B) = 0.1$.
- Suppose that we know from the hospital tests data that 20 percent of the patients diagnosed with heart disease are obese, i.e., $P(B|A) = 0.2$. The probability that a patient is obese, given that they have heart disease, is 20 percent. $P(B|A)$ is referred to as a likelihood function.
- Now, we are interested to find out the probability that a patient has heart disease if they are obese, i.e., $P(A|B)$. This new probability in the presence of evidence, obesity, is called posterior probability. It is calculated using the Bayes’ theorem as follows:

$$P(A|B) = P(B|A) \times P(A) / P(B)$$

$$P(A|B) = 0.2 \times 0.15 / 0.1 = 0.3.$$

This implies that if a patient is obese, their chances of having heart disease are 0.3 or 30 percent in contrast to 15 percent chances in the absence of the evidence, as suggested by the past collected data. Thus, the presence of evidence of one event alters the posterior probability of the other event.

```

1. # calculating the probability as a degree of belief
2. # calculating the probability of heart disease when a
   patient is obese
3.
4. # defining a function to find out  $P(A|B)$  given  $P(A)$ ,  $P(B)$ ,
    $P(B|A)$ 
5. def bayes_rule(p_a, p_b, p_b_over_a):
6.     # calculate  $P(A|B)$ 
7.     p_a_over_b = (p_b_over_a * p_a) / p_b
8.     return p_a_over_b
9.
10.
11. ### testing of the funciton that employs Bayes rule to get
    posterior from prior
12. p_a = 0.15
13. p_b_over_a = 0.2
14. p_b= 0.1
15.
16. # calculating the posterior  $P(A|B)$ 
17. p_a_over_b = bayes_rule(p_a, p_b, p_b_over_a)
18. # summarize
19. print('The posterior probability  $P(A|B) = %.1f\%$ ' % (p_a_
   over_b * 100))
Output:
The posterior probability  $P(A|B) = 30.0\%$ 
```

In lines 5 to 8, we define a function, namely `bayes_rule()` that takes the given probabilities $P(A)$, $P(B)$, and $P(B|A)$ as inputs and produces $P(A|B)$ as the output. Lines 12 to 19 test our function. We define arbitrary values of $P(A)$, $P(B)$, and $P(B|A)$ and call the function `bayes_rule()` within the `print` statement. Note that we have used string formatting operator `%` as `%.1f%%` that specifies the format of the floating-point number to be printed. This formatting is applied to the floating-point number returned by `(p_a_over_b * 100)` to give the result to one decimal place as specified by `.1f`.

Requirements

The Python scripts presented in this chapter have been executed using the Jupyter notebook. Thus, to implement the Python scripts, you should have the Jupyter notebook installed. Since Jupyter notebook has built-in libraries, we do not need to install them separately.

Further Readings

For the practice of questions related to probability theory, please visit the following links:

<https://bit.ly/2I3YQDM>

For details and applications of Bayes' theorem, visit

<https://bit.ly/3mT9IDi>

Hands-on Time – Source Codes

The Jupyter notebook containing the source code given in this chapter can be found in Resources/Chapter 2.ipynb. We suggest that the reader writes all the code given in this chapter to verify the outputs mentioned in this chapter.

2.10 Exercise Questions

Question 1:

In an experiment, two coins are flipped together. What is the probability that both coins lands heads?

- A. 1/2
- B. 1/4
- C. 1/8
- D. 1

Question 2:

Suppose that we draw a card randomly from a deck of 52 cards. From the options below, choose the correct probability that the drawn card is of black color?

- A. $1/2$
- B. $1/4$
- C. $1/8$
- D. 1

Question 3:

Suppose that we draw a card randomly from a deck of 52 cards. From the options below, choose the correct probability that the drawn card is a king of spades?

- A. $1/13$
- B. $1/4$
- C. $1/52$
- D. $4/52$

Question 4:

Which one of the following is not an example of independent events?

- A. Rolling a dice, then tossing a coin
- B. Buying a new car, then buying a pair of shoes
- C. Drawing a card from a deck, then drawing another card without replacing the first card
- D. Drawing a card from a deck, then drawing another card after replacing the first card

Question 5:

If the probability of an event $P(E) = 0.4$, $P(\text{not } E)$ will be:

- A. 0.4
- B. 0.5
- C. 0.6
- D. 1

Question 6:

The probability of drawing an ace from a deck of 52 cards is:

- A. $1/52$
- B. $1/26$
- C. $4/13$
- D. $1/13$

Question 7:

A dice is rolled. Find out the probability of getting either 2 or 3?

- A. $1/6$
- B. $2/6$
- C. $1/3$
- D. $1/2$

Question 8:

If a card is chosen from a deck of 52 cards, what is the probability of getting a one or a two?

- A. $4/52$
- B. $1/26$
- C. $8/52$
- D. $1/169$

Question 9:

A footballer scores at least one goal in 6 matches out of 30 matches. Find the probability of the matches in which he did not score any goal?

- A. $1/5$
- B. $2/5$
- C. $3/5$
- D. $4/5$

Question 10:

What is the probability of getting a sum of 10 from two rolls of a dice?

- A. $1/36$
- B. $2/36$
- C. $3/36$
- D. $4/36$

3

Random Variables & Probability Distributions

3.1 Random Variables: Numerical Description of Uncertainty

A basic concept when dealing with uncertainty is that of a *random variable* (RV). Randomly picking up a student from a university class and measuring their height or weight is a random process. We are uncertain about the outcome of this process beforehand. The measured height can be considered an RV. Thus, we may define an RV as “*a real-valued function of a sample space.*” The sample space is the grouping of all possible outcomes of an experiment. For instance, the set of numbers from 1 to 6 is the sample space for rolling dice once. In this case, the random variable can take on any value from 1 to 6.

As another example, the time it takes for a bus to get from station A to station B is a random variable. If the maximum time between two consecutive buses is T , the sample space for this experiment would be the interval $[0, T] = \{t: 0 \leq t \leq T\}$. This means that the random variable, t , can assume any length

of time in the interval $[0, T]$. In this case, the sample space is continuous in contrast to the discrete sample space of the experiment involving the dice.

Generally, the definition of the word “random” by looking it up in a dictionary would be “Lacking any definite plan or order or purpose,” or “Having no specific pattern or objective.” Statistically, this is not the correct definition. The word random in statistics implies a process or variable whose output is not determined in advance or its value is not deterministic. The output of a random process can be given by a probability distribution.

Additionally, it is possible for a random process to give more chances of occurrence to certain outcomes over other outcomes. In this case, the chances of the happening of different outcomes would differ from one another. As an instance, if we toss an unfair coin biased in favor of heads, again and again, we will have more frequency of occurrence of heads than tails. Even in this case, we consider tossing the biased coin as a random process.

Random variables can be either **qualitative** or **quantitative**.

- The qualitative variables are not measured or counted. These have non-numeric outcomes without any order. For instance, the names of animals, types of cars, and gender are all qualitative variables.
- The quantitative variables are measurable or countable. The outcome of these variables is numeric. For example, height, weight, amount of time, age, and the number of goals in a football game are all quantitative variables. There are two main categories of quantitative variables: **discrete** and **continuous** random variables.

- o Discrete random variables have a countable number of possible outcomes. For instance, the number of wins in a tournament and the number of defective items in a lot consisting of 10 items are discrete random variables.
- o Continuous random variables can assume any valid value over a continuous scale. For example, weight, height, age, and time are continuous random variables.

3.2 Generation of Random Numbers and Random Variables

In this section, we work with random numbers using Python. The generation of random numbers forms the basis for producing random variables. To generate a random integer from 0 to 50 and a random floating-point number from 0 to 1, type the following code:

```
1. from numpy import random
2.
3. # Generate a random integer from 0 to 100
4. x = random.randint(50)
5. print(x, '\n')
6.
7. #Generate a random floating point number from 0 to 1:
8. y = random.rand()
9.
10. print('%0.2f' %y)
Output:
39
0.55
```

The module random is imported from the NumPy library in line 1 of the program. The function randint (50) is used to generate a random integer in the range 0 to 50. Furthermore, in line

no. 8 of the code, the function rand () is used to generate a random floating-point number in the range 0 to 1.

We can also generate arrays of random numbers. The following program depicts this concept:

```
1. # Generating a 1-D array containing 10 random integers in
   the range 0 to 50
2. from numpy import random
3.
4. x1d=random.randint(50, size=(10))
5.
6. print(x1d)
7.
8. # Generating a 2-D array containing 4x3 random integers in
   the range 0 to 50
9. x2d = random.randint(50, size=(4, 3))
10.
11. print('\n\n',x2d)
```

Output:

```
[ 1 43 27 30 37 14 19 20 20 31]
```

```
[[43 20 36]
 [41 48 24]
 [28 48  3]
 [35 25  2]]
```

In line 4 of the code, the function randint (50, size=(10)) generates a 10-element array of random integers in the range 0 to 50. In line no. 9, randint (50, size=(4, 3)) generates a matrix of random numbers in the range 0 to 50. The size of the matrix is specified by the option size = (4, 3).

It is also possible to generate a multi-dimensional array of random floating-point numbers by specifying the size of the array in random.rand(x size, y size). For example x size = 5 and

y size = 6 will generate a 2-dimensional array of floating-point numbers having a total of $5 \times 6 = 30$ elements.

Having discussed the generation of random numbers using NumPy module random, this is the time to generate random variables using Python. There are different types of random variables, as discussed in the subsequent sections of this chapter. Here, we generate one common type of continuous random variable: uniform random variable.

A uniform random variable is defined for continuous data. It can assume any value in a certain range $[a,b]$ with equal probability. The Python function `random.uniform(a, b)` is used to generate a random variable.

```
1. ##### Generation of uniform random variable
2. from numpy import random
3. b = 20
4. a = 10
5. uni_rv = np.zeros(10)
6. for i in np.arange(10):
7.     uni_rv[i] = random.uniform(a,b)
8.
9. uni_rv
Output:
array([16.56806644, 19.4409342 , 19.16584349, 12.66223275,
       16.05819612, 16.92521575, 11.31674717, 10.34496818,
       17.61196685, 14.86797476])
```

The program given above uses `np.zeros(10)` to initialize an array `uni_rv` that is used to hold 10 values of a uniform random variable. The program uses a for loop to save values of a uniform random variable at different indices of the array `uni_rv`.

3.3 Probability Mass Function (PMF)

As discussed earlier, a random variable can be discrete or continuous. A discrete random variable assumes each of its values with some probability. For instance, if we toss a coin twice, we can have one out of four outcomes: $S = \{HH, HT, TH, TT\}$.

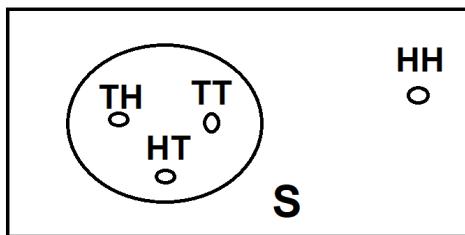


Figure 3.1: The Venn diagram showing the sample space of the experiment in which a coin is tossed twice.

Thus, the sample space contains 4 elements. The random variable X describing the number of tails assumes the value 0 with the probability $\frac{1}{4}$ because there is one outcome, HH , in which no tails occur. The variable X assumes the value 1 with the probability $\frac{1}{2}$ because there are two outcomes that have exactly one tails: HT and TH . Finally, the outcome TT has two tails that results in a probability of $\frac{1}{4}$ for $P(X=2)$. This can be summarized in the following table:

X	0	1	2
$P(X=x)$	$1/4$	$2/4$	$1/4$

Note that the value of random variable X can assume any possible value of the number of tails from the elements of sample space. Hence, the sum of probabilities of all the values of the random variable will always be 1. If these values do not

sum up to 1, we have to look for a possible mistake in the calculation of probabilities.

Since the random variable given above can assume **discrete** values only, the set of pairs of all values $(X, P(X=x))$ is called a **probability mass function** or **probability distribution** of a discrete random variable. Thus, describing the probability of each possible outcome of a random variable is known as a probability distribution. In the aforementioned example of tossing of coins, $(0, 1/4)$, $(1, 2/4)$, and $(2, 1/4)$ constitute a probability mass function. The probability mass function (PMF) of this experiment is given below in Figure 3.2.

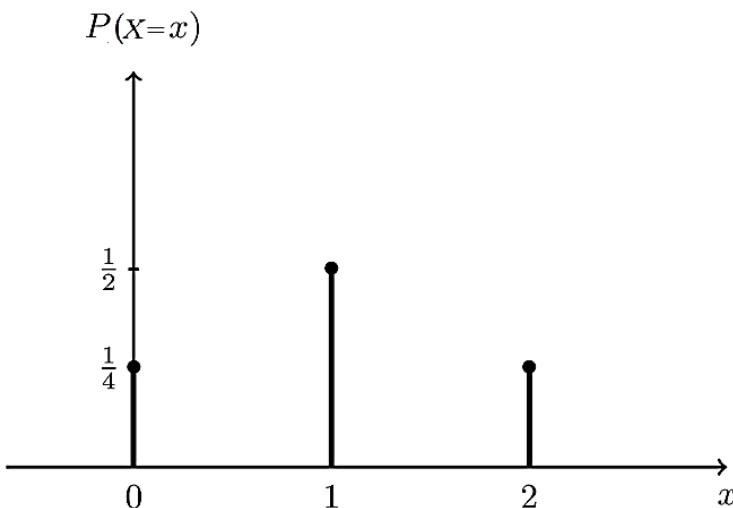


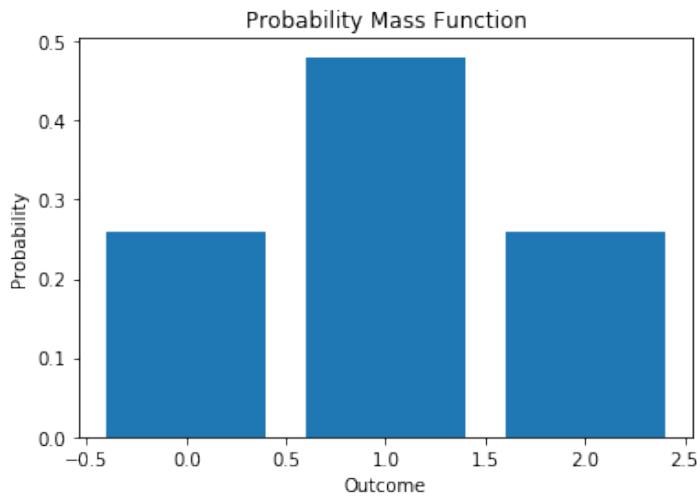
Figure 3.2: The probability mass function (PMF) of the random variable representing the number of tails when a coin is tossed twice.

The following Python script shows how we can run a statistical experiment, compute the probabilities and the probability mass function of a discrete random variable and plot the results.

```
1. ### Computing and Plotting probability mass function of a
discrete random variable
2.
3. import numpy as np
4. import matplotlib.pyplot as plt
5.
6. # inline magic function makes the plots to be stored in
the notebook document
7. %matplotlib inline
8.
9.
10. # We repeat the experiment multiple times to get a better
result of probabilities
11. num_rep_exp = 100
12.
13. # variable num_tails_rep saves the number of tails got
from each run of the experiment
14. num_tails_rep = np.zeros(num_rep_exp)
15.
16. for i in range(num_rep_exp):
17.     #
18.     outcome1 = np.random.choice(['H', 'T'])
19.     outcome2 = np.random.choice(['H', 'T'])
20.
21.     ### counting the number of tails from outcome1 and
outcome2
22.     if outcome1 == 'H' and outcome2 == 'H':
23.         num_tails = 0
24.     elif outcome1 == 'H' and outcome2 == 'T':
25.         num_tails = 1
26.     elif outcome1 == 'T' and outcome2 == 'H':
27.         num_tails = 1
28.     else:
29.         num_tails = 2
30.
31.     # Saving the number of tails from each experiment at
different indices of num_tails_rep.
32.     num_tails_rep[i] = num_tails
33.
```

```
34.  
35. outcome_value, outcome_count = np.unique(num_tails_rep,  
     return_counts=True)  
36. prob_count = outcome_count / len(num_tails_rep)  
37.  
38. # Now that we have tossed the coin twice for 1000 times,  
    we plot the results  
39. plt.bar(outcome_value, prob_count)  
40. plt.ylabel("Probability")  
41. plt.xlabel("Outcome")  
42. plt.title("Probability Mass Function")  
43. plt.show()
```

Output:



The program integrates multiple concepts together. After importing libraries in lines 3 and 4, we use the inline function of Matplotlib in line 7 that allows the plots generated by the program to be stored along with the notebook.

We repeat our experiment multiple times to plot the probability mass function after repeated trials of the experiment. Line number 14 initializes the value of the variable num_tails_rep that is used to save the result, number of tails, in each repetition

of the experiment. We use a for loop to run the experiment repeatedly. The function choice () from the random module of the NumPy library is utilized in lines 18 and 19 to choose from the specified outcomes ([‘H’, ‘T’]) randomly. The variables outcome1 and outcome2 save the values either H or T after each iteration of the for loop.

Lines 22 to 29 are then used to count the number of tails resulting from the output of lines 18 and 19. The output num_tails is stored in num_tails_rep[i] until the for loop finishes its execution. Note that the for loop starts at line 16 and ends at line 32.

The variable num_tails_rep holds 100 outcomes from 100 runs of the experiment. We use the function unique () that gives us the unique values, i.e., 0, 1, and 2 in variable outcome_value along with the number of occurrences of each value in variable outcome_count. We calculate the probability of each outcome in line 36. Finally, lines 39 to 43 use the bar function of the Matplotlib library to plot the probability mass function of our experiment.

Note that if the experiment is run, say 10 times, we may not get the shape of the PMF as we have got in the output of the code. In this case, the reason is that we are not following the basic assumption behind the frequentist interpretation of the probability, which demands us to run the experiment a large number of times. As an instance, if the experiment is run only two times, we may get 1 tails in the first run and 0 tails in the second run. Thus, the computed probabilities would be $P(X=0)=1/2$, $P(X=1)=1/2$, and $P(X=2)=0$, which do not correspond to the true PMF of the experiment.

3.4 Probability Density Function (PDF)

Similar to a continuous function, a continuous random variable has an infinite number of points. Getting the chances of occurrence of exactly one value would result in a 0 probability. Thus, unlike a PMF, we are unable to describe the distribution of a continuous random variable in tabular form. The probability distribution of a continuous random variable is a continuous function, which is also known as a probability density function (PDF).

The variable that describes the height of university students is a typical example of a continuous random variable. The height is usually measured in meters or centimeters. So, there would be an infinite number of real values between any two heights, say 170 and 171 centimeters. Even between 170.28 and 170.29 centimeters, there is an infinite number of values of heights. Thus, the probability of selecting a student at random who is exactly 170.284 centimeters tall would be extremely low because there would be many points near to the height of interest, i.e., 170.284 centimeters.

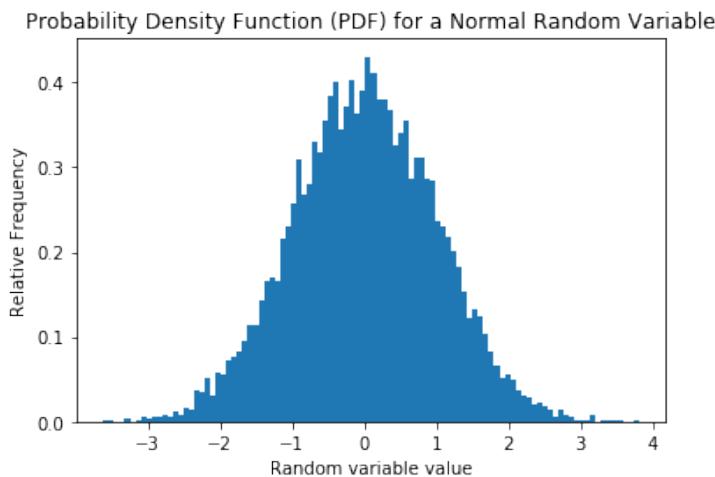
Since there are infinite numbers of heights in our sample space, we assign a probability of 0 to the event of getting a height measurement of 170.284 centimeters. In these types of cases, instead of selecting a single value, we are generally interested in a range of values. For example, we may be interested in selecting a person whose height is between 165 and 175 centimeters.

Thus, for continuous random variables, we deal with the intervals instead of a single point. To generate a continuous random variable, we can either use the NumPy library or

SciPyStats. Here, we use the latter to generate the PDF for a normally distributed continuous random variable.

```
1. from scipy.stats import norm
2. # generate random numbers from a normal probability
   density function (PDF) with zero mean and a standard
   deviation of 1: N(0,1)
3. norm_pdf = norm.rvs(size=10000,loc=0,scale=1)
4. plt.hist(norm_pdf, bins=100, density=1)
5. plt.xlabel('Random variable value')
6. plt.ylabel('Relative Frequency')
7. plt.title('Probability Density Function (PDF) for a Normal
   Random Variable')
8. plt.show()
```

Output:



After we import the necessary package `norm`, we use `norm.rvs()` in line 3 of the code to generate a Normally distributed continuous random variable. The options `loc=0` and `scale=1` specify the centre and the spread of the density function, respectively. The line 4 of the code uses `hist()` function from `Matplotlib.pyplot` module to generate **histogram** of the generated random variable.

A histogram is a plot of the data using bars, where the height of the bar represents the frequency or relative frequency of a group of data points. In a histogram, taller bars indicate that a large amount of the data falls in that range. A histogram is used to display the shape, center, and spread of continuous random variables. The option **bins** specify the number of bars or groups of the data. In this example, we have used 100 bins or 100 bars to group the overall range of the values that our normal random variable can assume.

The following code uses NumPy to generate the PDF of a continuous random variable. Try to observe the change in the output of the following code by changing the value of `number_of_data` in line 4 of the code. Moreover, changing the value of `bins` in line 8 of the code will also change the shape of the output. The reader is encouraged to play around with these values in the following code to observe changes in the output.

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4. number_of_data = 10000
5. random_data_set = np.random.randn(number_of_data)
6.
7. ## plotting the shape of the generated data as a histogram
8. plt.hist(random_data_set, bins=100)
9. plt.show()
```

Further Readings

The Python module SciPy.Stats offers a variety of continuous and discrete random variables along with useful functions to work with these distributions. For details of SciPy.Stats functions, visit

<https://bit.ly/3jYIMQs>

3.5 Expectation of a Random Variable

A **statistic** is a numerical value that summarizes our dataset or a part of the dataset. The expected value, average, or mean value of some data points, or a random variable is one of the most used statistic.

The expected or the mean value, mathematical expectation, or simply the expectation of a random variable is the long-run average value of repetitions of the experiment, which this random variable belongs to. The expectation is also known as the first moment.

For instance, the sample space for tossing one fair coin twice is

$$S = \{HH, HT, TH, TT\}.$$

All four possible outcomes are equally likely. For a random variable X representing the number of tails in both tosses, it follows that:

$$P(X = 0) = P(HH) = \frac{1}{4},$$

$$P(X = 1) = P(TH) + P(HT) = \frac{1}{2}, \text{ and}$$

$$P(X = 2) = P(TT) = \frac{1}{4}.$$

The outcome HH represents that heads occurred in both tosses, whereas HT means heads occurred in the first toss, and tails occurred in the second toss. The probabilities are the relative frequencies in the long run. Thus, to find the expected value denoted as μ , we take the average of all possible outcomes as follows:

$$\mu = E(X) = \sum x.P(X=x)$$

$$\mu = E(X) = (0)(1/4) + (1)(1/2) + (2)(1/4) = 1.$$

The expected value of rolling a six-sided dice can be computed as follows:

$$\mu = E(X) = (1)(1/6) + (2)(1/6) + (3)(1/6) + (4)(1/6) + (5)(1/6) + (6)(1/6) = 3.5.$$

To calculate the expected value of a random variable, we type the following code:

```
1. import numpy as np
2. sample_space = np.arange(6)+1
3. print("\nArray of numbers representing a sample space",
      sample_space)
4. print("The mean of the random variable is",
      np.mean(sample_space))
5. print("The average of the random variable is",
      np.average(sample_space))
```

Output:

```
Array of numbers representing a sample space [1 2 3 4 5 6]
The mean of the random variable is 3.5
The average of the random variable is 3.5
```

3.6 Probability Distributions

A probability distribution gives the probabilities of the occurrence of different values present in the dataset. Since a random variable can be discrete or continuous, we have corresponding discrete and continuous distributions. As mentioned earlier, the probability distribution of the discrete random variable is called PMF, whereas the continuous random variable's distribution is called PDF.

There are numerous continuous and discrete-valued distributions. In this section, we give the details of some of the distributions commonly encountered and used by statisticians and data scientists.

3.6.1 Bernoulli and Binomial Distribution

Bernoulli distribution is a discrete probability distribution that can take only two possible values, outputs, or outcomes: 1 for success with a probability p and 0 for failure with a probability $q = (1-p)$. This distribution assumes only one trial of the experiment that generates 0 or 1. Thus, the Bernoulli random variable assumes either value 1 (success) or 0 (failure).

The probability of success = p and

The probability of failure = q or $1-p$.

A Bernoulli RV can be completely specified by its parameter p . Mathematically, the probability function for a Bernoulli RV is given as:

$$f(x) = \begin{cases} p^x(1-p)^{(1-x)} & \text{if } x = 0, 1 \\ 0 & \text{otherwise} \end{cases}$$

A single experiment with two possible outcomes is called a Bernoulli trial, whereas a sequence of outcomes is called a Bernoulli process.

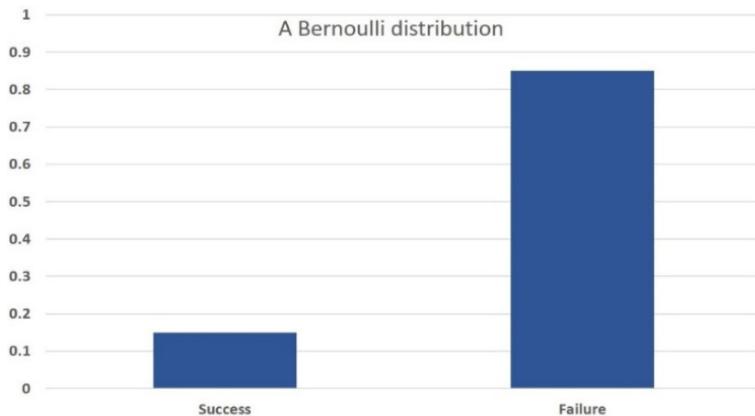


Figure 3.3: A Bernoulli distribution.

In Figure 3.3, the probability of success $p = 0.15$, and the probability of failure $q = 0.85$. The expected value is the mean of all the data values in the distribution. The expected value of a Bernoulli RV equals p .

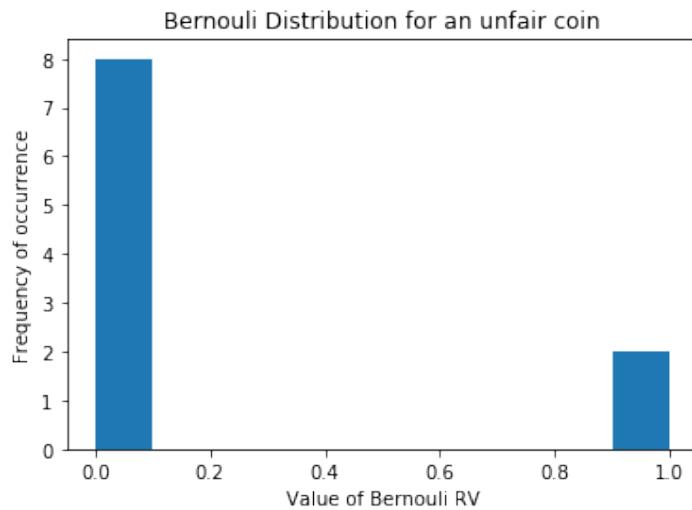
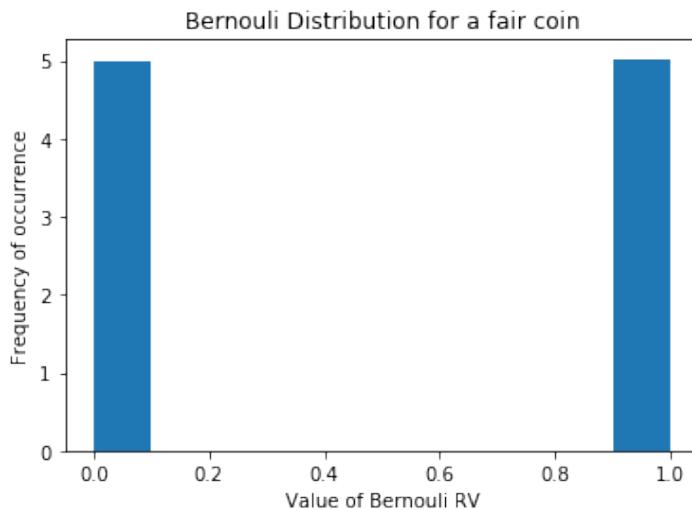
$$E(X) = xp(X=x)$$

$$=0(1-p) + 1(p) = p$$

To implement and visualize the results for Bernoulli distributions for a fair and an unfair coin, write down the following Python program:

```

1. from scipy.stats import bernoulli
2. from matplotlib import pyplot as plt
3.
4.
5. num_tosses = 1000
6. # p=0.5 is for fair coin, any other value of p results in
  unfair coin
7. fair_coin = bernoulli.rvs(p=0.5,size=num_tosses)
8. plt.hist(fair_coin)
9.
10. plt.title('Bernouli Distribution for a fair coin')
11. plt.xlabel('Value of Bernouli RV')
12. plt.ylabel('Frequency of occurrence')
13. plt.show()
14.
15. # plotting distribution for an unfair coin
16. unfair_coin = bernoulli.rvs(p=0.2,size=num_tosses)
17. plt.hist(unfair_coin)
18.
19. plt.title('Bernoulli Distribution for an unfair coin')
20. plt.xlabel('Value of Bernoulli RV')
21. plt.ylabel('Frequency of occurrence')
22. plt.show()
```

Output:

We run our experiment 1,000 times, as mentioned in line 5 of the program. A probability value $p = 0.5$ in line 7 of the code specifies a fair coin with equal probability of occurrence of tails and heads. However, $p = 0.2$ in line 16 specifies an unfair coin biased toward one of the outcomes. The plots in the output of the program verify our results.

The binomial distribution can be considered as an extension of a single experiment to multiple experiments. For a single trial, i.e., $n = 1$, the binomial distribution is a Bernoulli distribution. The binomial distribution is the basis for the popular binomial test of statistical significance.

This distribution aims to find the probability of success of an event that can assume one of two possible outcomes in a series of experiments. For example, we always get a heads or a tails when we toss a coin. For instance, to find the probability of exactly 10 heads in an experiment where a coin is repeatedly tossed 20 times, we use the binomial distribution. Here, we assume that the occurrence of heads corresponds to the successful event. Mathematically, the binomial distribution is given as:

$$f(k, n, p) = P(X = k) = \binom{n}{k} p^k (1 - p)^{(n-k)}$$

where n represents the number of runs of the experiment, and k is the parameter to represent the number of successes. The expression

$$p^k (1 - p)^{(n-k)}$$

is similar to that of Bernoulli's. To find out the total number of successes k in n runs, we find the number of successful combinations using:

$$\binom{n}{k} = \frac{n!}{k!(n - k)!}$$

where $n!$ means factorial of the number n .

$$n! = n (n-1) (n-2) \dots 3.2.1.$$

For instance, the factorial of number 5 is $5.4.3.2.1 = 120$.

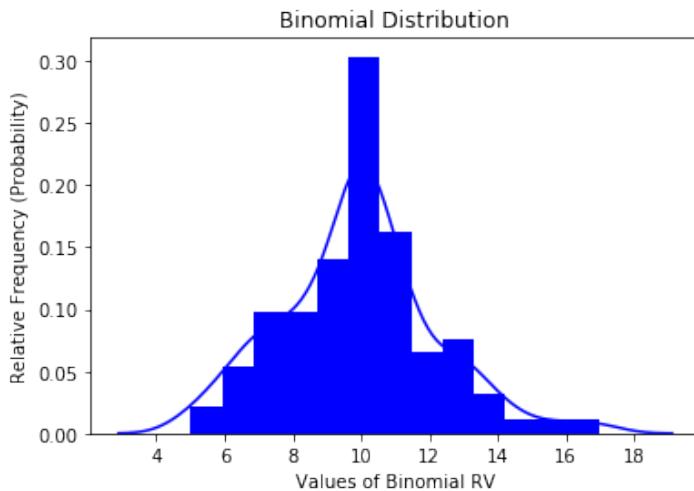
In the following Python example, we utilize SciPy.Stats package to create a binomial distribution. Though hist () function from Matplotlib library can be used to visualize probability distributions, here we import the **Seaborn** library that is based upon Matplotlib library for advanced in-built visualization functions to generate probability distribution graphs.

```
1. ## Binomial Random variable
2. import numpy as np
3. import seaborn as sns
4. from scipy.stats import binom
5.
6. ## n corresponds to the number of trials in each
   experiment, size refers to total runs of the experiment, p
   is the probability of success.
7. binom_rv = binom.rvs(n=20,p=0.5,loc=0,size=100)
8.
9. print('Number of successes in each trial having 20 coin
   tosses =', binom_rv)
10.
11.## distplot from seaborn library is used to visualize
   probability distributions
12.ax = sns.distplot(binom_rv, color='blue', hist_
   kws={"linewidth": 10,'alpha':1})
13.# hist_kws specifies keywords to be used with histogram,
   linewidth specifies the width of bars and alpha is used to
   adjust the color strength
14.ax.set(xlabel='Values of Binomial RV', ylabel='Relative
   Frequency (Probability)', title ='Binomial Distribution')
```

Output:

```
Number of successes in each trial having 20 coin tosses = [10
  13  9  9 16  7  6  9 11 10  9 13 11 10 10  8 10 13 11  7
  12 10  9 129 12 10 13  9  8 13  6 10  8  8 11 10 11 11  6
  13 11 10  7  8  7  8 14 10 10 10  7  8 11 10 10 17  8  6
  10 10 14 11 10 10 10 13 11 11 12  5 10
  8 11  9 10  9  7  9 10 11  7 10  5 15 10 11  9  7 14 12  9
  12 10  7 1110  6 10  9]
```

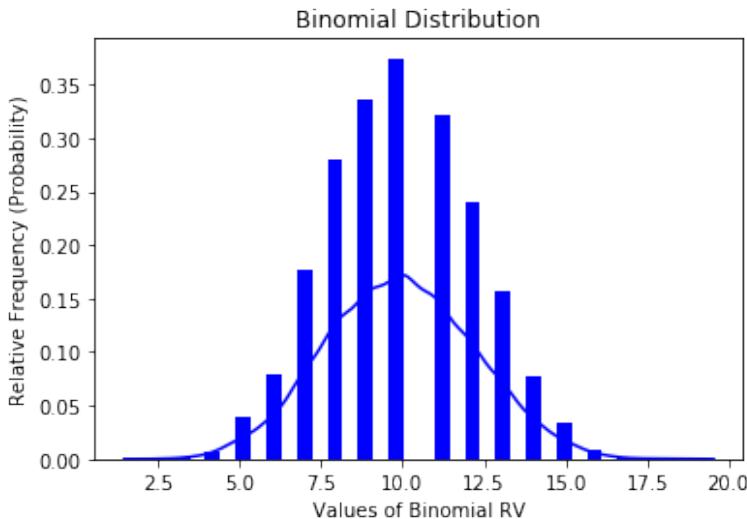
```
[Text(0, 0.5, 'Relative Frequency (Probability)'),
Text(0.5, 0, 'Values of Binomial RV'),
Text(0.5, 1.0, 'Binomial Distribution')]
```



In this code, we specify $n = 20$, that corresponds to the number of trials in each experiment, the option size = 100 in line 7 refers to the total runs of the experiment, and the value of p , the probability of success, is set to 0.5. From the output, we can observe that the probability of getting 10 heads from 20 tosses of a coin is about 0.30. Note that this value may change if we run the experiment again because out of 20 trials, we randomly get the outcomes. Thus, the event of getting exactly 10 heads in 20 trials will vary if we run the same program again. Therefore, it is important to run the experiment a large number of times to get a reliable estimate of the probability distribution of the binomial random variable.

In line 12, we use `sns.distplot` to plot the distribution of the random variable. Here, the option `hist_kws` specifies the keywords to be used with histogram, `line width` specifies the width of bars, and `alpha` is used to adjust the color strength.

If we run the same code for a size of 2,000 times, we get the following output:



We observe that the binomial distribution for a sufficiently large number of runs of the experiment approximates to a bell-shaped curve. The continuous Normal or Gaussian random variable, discussed later in this chapter, also has a bell-shaped curve.

3.6.2 Uniform Distribution

A uniform distribution is for the continuous-valued data. It has a single value, $1/(b-a)$, which occurs in a certain range $[a,b]$, whereas everything is zero outside that range. We can think of it as an indication of a categorical variable with two categories: 0 or the value. The categorical variable may have multiple values in a continuous range between some numbers a and b .



Figure 3.4: A Uniform distribution.

Mathematically, the uniform distribution is given as:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

To implement the uniform distribution and visualize the results, we use the following Python script.

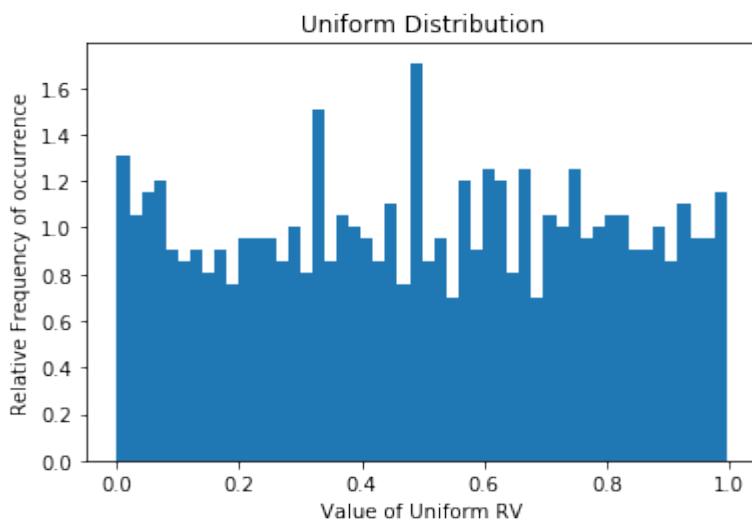
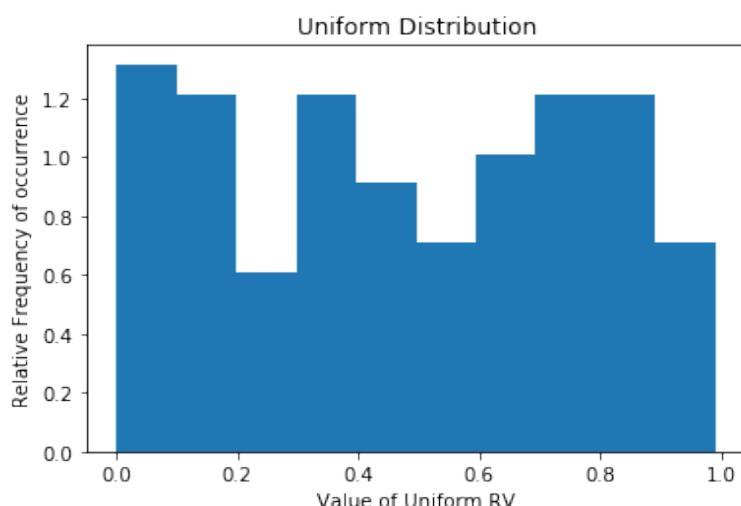
```

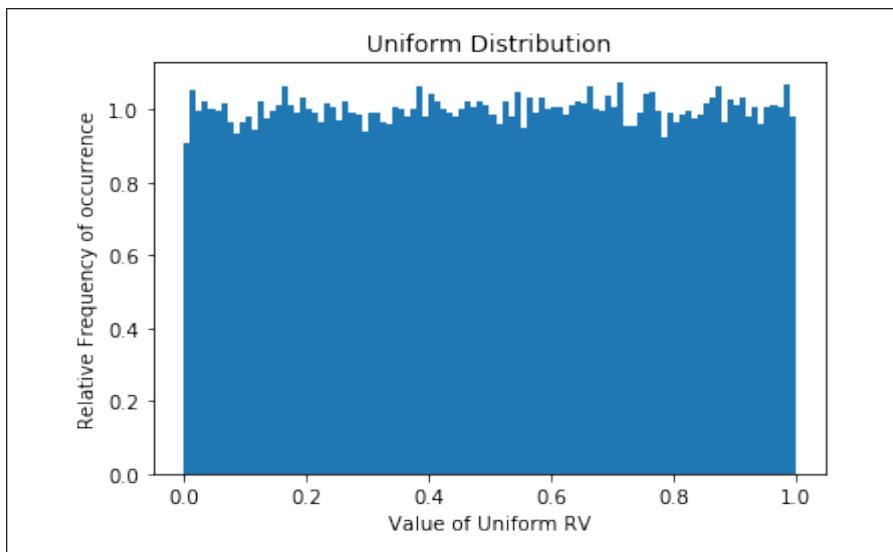
1. from scipy.stats import uniform
2. from matplotlib import pyplot as plt
3.
4. ### generating 100 samples of a uniform RV
5. uniform_rv1 = uniform.rvs(size = 100)
6. plt.hist(uniform_rv1, bins = 10, density = 1)
7. plt.title('Uniform Distribution')
8. plt.xlabel('Value of Uniform RV')
9. plt.ylabel('Relative Frequency of occurrence')
10. plt.show()
11.
12. ### generating 1000 samples of a uniform RV
13. uniform_rv2 = uniform.rvs(size = 1000)
14. plt.hist(uniform_rv2, bins = 50, density=1)
15. plt.title('Uniform Distribution')
16. plt.xlabel('Value of Uniform RV')
17. plt.ylabel('Relative Frequency of occurrence')
18. plt.show()
19.
20.

```

```
21. ### generating 100000 samples of a uniform RV
22. uniform_rv3 = uniform.rvs(size = 100000)
23. plt.hist(uniform_rv3, bins = 100, density=1)
24. plt.title('Uniform Distribution')
25. plt.xlabel('Value of Uniform RV')
26. plt.ylabel('Relative Frequency of occurrence')
27. plt.show()
```

Output:





In lines 6, 14, and 23, the option $\text{density} = 1$ normalizes the frequency of occurrence of each outcome to give us the relative frequency of the occurrence instead of the frequency. The option `bins` specify the number of groups of the data.

The first output image shows a rough approximation of the ideal uniform distribution due to an insufficient number of samples of the random variable, i.e., 100. Some of the bins/groups show a value of relative frequency more than 1. Others show a relative frequency of less than 1. However, the area under the curve of any probability distribution is always equal to 1.

It can be observed that when the size of RV increases, we get a better approximation of ideal uniform distribution as given in the last output, where we use 100,000 samples of the random variable.

3.6.3 Normal (Gaussian) Distribution

A Normal or Gaussian Distribution is defined by its mean and standard deviation. The data values are spread around the mean value, and the standard deviation controls the spread. A Gaussian distribution has most data values around the mean or center value. A smaller value of the standard deviation indicates that the data is highly concentrated and vice versa.

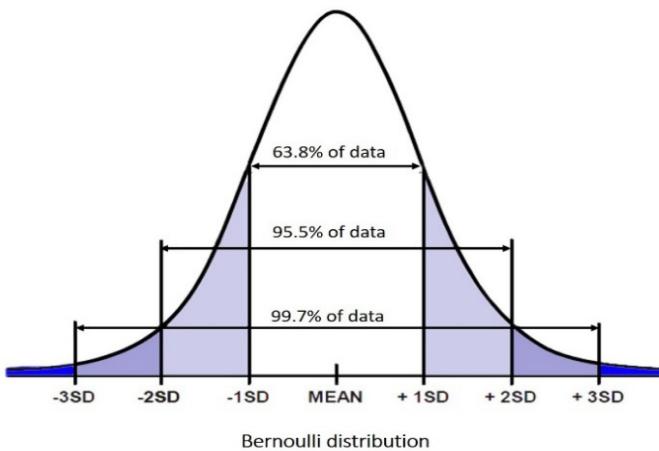


Figure 3.5: A Gaussian (Normal) distribution. The value of Gaussian RV is on the x-axis, whereas the y-axis represents the probability density.

In Figure 3.5, a Normal or Gaussian distribution is shown that is centered at MEAN. Note that 68.3 percent of the samples of Normally distributed data lies within one standard deviation, -1SD to $+1\text{SD}$, on either side of the MEAN. Moreover, 95.5 percent of the samples of Normally distributed data lie within two standard deviations, -2SD to $+2\text{SD}$, on either side of the MEAN. Mathematically, the Gaussian distribution is given as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

where e , π , μ , and σ are exponential Euler's numbers ($e = 2.718\dots$), $\pi = 3.14159\dots$, the mean, and the standard deviation, respectively. If we plot this function, we get a bell-curve, as shown in Figure 3.5. If a Normal distribution is centered around the value 0, and it has a standard deviation of 1, it is called a **standard Normal distribution**.

To implement Gaussian random variable and its distribution, type the following Python code:

```
1. from scipy.stats import norm
2. import matplotlib.pyplot as plt
3. import numpy as np
4.
5. # generating a Gaussian random variable having 50 samples
6. norm_rv1 = norm.rvs(size=50)
7. print(norm_rv1)
8. print('The mean of Normal RV1 is = %0.3f' % np.mean(norm_rv1))
9. print('The standard deviation of Normal RV2 is = %0.3f'
       % np.std(norm_rv1))
10.
11. # plotting the probability distribution of the generated
    random variable
12. plt.hist(norm_rv1)
13. plt.title('Normal (Gaussian) Distribution')
14. plt.xlabel('Value of Gaussian RV')
15. plt.ylabel('Frequency of occurrence')
16. plt.show()
17.
18. # generating a Gaussian random variable having 5000 samples
19. norm_rv2 = norm.rvs(loc= 10, scale = 5, size=5000)
20. #print(norm_rv2)
21. print('The mean of Normal RV2 is = %0.3f' %np.mean(norm_rv2))
22. print('The standard deviation of Normal RV2 is = %0.3f'
       %np.std(norm_rv2))
23.
```

```
24. # plotting the probability distribution of the generated  
random variable
```

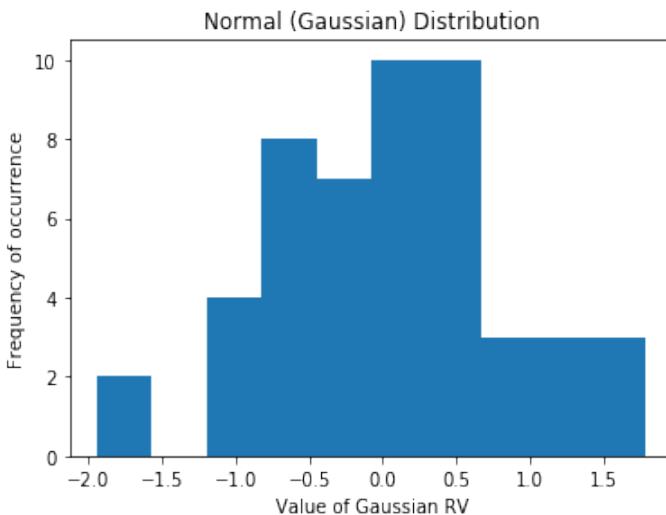
```
25. plt.hist(norm_rv2, bins=100)  
26. plt.title('Normal (Gaussian) Distribution')  
27. plt.xlabel('Value of Gaussian RV')  
28. plt.ylabel('Frequency of occurrence')  
29. plt.show()
```

Output:

```
[ 0.37376069 -0.93269653  1.79103153 -0.17525239  0.7234985   0.4447245  
 0.58314568  0.06923583  1.76857662  0.02561472 -0.4693774  -0.87189776  
 0.46097479  0.46539548  0.53569866 -1.94335259  0.18735439  0.08020707  
 0.61002039 -0.03137532  0.51216684 -0.34661192 -0.67804504  1.17631318  
-0.29321776 -0.78823176 -1.65098939  1.38928274  0.24679901  0.06046885  
-0.18963214 -0.21655229  0.44973762 -0.5473701   0.17112014 -0.53245687  
-0.92840228 -0.53733909 -0.55721695  1.30647944 -0.46756945 -0.2043573  
1.55485863 -0.3008035   0.22704038  0.58550703  0.18858938  0.83405865  
0.73521948 -0.93849856]
```

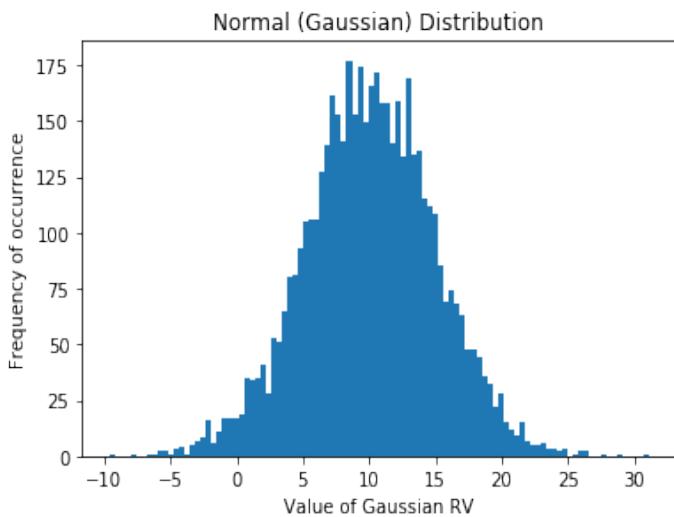
The mean of Normal RV1 is = 0.082

The standard deviation of Normal RV2 is = 1.041



The mean of Normal RV2 is = 9.999

The standard deviation of Normal RV2 is = 4.950



We generate a Gaussian RV of 50 samples in line 6 of the code. These values are printed as an array of numbers using the print command given in line 7. Lines 8 and 9 are used to calculate the mean and the standard deviation of the generated random variable. The code to generate the distribution is given in lines 12 to 16. Since the number of points is limited, we get a rough approximation of the ideal normal distribution.

We generate another Normal RV with 5,000 samples in line 19 of the code. Its center and spread are specified using the options loc and scale, respectively. The mean and the standard deviation of this RV are 9.99 and 4.95, respectively, which are close to the ideal values 10 and 5. The second plot in the output shows a reasonably good approximation of the ideal normal distribution.

3.6.4 Poisson Distribution

A Poisson distribution is a discrete distribution. Its shape is similar to the continuous Normal distribution but with some skewness. A Poisson distribution has a relatively uniform spread in all directions, just like the Normal distribution; however, the spread becomes non-uniform for increasing values of skewness.

A Poisson distribution with a low mean is highly skewed. The tail of the data extends to the right. However, if the mean is larger, the distribution spreads out, tends to be more symmetric, and becomes more like the Gaussian distribution. Mathematically, the probability mass function (PMF) of a Poisson distribution is given as:

$$f(k, \mu) = P(X = k) = \frac{\mu^k e^{-\mu}}{k!}$$

where the parameters μ and k represent the expected (average) rate of occurrence and the number of occurrences of an event, respectively, whereas e is the Euler's number ($e = 2.718\dots$), and $k!$ is the factorial of k .

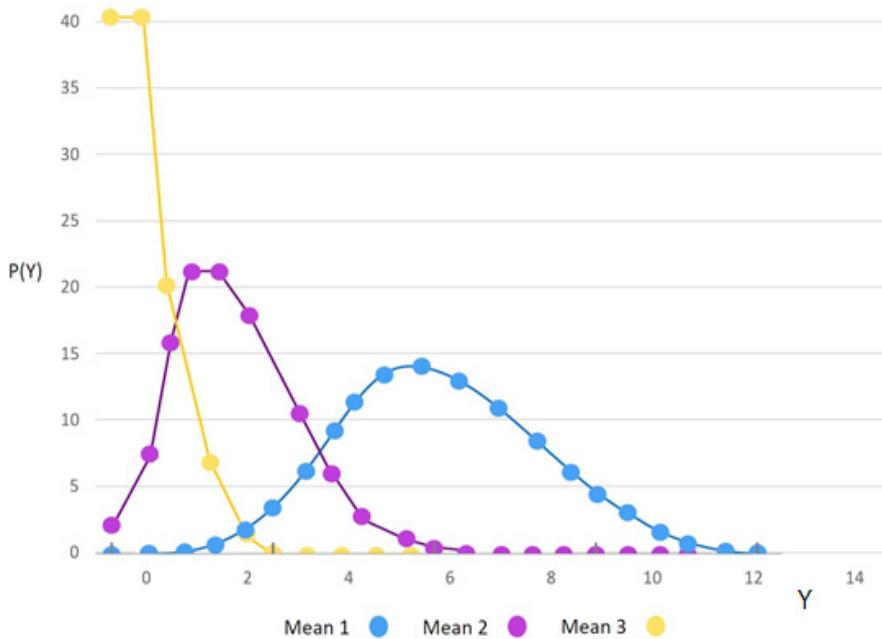


Figure 3.6: Poisson distributions with different values of the mean.

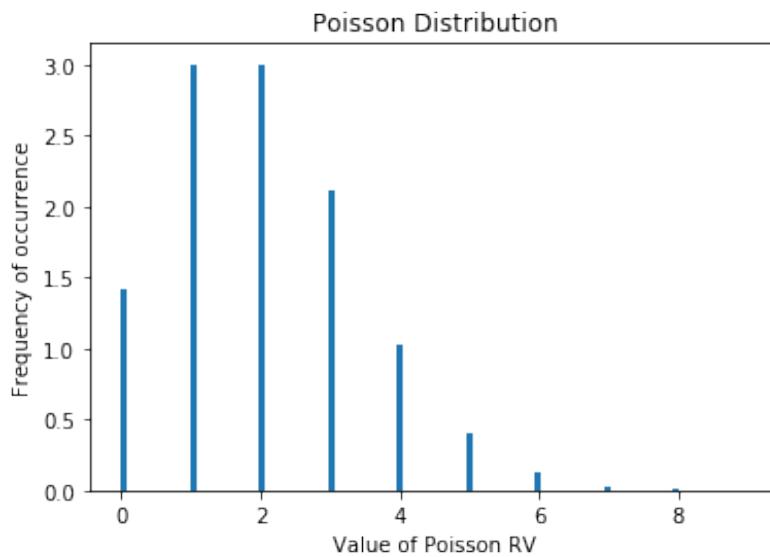
A Poisson distribution is used to estimate the number of times an event can occur within some specified time. It is used for independent events that occur at a constant rate within a given time interval.

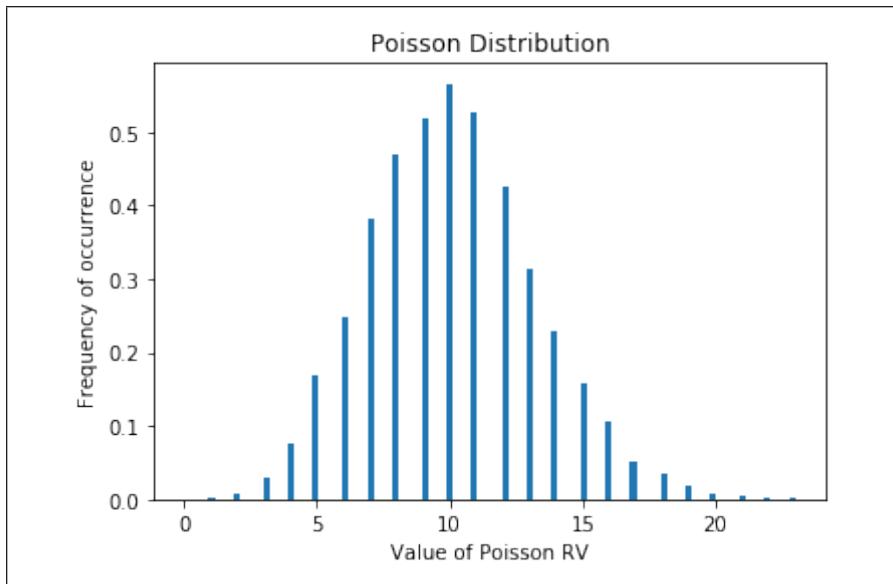
To find out the probability of observing k events in an interval is given by the aforementioned equation. For instance, the number of users visiting a website in a given interval can be thought of as a Poisson process. The number of cars passing a specific portion of a road in a given interval of time is another example of a Poisson process. If the number of cars passing in one hour is 1,000, the average number of cars passing per minute is $\mu = 1000/60$. To find the probability that not more than 10 cars pass in a minute is the sum of the probabilities for 0,1,2,3,..., 10 cars.

Type the following Python script to simulate a Poisson distribution:

```
1. from scipy.stats import poisson
2. import matplotlib.pyplot as plt
3.
4. poisson_rv1 = poisson.rvs(mu=2, size=10000)
5. plt.hist(poisson_rv1,bins=100, density = 1)
6. plt.title('Poisson Distribution')
7. plt.xlabel('Value of Poisson RV')
8. plt.ylabel('Frequency of occurrence')
9. plt.show()
10.
11.
12. poisson_rv2 = poisson.rvs(mu=10, size=10000)
13. plt.hist(poisson_rv2,bins=100, density = 1)
14. plt.title('Poisson Distribution')
15. plt.xlabel('Value of Poisson RV')
16. plt.ylabel('Frequency of occurrence')
17. plt.show()
```

Output:





It can be observed from the first plot in the output that as the average rate, the mean of the Poisson RV is decreased, it results in a skewed distribution. The first plot is a rough approximation to the Poisson distribution due to the limited number of observations. The second plot shows a distribution that is less skewed than the first because it uses an average rate of 10 instead of 2.

Further Reading

More information about functions provided by SciPy.Stats and the probability distributions using Python can be found at <https://bit.ly/3lb6JFX>

3.7 Exercise Questions

Question 1:

The median and mode of the numbers 15, 11, 9, 5, 15, 13, 17 are respectively:

- A. 13, 6
- B. 13, 18
- C. 13, 15
- D. 15, 16

Question 2:

A coin is tossed three times. The random variable that describes the number of heads has a probability of _____ for 0 heads.

- A. 1/8
- B. 2/8
- C. 3/8
- D. 0

Question 3:

Which of the following distributions are used for discrete Random Variables?

- A. Gaussian Distribution
- B. Poisson Distribution
- C. Uniform Distribution
- D. None of the mentioned.

Question 4:

When we show all possible values of a discrete random variable along with their matching probabilities is called as:

- A. Probability Density Function (PDF)
- B. Probability Mass Function (PMF)
- C. Cumulative distribution function
- D. All of the above.

Question 5:

The expectation or the mean value of a discrete random variable X can be given as:

- A. $P(X)$
- B. $\sum P(X)$
- C. $\sum X P(X)$
- D. 1

Question 6:

If the expectation of X is $E(X) = 10$ and the expectation of Y is $E(Y) = -2$, then $E(X - Y) = ?$

- A. 8
- B. -12
- C. 12
- D. Cannot be determined.

Question 7:

If a random variable assumes all negative values, it will result in _____

- A. Positive probability
- B. Negative probability
- C. Negative as well as positive probabilities
- D. All of the abovementioned options are possible.

Question 8:

If for a random variable X , $\sum P(X) = N^2 - 15$, then the value of N will be:

- A. 1
- B. 0
- C. 4
- D. Cannot be determined.

Question 9:

If the probability of a random variable X is $P(X=0) = 0.9$, and the random variable assumes either value 0 or 1, then the expectation of X , $E(X)$ would be:

- A. 1
- B. 0.1
- C. 4
- D. 0

Question 10:

A Normal distribution is symmetric about?

- A. Variance
- B. Mean
- C. X-axis
- D. Covariance

Question 11:

For a standard Normal random variable, the value of its mean is:

- A. Infinity
- B. 0
- C. 1
- D. Insufficient data

Question 12:

The spread of the normal distribution curve depends upon its:

- A. Mean
- B. Standard deviation
- C. Correlation
- D. Not given.

4

Descriptive Statistics: Measure of Central Tendency and Spread

Descriptive statistics describe the elementary features present in the sample data or the measured / observed values in a statistical experiment. Combined with the visualization tools, descriptive statistics provide a quantitative summary of the data.

Descriptive statistics, as opposed to inferential statistics, describe what is present in the data. It is not used to reach conclusions based on the observed data. Inferential statistics, the topic of chapter 6 onward, is used to make decisions after drawing conclusions from the data.

Through the use of descriptive statistics such as a simple average or mean of the data, we can summarize large amounts of observed data in a meaningful way. However, when the data is summarized, we risk losing the details present in the data.

4.1 Measuring the Central Tendency of Data

The central tendency of statistical data or a probability distribution gives us an estimate of the middle (center) of our

data or a probability distribution. There are three major types of statistics for central tendency:

- The mean,
- The median,
- The mode.

We describe them in the following subsections.

4.1.1 The Mean

The mean or the average value of given statistical data is computed by summing up all the values and then dividing by the total number of values. For a list of numbers: [3, 8, 1, 3, 6, 21, -4],

$$\text{Mean} = [3, 8, 1, 3, 6, 21, -4] / 7 = 5.43.$$

We can also compute the mean or the expected value of a random variable by using the formula for expectation, as described in Chapter 3. We take the average of all possible outcomes of a random variable to find the expected value μ as follows:

$$\mu = E(X) = \sum x.P(X=x)$$

$$\mu = E(X) = (0) (1/4) + (1) (1/2) + (2) (1/4) = 1.$$

The expected value or the mean of rolling a six-sided dice can be computed as follows:

$$\begin{aligned}\mu = E(X) &= (1) (1/6) + (2) (1/6) + (3) (1/6) + (4) (1/6) + (5) (1/6) \\ &\quad + (6) (1/6) = 3.5.\end{aligned}$$

4.1.2 The Median

The median is the statistic for the central tendency that describes the middle value. To find the median of numeric

data, we sort the numbers from smallest to largest. When we have an odd number of data points, the median is found as **(number of data points // 2) +1**, where // represents the floor division. Thus, out of 7 sorted values, the median would be the 4th value.

$$\text{Median } [-4, 1, 3, \textcolor{red}{3}, 6, 8, 21] = 3$$

If there are an even number of values, there is no middle number. In this case, the median of the list will be the mean of the middle two values within the list.

$$\text{Median } [10, 12, 14, \textcolor{red}{18}, \textcolor{red}{20}, 24, 28, 40]$$

$$= (18+20) / 2 = 19$$

4.1.3 The Mode

The mode is the value that occurs the most in the observed data. If there is no repetition of any number in the list, then there is no mode for the list.

$$\text{Mode } [-4, 1, \textcolor{red}{3}, \textcolor{red}{3}, 6, 8, 21] = 3$$

To find the statistics for the central tendency of the data, we may write the following Python script:

```

1. import statistics as st
2.
3. raindata = [2, 5, 4, 4, 0, 2, 7, 8, 8, 8, 1, 3]
4.
5. # Printing the Mean, Median and the Mode of the data
6. print("mean = %0.3f" %st.mean(raindata))
7. print("median = %0.3f" %st.median(raindata))
8. print("mode = %0.3f" %st.mode(raindata))

Output:
mean = 4.333
median = 4.000
mode = 8.000

```

The NumPy library also has functions to compute mean and median, i.e., `np.mean()` and `np.median()`. Their usage will also give the same results as `st.mean()` and `st.median()`. However, the NumPy library does not have a function to find out the mode of the data.

4.2 Measuring the Spread of Data

Spread or dispersion of the statistical data measures the variation of the data around the central tendency such as the mean. The three common measures of the spread of the data are:

- The range,
- The variance, and
- The standard deviation.

4.2.1 The Range

The range is a simple descriptive statistic to measure the spread of the data. It can be found by subtracting the minimum value present in the data from the maximum value. For instance, if the highest value in the data is 80 and the lowest is 25, the range of the given data is $80 - 25 = 55$.

4.2.2 The InterQuartile Range (IQR)

InterQuartile Range (IQR) gives us a better picture of the underlying data as compared to the simple range. In IQR, we divide our data into four quarters after we sort it in ascending order. A popular plot that shows these quartiles is known as a Box and Whisker plot shown in Figure 4.1.

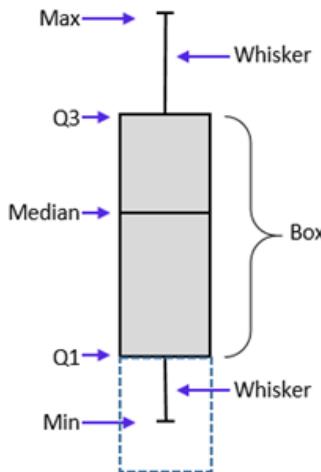


Figure 4.1: A Box and Whisker plot. The length of the box represents IQR.

In simple terms, the IQR is the difference between the third and the first quartiles or the length of the box in the plot:

$$\text{IQR} = \text{Q3} - \text{Q1}$$

To calculate the IQR, we can enter the following script.

```

1. ### InterQuartile Range
2.
3. import pandas as pd
4. student_frame = pd.DataFrame({'Student Name':
5.     ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
6.     'Sex': ['M', 'F', 'M', 'F', 'F', 'M', 'M'],
7.     'Age': [10, 14, 18, 15, 16, 15, 11],
8.     'School': ['Primary', 'High', 'High',
9.     'High', 'High', 'High', 'Primary']})
10. Q1 = student_frame.quantile(0.25)
11. # 75% or quartile 3 (Q3)
12. Q3 = student_frame.quantile(0.75)
13. # InterQuartile Range (IQR)
14. IQR = Q3 - Q1

```

```
15. print('The Q1 = %.3f, Q3 = %.3f and the InterQuartile
Range (IQR) = %0.3f' %(Q1, Q3, IQR))
```

Output:

```
The Q1 = 12.500, Q3 = 15.500 and the InterQuartile Range (IQR)
= 3.000
```

The only numeric variable in our data is ‘Age,’ whose IQR is found using the quantile () function of the Pandas library. We store our data in a Pandas DataFrame in lines 4 to 7.

Specifying any quantile such as 0.25 (25%), 0.5 (50%), and 0.75 (75%) as an argument to the quantile () function (lines 10 and 12) give the 25th percentile (the first quartile – Q1), the 50th percentile (the median – Q2), and the 75th percentile (the third quartile – Q3) of the values. The difference Q3 – Q1 in line 14 of the code gives us the IQR.

4.2.3 The Variance

Suppose we are measuring the heights of a group of people. Most measurements lie within an acceptable range. However, due to some problems with the measuring scale or a human error, one measurement is recorded as 300cm. If the minimum height in the data is 150cm, the range would be 300–150 = 150cm, which is incorrect in this case. A very small or very large unacceptable value in the collected data is called an **outlier**. The range is highly susceptible to outliers in the data, and the range is significantly exaggerated.

A better and detailed descriptive statistic to measure the spread of the data under possible outliers is the variance. The variance shows the relationship of every observed value present in the data to the mean of the data. The variance σ^2 is a measure of the variability within the data around the mean

value. The variance of a random variable X can be computed as follows:

$$\sigma^2 = E[(X - \mu)^2] = \sum_x (x - \mu)^2 f(x)$$

where E represents the mean or the expected value, μ is the mean of the data points, and $f(x)$ represents the probability distribution.

For instance, we have the following probability distribution of a random variable:

x	0	1	2	3	4
$f(x)$	0.2	0.1	0.3	0.3	0.1

To compute the variance of a random variable, we:

1. compute its mean,
2. find the squared deviation of all the possible values of the random variable from the mean,
3. calculate the expectation or the mean of the squared differences from 2.

The mean $\mu = \Sigma x f(x)$

$$= (0)(0.2) + (1)(0.1) + (2)(0.3) + (3)(0.3) + (4)(0.1) = 2.0.$$

$$E[(X-\mu)^2] = \Sigma (x - \mu)^2 f(x)$$

$$= (0 - 2)^2(0.2) + (1 - 2)^2(0.1) + (2 - 2)^2(0.3) + (3 - 2)^2(0.3) + (4 - 2)^2(0.1) = 1.6.$$

For a continuous random variable or a continuous range of data, we cannot sum up an infinite number of values. In this case, we replace the summation with integration to find out the mean or the variance.

As mentioned in Section 2.3 that in statistical studies or experiments, numerous observations collected from the study constitute the data, and an assortment of all possible outcomes of the experiment is called the **population**. Since we cannot observe the whole population, we take a **sample** that is a small portion of the population.

There is a slight difference between the variance of a sample and that of the whole population. In the calculation of population variance, we divide by N . However, in the case of sample variance, we divide by $(N-1)$, where N represents the number of data points. Therefore, we get slightly different results. To find out the variance of a population, we use the following formula:

$$\sigma^2 = \sum_x (x - \mu)^2 / N$$

For the sample variance, we use the formula given below:

$$\sigma^2 = \sum_x (x - \mu)^2 / (N - 1)$$

The reason why we divide by $(N-1)$ instead of N is that a sample taken from the population does not provide us with the complete information of the population. We can just estimate the variance of the population based on the sample data. Thus, the research in statistics suggests that subtracting one from the total number of data points, i.e., $(N-1)$, gives a better estimate of the population variance as compared to the case when we divide by N .

4.2.4 The Standard Deviation

The standard deviation is just the square root of the variance.

$$\text{STD} = \sqrt{\sigma^2} = \sigma$$

To find the range, variance, and standard deviation of the quantities, we may write the following Python script:

```

1. import statistics as st
2.
3. raindata = [2, 5, 4, 4, 0, 2, 7, 8, 8, 8, 1, 3]
4.
5. data_range = max(raindata)-min(raindata)
6. # Range of the data
7. print("The range of the data is = %.3f" % data_range)
8.
9. # population variance
10. print("population variance = %.3f" % st.
    pvariance(raindata))
11. # population standard deviation
12. print("population standard deviation = %.3f" % st.
    pstdev(raindata))
13.
14. # sample variance
15. print("variance = %.3f" % st.variance(raindata))
16. # sample standard deviation
17. print("standard deviation = %.3f" % st.stdev(raindata))
Output:
The range of the data is = 8.000
population variance = 7.556
population standard deviation = 2.749
variance = 8.242
standard deviation = 2.871

```

If the statistical data contains multiple attributes / features, we can still compute the variance and standard deviation of each feature. For example, we have been given names, ages, and grades of multiple students. We create a Pandas DataFrame to store multiple features of the data together. Pandas built-

in statistical functions are then used to compute the simple descriptive statistics of the data. The following Python script illustrates this concept:

```

1. import pandas as pd
2. #Create a Dictionary of series
3. mydata = {'Name':pd.Series(['Liam', 'Olivia', 'Noah',
   'Emma', 'Oliver', 'Ava',
   'William', 'Sophia', 'Elijah',
   'Isabella', 'James', 'Charlotte']),
4.           'Age':pd.Series([20,24,21,23,32,25,23,31,30,32,26,22]),
5.           'Grades':pd.
   Series([3.23,3.24,3.98,2.56,3.20,3.6,3.8,3.7,2.98,3.8,
   3.10,3.65])}
7. #Creating a Pandas DataFrame
8. my_df = pd.DataFrame(mydata)
9.
10. # Calculating the variance and the standard deviation of
    the data
11. print('The variance of the data is', my_df.var())
12.
13. print('\nThe standard deviation of the data is', my_
    df.std())

```

Output:

```

The variance of the data is Age      19.295455
Grades    0.174115
dtype: float64
The standard deviation of the data is Age  4.392659
Grades    0.417271
dtype: float64

```

In lines 3 to 6, we specify the features of the data as Pandas Series objects. Line 8 creates Pandas DataFrame from multiple Series objects. Finally, lines 11 and 13 compute the sample variance and the sample standard deviation of each numeric feature present in the data. Note that the descriptive statistics such as variance and standard deviation are not defined for non-numeric data such as the names of the students.

4.3 Covariance and Correlation

Up to this point in this chapter, we have provided simple descriptive statistics for univariate data. This means that we have discussed measures of central tendency of the data such as the mean, median, and mode, and measures of the spread of the data such as the range, standard deviation, and variance. These statistics are applied to a *single variable / feature/ attribute* of the data; hence, the name univariate descriptive statistics.

Suppose we want to know the relationship between two variables present in the data, such as student age and the grades they have obtained. In this case, we resort to *Bivariate Descriptive Statistics*. The most common bivariate statistics are

1. The covariance,
2. The correlation.

The covariance is used to find the relationship / dependency between two variables. It is defined as,

$$\text{Cov}(x, y) = E[(x - \mu_x)(y - \mu_y)]$$

where x and y are the two features present in the data, μ_x and μ_y are the means or expected values of x and y , and E represents the expectation operator. The covariance between two features can be positive or negative.

The terms $(x - \mu_x)$ and $(y - \mu_y)$ are computed for each data point. These are multiplied together, and finally, the mean or average of all the products is calculated to find a single number as the covariance between features x and y .

Consider the case when most of the data points result in a positive result for both terms $(x - \mu_x)$ and $(y - \mu_y)$. In this case, the product $(x - \mu_x)(y - \mu_y)$ would be positive. Moreover, if most of the data points result in a negative result for the terms $(x - \mu_x)$ and $(y - \mu_y)$ are also negative. In this case, the product $(x - \mu_x)(y - \mu_y)$ would again be positive. We shall get a positive value for the covariance. We say that there is a positive relationship between features.

Conversely, if positive $(x - \mu_x)$ values have corresponding negative $(y - \mu_y)$ values and vice versa, the product $(x - \mu_x)(y - \mu_y)$ would be negative. Hence, the covariance would be negative. A negative relationship between features is evident when we get a negative result after the computation of the covariance.

The correlation or the correlation coefficient is obtained by normalizing the covariance. It is obtained when we divide the covariance by the product of individual standard deviations of the variables.

$$\rho(x, y) = \frac{\text{Cov}(x, y)}{(\sigma_x)(\sigma_y)}$$

While the covariance can result in any arbitrary positive or negative real number, the correlation is always between -1 to 1 due to the normalization by the individual standard deviations. Thus, the correlation is used mostly to identify the strength of the relationship between two features.

To find the covariance and the correlation between two different features, we may use the following code:

```
1. import numpy as np
2. npmycov = np.cov([1, 2, 3], [1.0, 2.5, 7.5])
3. print('The covariance between x and y is \n')
4. mycov
Output:
array([[ 1.          ,  3.25       ],
       [ 3.25      , 11.58333333]])
5. mycorr = np.corrcoef([1, 2, 3], [1.0, 2.5, 7.5])
6. print('The correlation between x and y is \n')
7. mycorr
Output:
array([[1.          ,  0.95491911],
       [0.95491911,  1.        ]])
```

The output of this code shows four values instead of a single value for the covariance and the correlation. The reason is that the NumPy functions cov() and corrcoef () gives us variances and the normalized variances on diagonal entries, and the covariance and the correlation coefficient on off-diagonal entries, respectively. We see the same values on off-diagonal entries because the covariance and the correlation coefficient between x and y is the same as that of between y and x.

Further Reading

More information about descriptive statistics and Python code to implement these statistics can be found at
<https://bit.ly/3kZcwOW>

4.4 Exercise Questions

Question 1:

An outlier can be defined as:

- A. A variable that cannot be observed
- B. A variable that is hard to quantify
- C. A missing data value
- D. An extreme value

Question 2:

Variance of the data is calculated from:

- A. The Mode
- B. The Mean
- C. The Median
- D. None of the mentioned

Question 3:

Variance and standard deviation of the data:

- A. can be negative
- B. are always positive
- C. can be both negative and positive
- D. can never be zero.

Question 4:

Covariance and correlation between two features of the data:

- A. can be negative
- B. are always positive
- C. can be negative or positive
- D. can never be zero.

Question 5:

The median of the data [5, -9, 7, 6, -20, 20] is:

- A. 5
- B. 6
- C. 5.5
- D. Not given

Question 6:

The mode of the data [18, 11, 10, 12, 14, 4, 5, 11, 5, 8, 6, 3, 12, 11, 5] is:

- A. 11
- B. 5
- C. 0
- D. No unique mode

Question 7:

The range of the data [21, 18, 9, 12, 8, 14, 23] is:

- A. 23
- B. 8
- C. 15
- D. 7

Question 8

If the sample variance of the data [21, 18, 23] is 2.51, the population variance would be:

- A. less than the sample variance
- B. more than the sample variance
- C. equal to the sample variance
- D. cannot be determined.

5

Exploratory Analysis: Data Visualization

5.1 Introduction

In the previous chapter, we have seen that descriptive statistics provides a useful summary by exploring the underlying statistical data. In this chapter, we perform further exploration using plots and visualization tools.

The purpose of the exploratory analysis is to get familiarized with the structure and important features of the data. In the exploratory analysis, we employ numerous techniques to reveal the structure of the data. These include:

- **Univariate visualization** is used to generate summary statistics for each numeric variable of the data. We summarize our dataset through descriptive statistics that uses a variety of statistical measurements to better understand the dataset. Visualization tools such as bar plots and histograms are used for univariate visualization.
- **Bivariate visualization** is used to find the relationship between two variables of the data. It uses correlations,

scatter plots, and line plots to reveal the relationship between two variables of the data.

- **Multivariate visualization** is performed to understand the relationship between multiple variables of the data. It uses line plots, scatter plots, and matrices with multiple colors.

Visualization tools help us reveal the structure of variables, discover data patterns, spot anomalies such as missing values and outliers, and check assumptions about the data.

In the following sections, we present different types of visualization tools and explain the process of exploratory analysis along with practical Python examples.

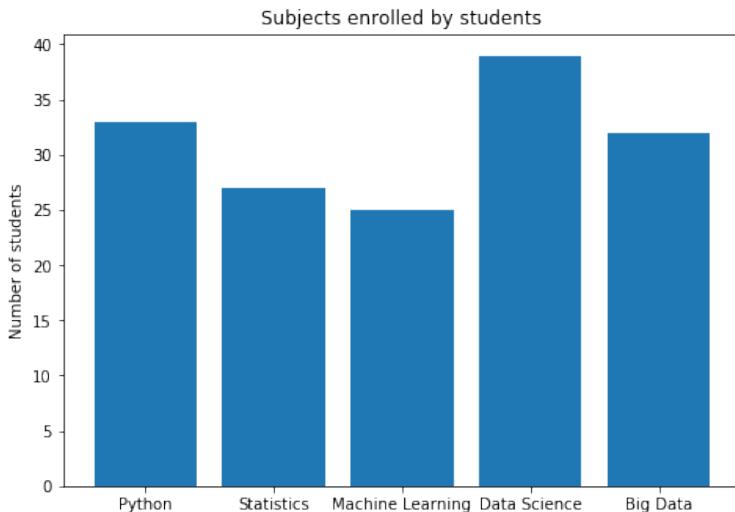
5.2 Bar (Column) Charts

If we have categorical or discrete data that can take on a small set of values, we use bar charts to show the categories as rectangular bars whose lengths are proportional to the values belonging to these categories. Bar charts are frequently referred to as column charts.

As an instance, to display the number of students studying different subjects as a bar chart, type the following Python script:

```
1. import matplotlib.pyplot as plt
2.
3. fig = plt.figure()
4. ax = fig.add_axes([0,0,1,1])
5.
6. name_of_class = ['Python', 'Statistics', 'Machine
   Learning', 'Data Science', 'Big Data']
7. students = [33,27,25,39,32]
8.
9. ax.bar(name_of_class,students)
10. plt.ylabel('Number of students')
11. plt.title('Subjects enrolled by students')
12.
13. plt.show()
```

Output:

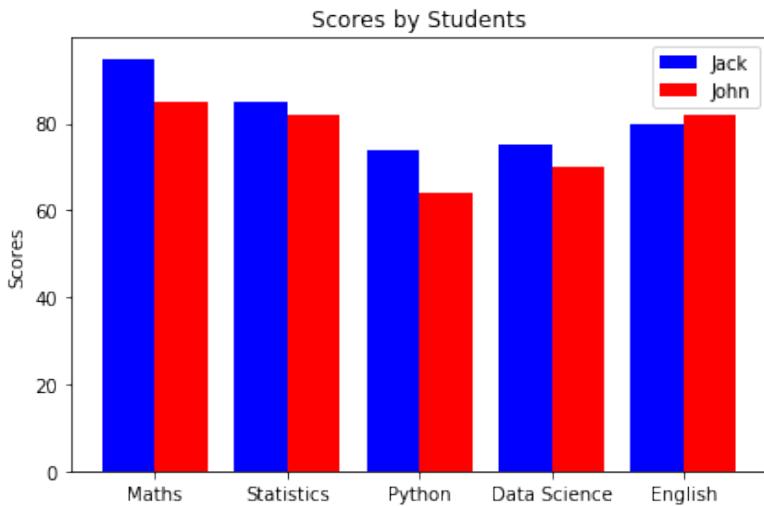


Line 3 creates a figure object whose options can be set. We use line 4 to set the size of the figure and the spacing between the subplots. The `add_axes` (`x0, y0, dx, dy`) method takes a list of four values: `x0`, `y0`, `dx`, and `dy` for the subplot. The values `x0` and `y0` are the coordinates of the lower-left corner of the subplot, and `dx` and `dy` are the width and height of the subplot, with all values specified in relative units: 0 represents

left bottom corner, and 1 represents top right corner. In line 9, we call the bar () function using the axis object ax to plot the bar chart.

To plot multiple variables on a bar chart, we may type the following Python script:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4.
5. scores_Jack = ( 95 , 85 , 74 , 75 , 80 )
6. scores_John = ( 85 , 82 , 64 , 70 , 82 )
7.
8. # create plot
9. fig, ax = plt.subplots ()
10. indexes = np.arange (len (scores_Jack))
11.
12. bar_width = 0.4
13.
14. data1 = plt.bar (indexes, scores_Jack, bar_width, color =
   'b' , label = 'Jack' )
15.
16. data2 = plt.bar (indexes + bar_width, scores_John, bar_
   width, color = 'r' , label = 'John' )
17.
18. plt.ylabel ( 'Scores' )
19. plt.title ( 'Scores by Students' )
20. plt.xticks (indexes + bar_width / 2 , ( 'Maths' ,
   'Statistics' , 'Python' , 'Data Science' , 'English' ))
21. plt.legend ()
22.
23. plt.tight_layout ()
24. plt.show ()
```

Output:

In the bar chart given above, we plot the scores obtained by two students in five different subjects. We create variable *indexes* that is used as an independent variable to keep track of location on the x-axis to plot the scores obtained in different subjects. Note that in line 16, we use *indexes + bar_width* instead of *indexes* as used in line 14 to specify the location of the bars for the student, John. This is done to separate bars of both students from each other. Line 20 shows the use of the method `xticks()` to specify the location of the subject names in the middle of both bars, i.e., *indexes + bar_width / 2*.

We can add titles, labels, and legends to the generated plots. To add titles, labels, and legend to a plot, we use the `title`, `xlabel`, `ylabel`, and `legend` methods of the `pyplot` module, respectively. We pass string values to these methods, which appear on the plots, as shown in the output.

5.3 Pie Charts

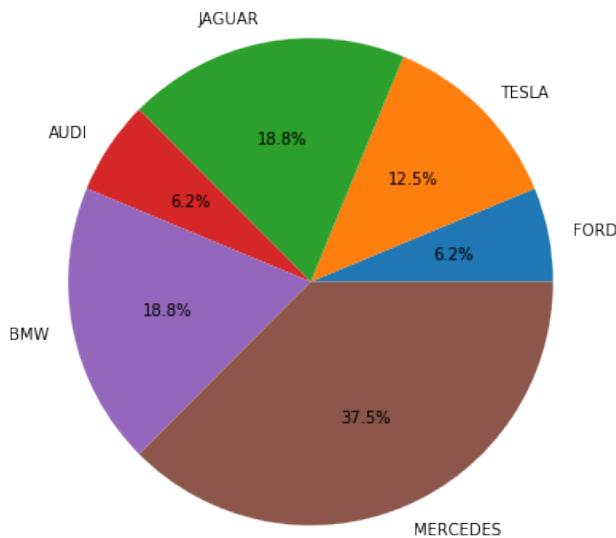
A pie chart, a circular statistical chart, is used to display the percentage distribution of **categorical variables**. Categorical variables are those variables that represent the categories such as gender, name of countries, and type of pet. The area of the whole chart represents 100 percent or the whole data. The areas of the pies in the chart denote the percentage of shares the categories have in the data.

Pie charts are popular in business communications because they give a quick summary of business events such as sales and operations. Pie charts can be used, for example, to summarize results from surveys and usage of memory in a computer system.

To draw a pie chart, we use the function `pie()` in the `pyplot` module. The following Python code draws a pie chart showing the number of cars by types.

```
1. # Import libraries
2. from matplotlib import pyplot as plt
3. import numpy as np
4.
5. cars = ['FORD', 'TESLA', 'JAGUAR', 'AUDI', 'BMW',
       'MERCEDES']
6.
7. numbers_cars = [13, 26, 39, 13, 39, 78]
8.
9. fig = plt.figure(figsize =(10, 7))
10. plt.pie(numbers_cars, labels=cars, autopct='%1.1f%')
11.
12. plt.show()
```

Output:



The `autopct='%.1f%%'` string formatting is used for the formatting of how the percentages appear on the pie chart.

Further Readings—Matplotlib Plots

To study more about Matplotlib plots, please check Matplotlib's official documentation for plots.

<https://bit.ly/3jQTBuQ>

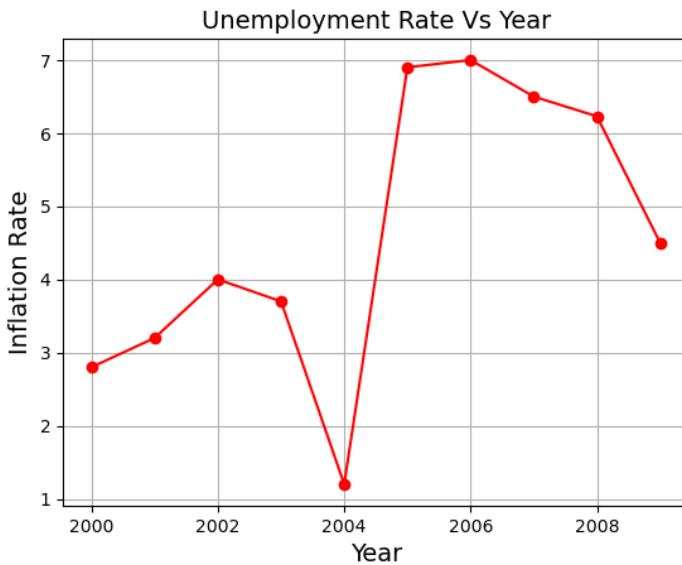
You can explore more features of Matplotlib by searching and reading this documentation.

5.4 Line Plots for Continuous Data

Line plots are useful to convey the behavior of one or more variables that change over space or time. Line plots display the trend of data along a scale divided into equal intervals. Let us generate a simple line plot.

```
1. import matplotlib.pyplot as plt
2.
3. Year = [2000,2001,2002,2003,2004,2005,2006,2007,2008,2009]
4. inflation_rate = [2.8, 3.2, 4, 3.7, 1.2, 6.9, 7, 6.5, 6.23,
   4.5]
5.
6. plt.plot(Year, inflation_rate, color='red', marker='o')
7. plt.title('Inflation Rate Vs Year', fontsize=14)
8. plt.xlabel('Year', fontsize=14)
9. plt.ylabel('Inflation Rate', fontsize=14)
10. plt.grid(True)
11. plt.show()
```

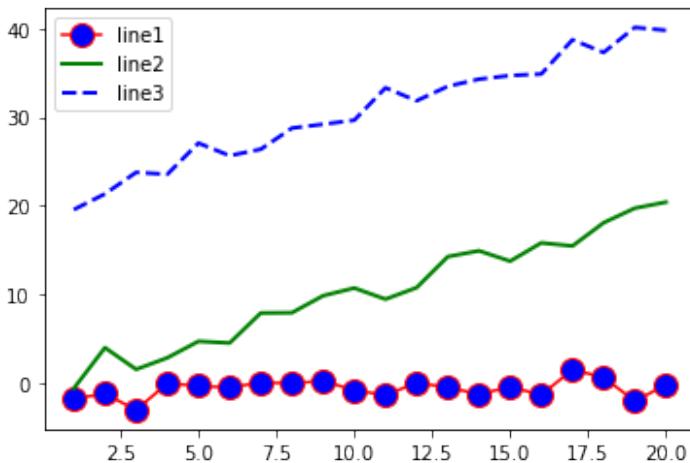
Output:



In this graph, we have plotted the unemployment rate of adults belonging to an arbitrary location against the years from 2000 to 2009. We use the function `plot()` from the `pyplot` module of Matplotlib, pass it the values for the values of years, and the unemployment rate as x and y axes to generate a line plot.

Multiple plots can be drawn on the same figure as shown in the Python script given below:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. import pandas as pd
4.
5. df=pd.DataFrame(
6.     {'x': range(1,21),
7.      'line1': np.random.randn(20),
8.      'line2': np.random.randn(20)+range(1,21),
9.      'line3': np.random.randn(20)+range(21,41)
10.     })
11.
12. # multiple line plots
13. plt.plot('x', 'line1', data=df, marker='o',
14.           markerfacecolor='blue', markersize=12, color='r')
15. plt.plot('x', 'line2', data=df, marker='', color='g',
16.           linewidth=2)
17. plt.plot('x', 'line3', data=df, marker='', color='b',
18.           linewidth=2, linestyle='dashed')
19. plt.legend()
20. Output:
```



Here, we plot three different data variables: line1, line2, and line 3 against the variable x. We have used DataFrame from the Pandas library to save all four variables. We have used different options inside plt.plot() function for distinguishing

plots from each other and for better visualization. The options marker, marker face color, color, line width, and marker size are used to adjust the marker type, line color, line thickness, and marker size, respectively.

5.5 Scatter Plot

A scatter plot is used to visualize the relationship between two variables in two-dimensions. It uses dots or marks to plot values of two variables, one along the x-axis and the other along the y-axis.

If an increase in one variable causes an increase in another variable and vice versa, we can conclude that there is a positive linear relationship between two variables. However, if increasing the first variable reveals a decrease in the second variable, we say that there is a negative linear relationship between both variables.

To plot a scatter plot, type the following piece of code:

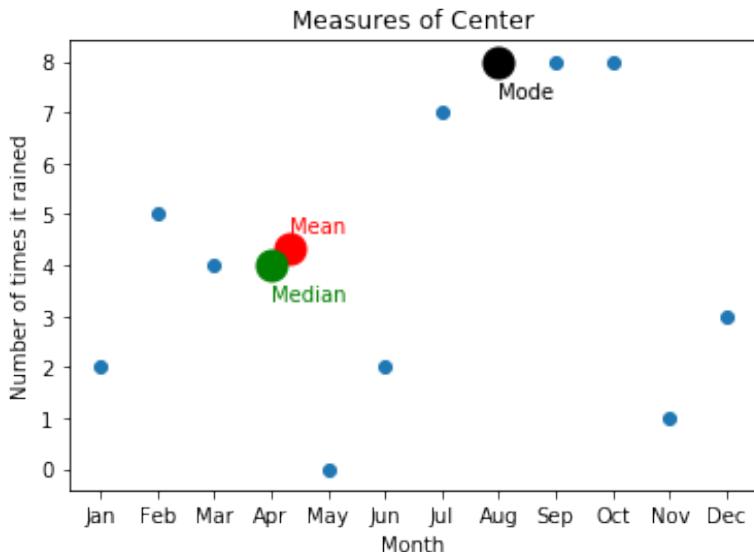
```
1. import matplotlib.pyplot as plt
2. import statistics as st
3.
4. month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
   'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
5. months = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6.
7. fig, ax = plt.subplots(nrows=1, ncols =1)
8.
9. ax.set_title("Measures of Center")
10. ax.set_xlabel("Month")
11. ax.set_ylabel("Number of times it rained")
12.
13. ax.scatter([1,2,3,4,5,6,7,8,9,10,11,12],raindata)
14.
15. plt.xticks(np.arange(12)+1, month_names, color = 'black')
```

```

16.
17. # draw points for mean, median, mode
18. ax.plot([st.mean(raindata)], [st.mean(raindata)],
   color='r', marker="o", markersize=15)
19. ax.plot([st.median(raindata)], [st.median(raindata)],
   color='g', marker="o", markersize=15)
20. ax.plot([st.mode(raindata)], [st.mode(raindata)],
   color='k', marker="o", markersize=15)
21.
22. # Annotation
23. plt.annotate("Mean", (st.mean(raindata),
   st.mean(raindata)+0.3),color="r")
24. plt.annotate("Median", (st.median(raindata),
   st.median(raindata)-0.7),color="g")
25. plt.annotate("Mode", (st.mode(raindata), st.mode(raindata)-
   0.7),color="k")
26.
27. plt.show()

```

Output:



We plot measures of the center of the given data in this plot. After settings display options in lines 7 to 11, the function

scatter () in line 13 plots the number of rainy days in 12 months of the year.

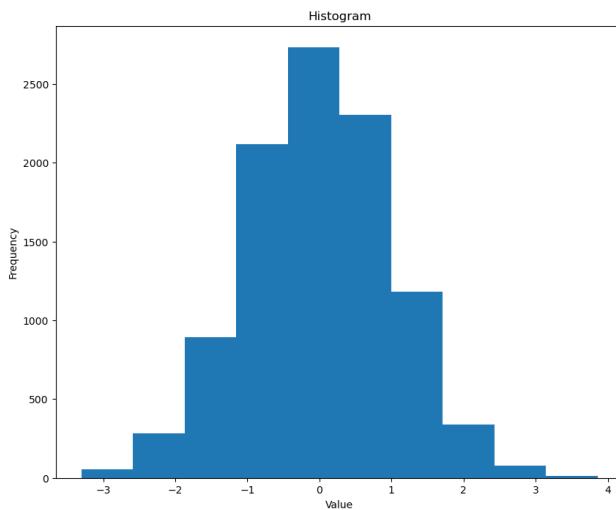
We plot the mean, the median, and the mode of the given data as well using the commands given in lines 18 to 20. Lines 23 to 25 are used to annotate the measures of the center of the data.

5.6 Histogram

A histogram is a bar chart that shows the frequency distribution or shape of a numeric feature in the data. This allows us to discover the underlying distribution of the data by visual inspection. To plot a histogram, we pass a collection of numeric values to the method hist () of the Matplotlib.pyplot package.

For example, the following code plots the distribution of values of a Normal random variable.

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. #Creating a normal random variable
5. randomNumbers = np.random.normal(size=10000)
6.
7. #Draw a histogram
8. plt.figure(figsize=[10,8])
9.
10. plt.hist(randomNumbers)
11. plt.title("Histogram")
12. plt.xlabel("Value")
13. plt.ylabel("Frequency")
14. plt.show()
```

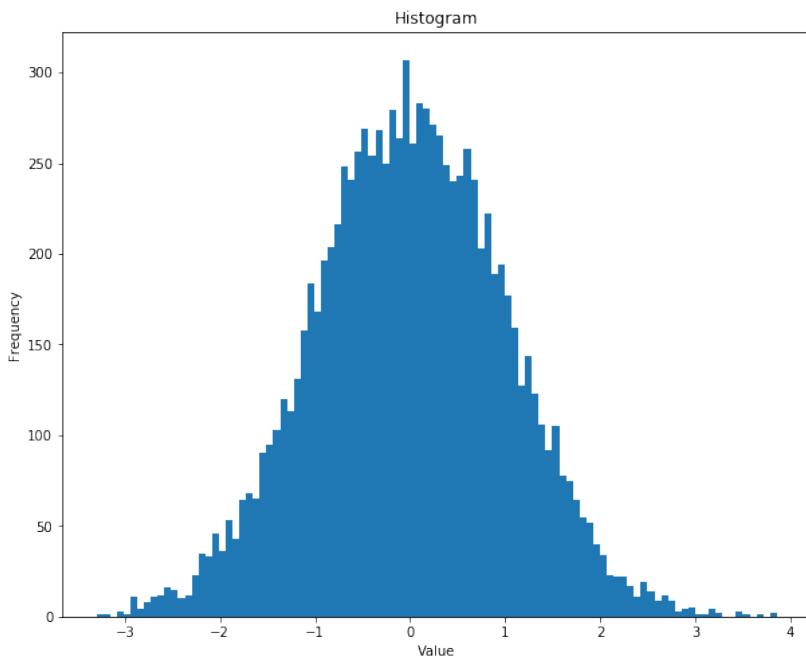
Output:

SciPyStats package can be used to generate random variables. However, here we have used NumPy's `random.normal()` method to generate a Normally distributed data. The default settings of this method generate a random variable of zero mean and unit standard deviation.

The plot shown in the output of the program reveals that more than 2,500 data points out of 10,000 have a value around 0. A few values are less than -3 and greater than 3. By default, the method `hist()` uses 10 bins or groups to plot the distribution of the data. We can change the number of bins in line 10 of the code by using the option **bins**.

```
1. plt.hist(randomNumbers, bins=100)
```

Output:

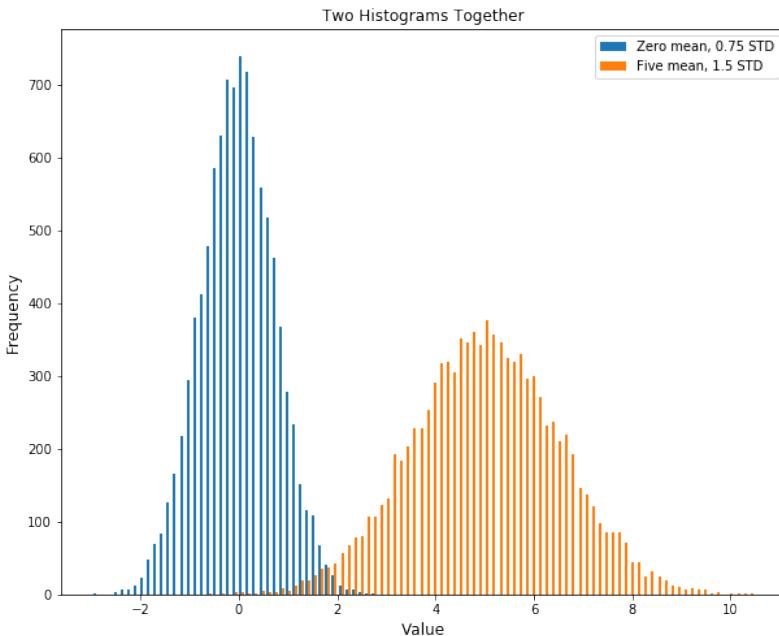


This plot is smoother than the previous one, which was generated using 10 bins. It is obvious from this plot that the generated data follows a Normal or Gaussian distribution (bell curve).

It is also possible to generate multiple histograms on the same plot. The following example illustrates this concept.

```
1. plt.figure(figsize=[10,8])
2.
3. # Creating random numbers using numpy
4. x = 0.75 * np.random.randn(10000)
5. y = 1.5 * np.random.randn(10000) + 5
6.
7. plt.hist([x, y], bins=100, label=['Zero mean, 0.75
   STD','Five mean, 1.5 STD'])
8. plt.xlabel('Value', fontsize=12)
9. plt.ylabel('Frequency', fontsize=12)
10. plt.title('Two Histograms Together', fontsize=12)
11. plt.legend()
12.
13. plt.show()
```

Output:



In line 1 of the code, we adjust the size of the figure. Lines 4 and 5 create two Normal random variables. The multiplying factors 0.75 and 1.5 specify the standard deviations of both variables x and y, respectively. The addition of 5 in line 5 shifts

the center of y from 0 to 5. We specify labels in the function plt.hist () in line 7. The invocation of plt.legend() puts these labels on the generated plot.

5.7 Creating a Frequency Distribution

A frequency distribution indicates the frequency of occurrence of various outcomes in the sample data. Sometimes, the frequency distribution is displayed as a plot or a graph. We can either use a bar chart or a histogram to plot the frequency distribution. Each entry of this distribution shows the frequency of occurrence of different values within specific groups. As an instance, the following table shows the frequency of people having weight in the specified range.

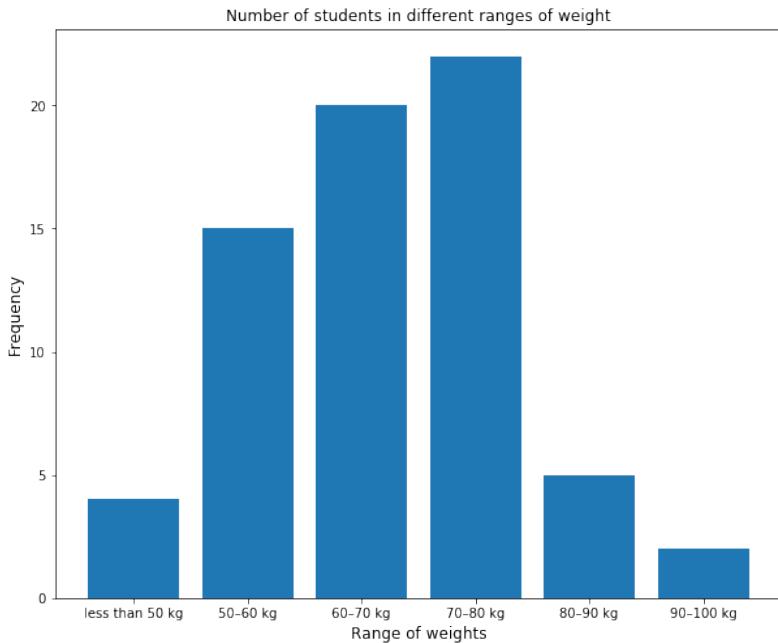
Table 5.1: Frequency of occurrence of people having weight in the specified range.

Weight range	Frequency
less than 50 kg	4
50-60 kg	15
60-70 kg	20
70-80 kg	22
80-90 kg	5
90-100 kg	2

The following code utilizes a bar chart to plot the frequency distribution given in Table 5.1.

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4. weight_range = ['less than 50 kg', '50-60 kg', '60-70 kg',
   '70-80 kg', '80-90 kg', '90-100 kg']
5. num_students = [4, 15, 20, 22, 5, 2]
6.
7. # plotting the frequency distribution
8. plt.figure(figsize=[10,8])
9.
10. plt.bar(weight_range, num_students)
11. plt.xlabel('Range of weights', fontsize=12)
12. plt.ylabel('Frequency', fontsize=12)
13. plt.title('Number of students in different ranges of
   weight', fontsize=12)
14. plt.show()
```

Output:

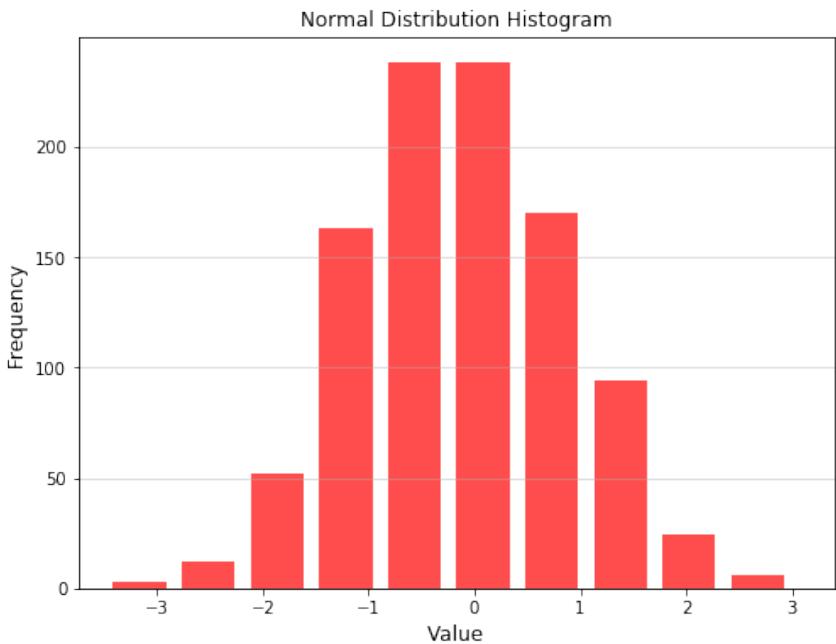


The bar () method takes values of x and y axes and plots the values of y as vertical bars.

When we have information on both x and y axes, we use a bar chart to show the frequency distribution. However, there are cases when we do have values for the y-axis, but we do not know what values of x they belong to. In this case, it is more convenient to use the histogram to plot the frequency distribution. The histogram splits the data into small equal-sized bins and places the frequency of occurrence of y variables in the respective bins. We do have the option to change the number of bins for a better display of the data.

The following code plots the frequency distribution as a histogram:

```
1. #Plotting the Frequency Distribution with Histogram
2. import numpy as np
3. import matplotlib.pyplot as plt
4.
5. #Creating a normal random variable
6. randomNumbers = np.random.normal(size=1000)
7.
8. plt.figure(figsize=[8,6])
9. plt.hist(randomNumbers, width = 0.5, color='r',alpha=0.7)
10. plt.grid(axis='y', alpha=0.5)
11. plt.xlabel('Value',fontsize=12)
12. plt.ylabel('Frequency',fontsize=12)
13. plt.title('Normal Distribution Histogram',fontsize=12)
14. plt.show()
```

Output:

In this example, we generate 1,000 data points from the normal distribution in line 6. Since we do not have the information on the number of groups to divide our data into, we use the histogram instead of the simple bar chart. We draw the histogram using the `hist()` function in line 9 of the code. The options `width`, `color`, and `alpha` are used to adjust the width of the bars, their color, and the color transparency, respectively.

5.8 Relation between PMF, PDF, and Frequency Distribution

It is important to know that there is a close link between frequency distributions and the probability mass and density functions. If we normalize the frequency of occurrence given on the y-axis of a frequency distribution plot (either bar chart

or a histogram), we get the normalized or relative frequency. The sum of all the relative frequencies of occurrence of all the events or groups of data equates to 1.

We also know that the sum of probabilities of all the possible outcomes related to the values of random variables is also 1. Thus, once we have a frequency distribution plot of a random variable, we normalize its y-axis, i.e., we get the relative frequency. In this way, we obtain the probability function of the random variable. Now, if the random variable is discrete, we get the probability mass function (PMF). For continuous random variables, we get the probability density function (PDF).

5.9 Cumulative Frequency Distribution and Cumulative Distribution Function (CDF)

A cumulative frequency distribution represents the sum of frequencies of occurrences of a group and all groups below it in a frequency distribution. This implies that we add up the frequencies of all the groups below a certain group to find the cumulative frequency of that group. We continue the example given in Table 5.1 to draw the cumulative frequency distribution.

Table 5.2: Cumulative frequency of occurrence of people having a weight equal to or less than a particular range.

Weight range	Frequency	Cumulative Frequency
less than 50 kg	4	4
50–60 kg	15	19
60–70 kg	20	39
70–80 kg	22	61
80–90 kg	5	66
90–100 kg	2	68

If we have to find out the number of people who have their weights up to 70 kg, we have to add the frequency of people in the range of less than 50 kg, 50–60 kg, and 60–70 kg. In this case, we shall get a cumulative frequency of 39 given in the 3rd column of Table 5.2.

The following code plots the cumulative frequency distribution as a bar chart:

```

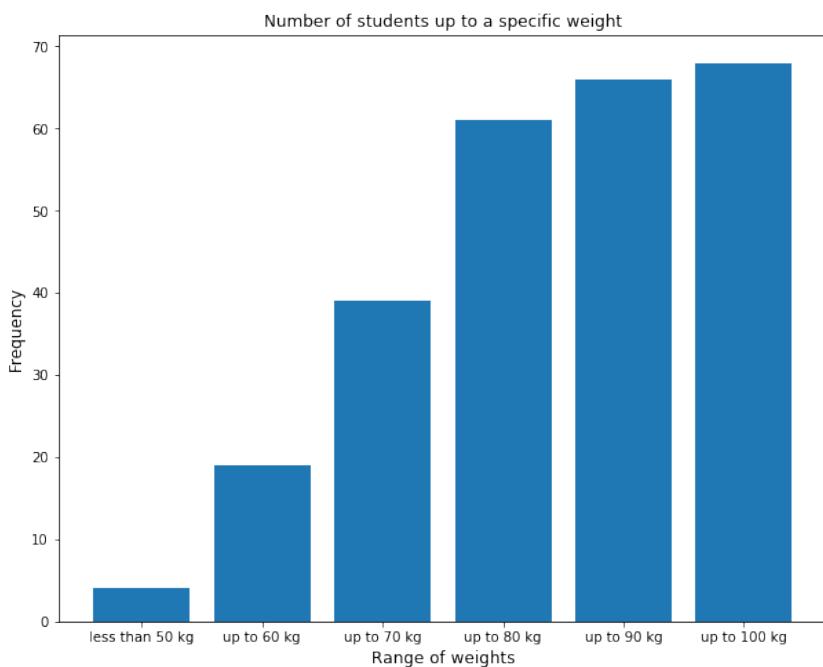
1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. weight_range = ['less than 50 kg', 'up to 60 kg', 'up to
    70 kg',
5.                 'up to 80 kg', 'up to 90 kg', 'up to 100 kg']
6.
7. num_students = [4, 15, 20, 22, 5, 2]
8. cum_freq_students = np.cumsum(freq_students)
9.
10. # plotting the frequency distribution
11. plt.figure(figsize=[10,8])
12.
13. plt.bar(weight_range, cum_freq_students)
14. plt.xlabel('Range of weights', fontsize=12)
15. plt.ylabel('Frequency', fontsize=12)

```

```
16. plt.title('Number of students up to a specific weight',  
    fontsize=12)
```

```
17. plt.show()
```

Output:



The output of the program shows that the cumulative frequency distribution is an increasing function because the frequency of occurrence is never negative.

For the sake of completeness, we plot the cumulative frequency distribution as a continuous line. Suppose we want to know the number / percentage of people among a group who own at least a specific number of cars. We have a total of 600 people who have at least one car. We use Pandas to store the data and first compute its frequency distribution as follows:

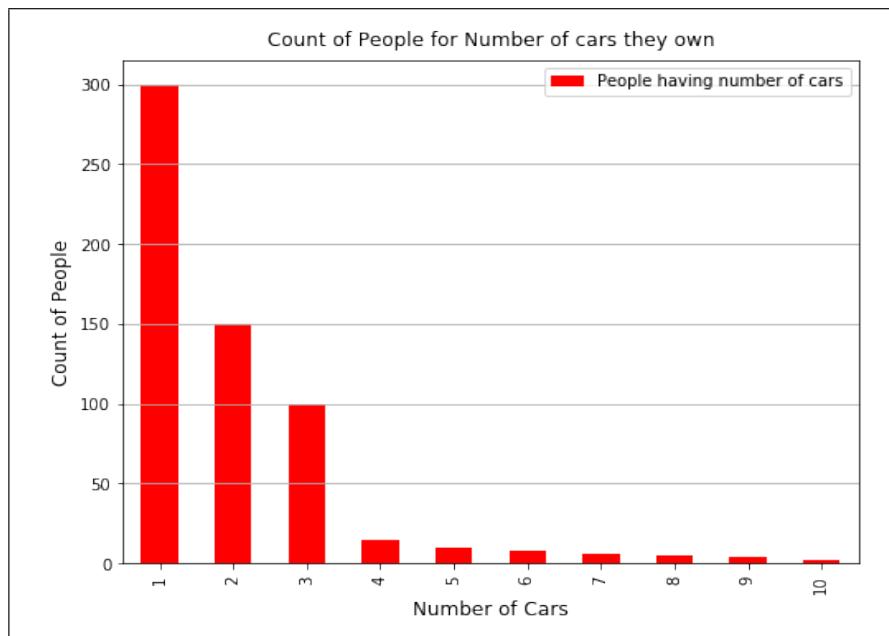
```
1. import pandas as pd
2. import matplotlib.pyplot as plt
3.
4. count_people = 600
5.
6. people_car_data = {'Number of cars': [1, 2, 3, 4, 5, 6, 7,
8, 9, 10],
7.                     'People having number of cars': [300,
150, 100, 15, 10, 8, 6, 5, 4, 2]}
8.
9. df = pd.DataFrame(data=people_car_data)
10. print(df)
```

```
11.
12. df.plot(kind='bar', x='Number of cars', y='People having
number of cars',
13.           figsize=(8, 6), color='r');
14.
15. plt.grid(axis='y', alpha=1)
16. plt.title("Count of People for Number of cars they own",
y=1.01, fontsize=12)
17. plt.ylabel("Count of People", fontsize=12)
18. plt.xlabel("Number of Cars", fontsize=12)
```

Output:

	Number of cars	People having number of cars
0	1	300
1	2	150
2	3	100
3	4	15
4	5	10
5	6	8
6	7	6
7	8	5
8	9	4
9	10	2

```
Text(0.5, 0, 'Number of Cars')
```



From this frequency distribution, we compute and plot the cumulative frequency distribution as follows.

```

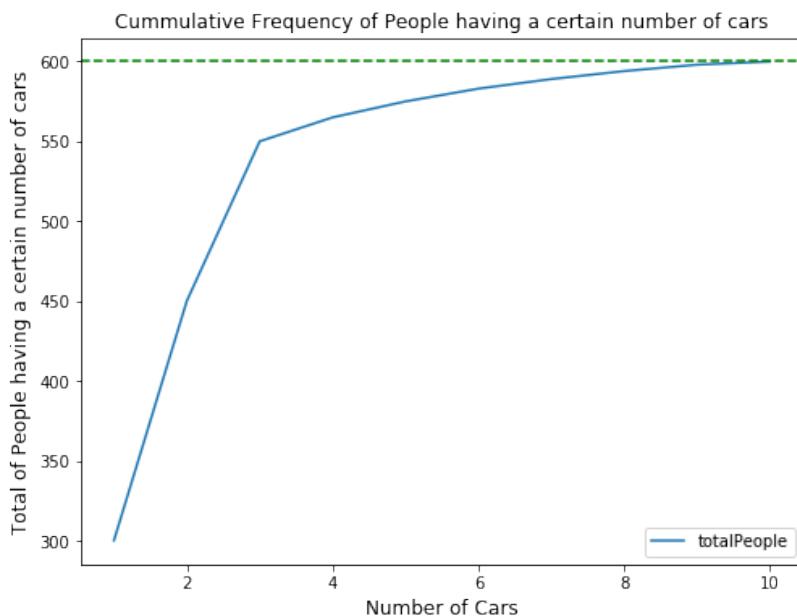
1. # Cummulative Frequency Graph
2.
3. df['totalPeople'] = df['People having number of cars'].
   cumsum()
4. print(df)
5.
6. df.plot(x='Number of cars', y='totalPeople', kind='line',
7.           figsize=(8, 6))
8.
9. plt.axhline(y=count_people, color='green', linestyle='--')
10. plt.title("Cummulative Frequency of People having a
    certain number of cars", fontsize=12)
11. plt.ylabel("Total of People having a certain number of
    cars", fontsize=12)
12. plt.xlabel("Number of Cars", fontsize=12)

```

Output:

	Number of cars	People having number of cars	totalPeople
0	1	300	300
1	2	150	450
2	3	100	550
3	4	15	565
4	5	10	575
5	6	8	583
6	7	6	589
7	8	5	594
8	9	4	598
9	10	2	600

```
Text(0.5, 0, 'Number of Cars')
```



A cumulative frequency distribution plot of a random variable with the normalized y-axis, i.e., relative frequency, gives us the cumulative distribution function (CDF). Dividing the right side of the command given in line 3 of the previous program, `df['totalPeople'] = df['People having a number of cars']`.

cumsum(), results in the normalized y-axis. Hence, we get the cumulative distribution function (CDF). Mathematically,

$$F(x) = P(X \leq x)$$

where $F(x)$ is the CDF, and the right-hand side of the equation says that the probability of the random variable X having any value equal to or less than value x .

5.10 The Quantile Function

A quantile is a cut point that divides the range of a probability distribution or sample observations into equal-sized intervals having equal probabilities. A famous example of a quantile is the median of the data or a probability distribution. The median is the point such that half of the data have values less than the median, and the remaining half values are greater than the median. The median is also referred to as 2-quantile.

The division of distribution into four equal parts constitutes four **quartiles**, whereas the division into 100 equal parts makes up 100 **percentiles**. When the distribution is divided into 10 equal parts, we get 10 **deciles**.

The **quantile function**, also known as **inverse CDF** or the **percent-point function (PPF)**, is associated with the distribution of random variables. While a CDF tells us the probability of random variable X to have a value equal to or less than a specific value x , a quantile function tells us the value of the random variable such that the probability of the variable being less than or equal to that value equals the given probability. Mathematically, the quantile function $Q(P)$ is given as

$$Q(P) = x$$

or

$$Q(P) = \min \{x : F(x) \geq p\} \quad \forall p \in (0, 1)$$

where min on the right side of the equation means the quantile function returns the minimum value of x from all those values such that their distribution $F(x)$ equals or exceeds probability p , and “ p ” means all probability values lie in 0 to 1 range. The value x returned by $Q(P)$ obeys the CDF equation.

$$F(x) = P(X \leq x)$$

The PPF or the quantile function can be used to get the values/samples of the variable X from the given distribution. If $F(x)$ is the distribution function, we can use the quantile function to generate the random variable that has $F(x)$ as its distributions function.

Example: Suppose a random variable X has the following probability mass $P(X)$ and the probability distribution $F(X)$

$$X=[-2 \quad 0 \quad 1 \quad 3]$$

$$P(X)=[0.2 \quad 0.1 \quad 0.3 \quad 0.4]$$

$$F(X)=[0.2 \quad 0.3 \quad 0.6 \quad 1]$$

We plot both $F(X)$ and the quantile function using the following Python script.

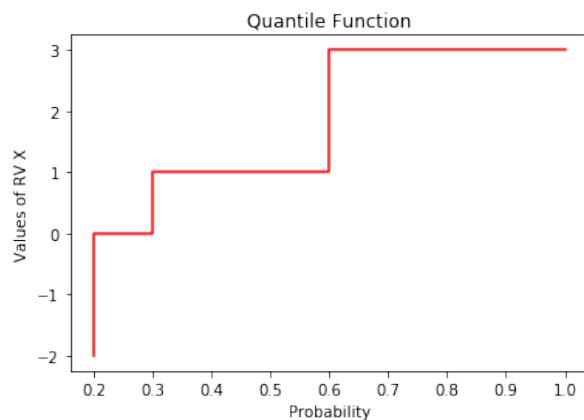
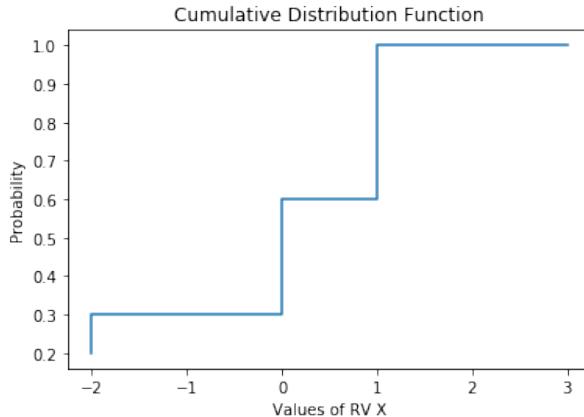
```

1. ## Quantile function
2.
3. import numpy as np
4. import matplotlib.pyplot as plt
5.
6. x = [-2, 0, 1, 3]
7. cdf_func = [0.2, 0.3, 0.6, 1]
8.

```

```
9. ## Plot of the cumulative distribution function
10. plt.step(x,cdf_func)
11. plt.xlabel('Values of RV X')
12. plt.ylabel('Probability')
13. plt.title('Cumulative Distribution Function')
14. plt.show()
15.
16. ## Plot of the quantile function
17. plt.step(cdf_func,x, color='r')
18. plt.xlabel('Probability')
19. plt.ylabel('Values of RV X')
20. plt.title('Quantile Function')
21. plt.show()
```

Output:



It is obvious from the outputs that the quantile function is obtained by reflecting the graph of the CDF about the horizontal axis and rotating the resulting graph in the counter-clockwise direction. A value of the RV X , for example, 1 in the CDF graph, gives us 0.6 value of the probability.

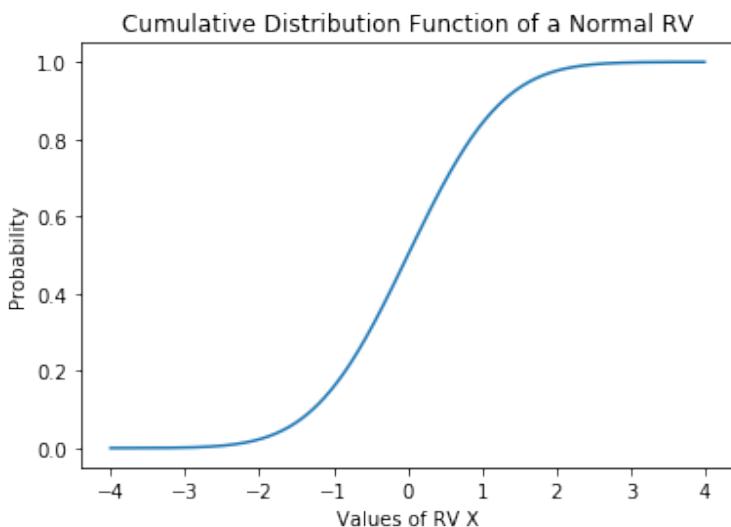
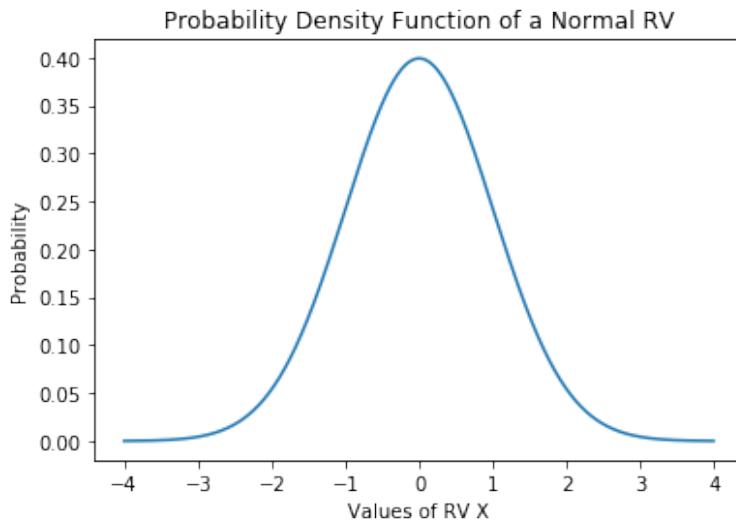
In a similar way, getting any value of X less than or equal to 1 has a corresponding probability value of 0.6 in the quantile plot. The horizontal axis goes from zero to one as it is a probability. The vertical axis also goes from the smallest value to the largest value of the cumulative distribution function of the random variable.

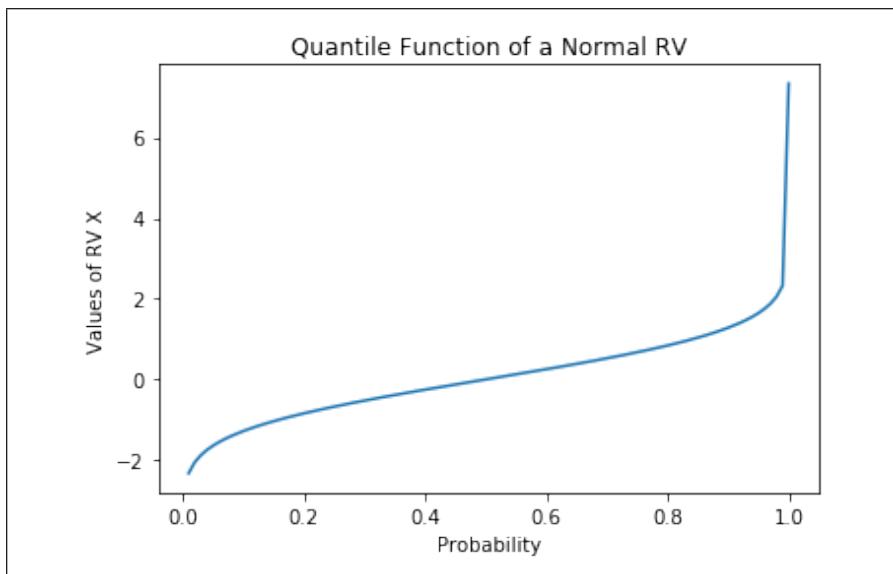
For a normal RV, we plot its PDF, CDF, and the quantile function using the Python script given below:

```
1. from scipy.stats import norm
2. import numpy as np
3. import matplotlib.pyplot as plt
4.
5. # Generating a range of values from -4 to 4 because a
   standard Normal RV has most values between -3 to 3
6. x= np.arange(-4,4,0.01)
7.
8. # Plot of PDF
9. plt.plot(x,norm.pdf(x))
10. plt.xlabel('Values of RV X')
11. plt.ylabel('Probability')
12. plt.title('Probability Density Function of a Normal RV')
13. plt.show()
14.
15. # Plot of CDF
16. plt.plot(x,norm.cdf(x))
17. plt.xlabel('Values of RV X')
18. plt.ylabel('Probability')
19. plt.title('Cumulative Distribution Function of a Normal
   RV')
20. plt.show()
```

```
21.  
22. # Plot of Inverse CDF (or PPF or Quantile function)  
23. plt.plot(x, norm.ppf(x))  
24. plt.xlabel('Probability')  
25. plt.ylabel('Values of RV X')  
26. plt.title('Quantile Function of a Normal RV')  
27. plt.show()
```

Output:





5.11 The Empirical Distribution Function

The empirical distribution function (EDF) or simply empirical distribution is used to describe a sample of observations of a variable. The value of this distribution at a given point equals the proportion of observations from the sample that are less than or equal to the point.

EDF is a cumulative distribution associated with a sample. It increases by $1/n$ at each of the n observations. Its value at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value.

Suppose a sample of size n has the following observations: $S = [x_1, x_2, \dots, x_n]$. The empirical distribution function of the whole sample is given as:

$$F_n(x) = \frac{1}{n} \sum_i (1) \quad \text{if } x_i \leq x$$

The value of the empirical distribution at a specific point x can be calculated by counting the number of observations that are less than or equal to x . Finally, divide the counted number by the total number of observations. Thus, we obtain the proportion of observations less than or equal to x .

We compute the empirical distribution function for the following sample:

$$S = [0.3 \quad 0.4 \quad 0.3 \quad 0.0 \quad 0.5 \quad 0.6 \quad 0.8 \quad -0.5]$$

We sort elements of S as follows:

$$S = [-0.5 \quad 0.0 \quad 0.3 \quad 0.3 \quad 0.4 \quad 0.5 \quad 0.6 \quad 0.8]$$

The EDF of this sample is given as:

$$F_n(X) = [1/8 \quad 2/8 \quad 3/8 \quad 4/8 \quad 5/8 \quad 6/8 \quad 7/8 \quad 1]$$

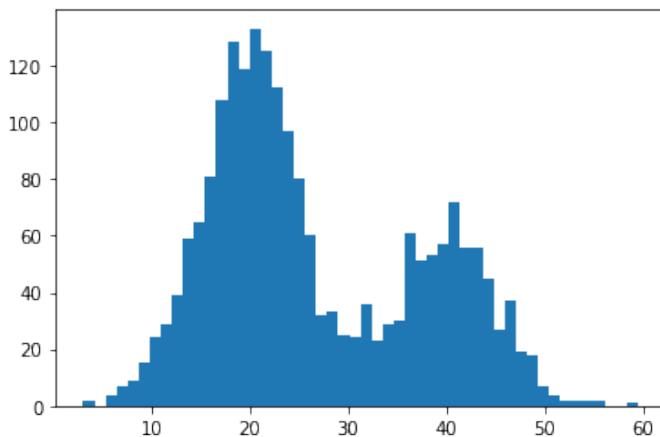
The EDF gives us an estimate of the cumulative distribution function from which the points in the sample are generated.

A Python script to estimate EDF is given below:

```
1. from matplotlib import pyplot
2. from numpy.random import normal
3. from statsmodels.distributions.empirical_distribution
   import ECDF
4. import numpy as np
5.
6. # generate a sample
7. sample1 = normal(loc=40, scale=5, size=700)
8. sample2 = normal(loc=20, scale=5, size=1400)
9. sample = np.concatenate((sample1, sample2))
10.
11. # plot the histogram
12. pyplot.hist(sample, bins=50)
13. pyplot.show()
14.
15. # fit a edf
16. ecdf = ECDF(sample)
```

```
17.  
18. # get the cumulative probability for values  
19. print('P(x<25): %.3f' % ecdf(25))  
20. print('P(x<50): %.3f' % ecdf(50))  
21. print('P(x<75): %.3f' % ecdf(75))  
22.  
23. # plot the edf  
24. pyplot.plot(ecdf.x, ecdf.y)  
25. pyplot.show()
```

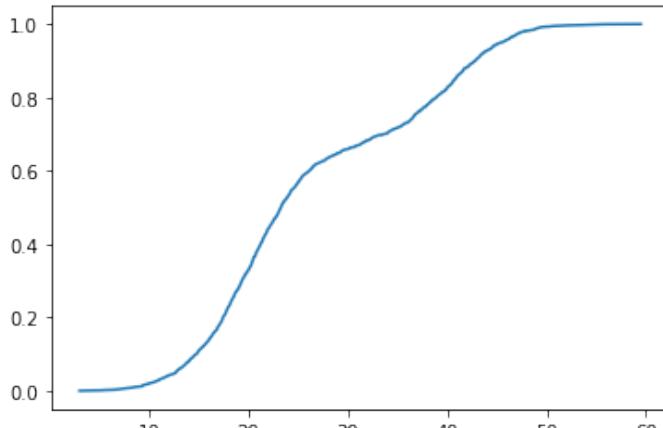
Output:



P(x<25): 0.569

P(x<50): 0.993

P(x<75): 1.000



The output of the program first shows the histogram of the generated sample. This sample consists of two Normal distributions. We show the cumulative probability for values of $x = 25, 50$, and 75 in the output. It can be seen that more than 99 percent of the samples have a value of x less than or equal to 50. The final plot shows the estimated empirical distribution function that converges to 1.

Hands-on Time – Exercise

To check your understanding of data plotting and visualization for exploratory analysis, complete the following exercise questions. The answers to the questions are given at the end of the book.

5.12 Exercise Questions

Question 1:

We generate a _____ to plot the frequency distribution of a data variable.

- A. Bar plot
- B. Pie plot
- C. Histogram
- D. Any of the above

Question 2:

When we decrease the number of bins in a histogram, the plot becomes:

- A. Expanded
- B. Contracted
- C. Smooth
- D. rough

Question 3:

Matplotlib's function plt.plot() is used to create a:

- A. Scatter plot
- B. Line plot
- C. Pie plot
- D. Any of the above

Question 4:

The range of the values plotted on the y-axis of a cumulative distribution plot is

- A. 0 - 1
- B. -3 - 3
- C. 0 - infinity
- D. Depends upon the values of the random variable

Question 5:

The range of the values plotted on the y-axis of a quantile plot is:

- A. 0 - 1
- B. -3 - 3
- C. 0 - infinity
- D. Depends upon the values of the random variable

Question 6:

The pie chart is used when we have _____ data.

- A. Continuous
- B. Discrete
- C. Either continuous or discrete, one type at a time.
- D. Both continuous and discrete together.

Question 7:

Matplotlib's function plt.plot () can plot _____

- A. time-varying data
- B. space varying data
- C. time and space varying data
- D. any two variables together where one depends upon the other.

Question 8:

The _____ plot with the normalized values on the y-axis gives us the probability density or mass function.

- A. Line
- B. Bar
- C. Histogram
- D. All of the above.

6

Statistical Inference

6.1 Basics of Statistical Inference and How It Works?

In most real-world situations, we have sample data, but we do not have access to the whole population we are interested in. For instance, we want to calculate the average marks of all the students of a city who appeared for a particular exam. Usually, it is not easy to collect the marks of all the students. In this case, we measure / take the sample of a small group of students, which represents the whole population of the students of the city who appeared in that particular exam.

Descriptive statistics provides us the information about the sample data. For instance, we sample marks of 100 students in a particular exam. We calculate the properties of the sample, such as the mean and the standard deviation, to get valuable information about these 100 students. These properties are called **statistics**. Chapter 4 deals with calculating statistics from sample data.

The properties of a population are called the **parameters** of the population. **Statistical inference** (or inferential statistics) comprises of the methods and the techniques which permit

us to use the information from the samples of a population to make decisions about the populations (generalizations). Thus, it is important to have a sample that represents the population with fairly large accuracy.

Techniques of statistical inference mostly deal with the:

- (1) parameter(s) estimation (learning) and
- (2) hypothesis testing.

In parameter estimation, we assume that the sample data comes from a statistical model or a particular probability distribution. The sample is analyzed to estimate the parameters of the assumed model or the probability distribution. For example, if we assume a Normal distribution, we have to estimate the mean and the standard deviation of the Normal distribution.

In hypothesis testing, we have to check certain assumptions about the population under consideration.

For instance, in medical research, based upon the experimental data and evidence, the researchers may have to decide whether there is any impact of a particular food item or a drug on the disease. Thus, it is hypothesized that food or drug has a significant impact on the disease. The hypothesis or the assumption may or may not be true. It has to be tested for truth. Hypothesis testing is used to accept or reject these assumptions.

6.2 Statistical Models and Learning

The methods of statistical inference make reasonable assumptions about the available data. Statistical models are a set of assumptions concerning the generation of the observed data.

For example, a company spends money on advertising its products to increase sales at different stores. The relationship between dollars spent on advertisement and sales in dollars for this company is given in Figure 6.1. Therefore, a model for the aforementioned problem can be a linear relationship or a line between advertisement and sales. A good model, which makes accurate assumptions about the data, is necessary for statistical inference.

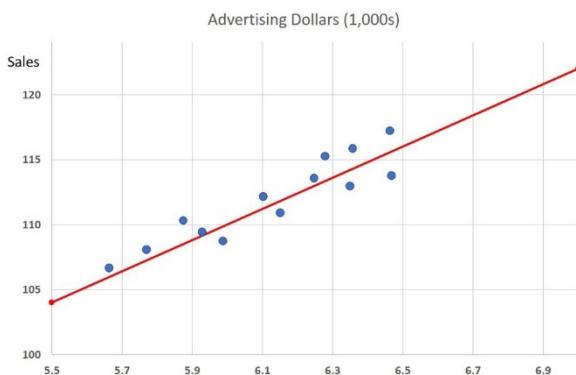


Figure 6.1: Relationship between dollars spent on advertisement and sales in dollars. Every point on the graph shows the data from one store. The line captures the overall trend of the data.

A statistical model is built from the observed data, sometimes referred to as the **training data**. Once a model is created, it is used to make inferences or predictions on the future data (or the **test data**). The process of inferring the parameters of the model, for example, slope and intercept of a simple linear model, is sometimes referred to as **learning** (or learning the parameters of a model).

Regression, a statistical method, finds the relationship between input and output variables so that this relation can be used to predict the most probable value of the output variable for a given input value. The regression assumes that

the output variable is continuous, contrary to the **classification** that works with discrete output variables.

The objective of a linear regression model is to find a relationship between one or more independent input features and a continuous target variable that is dependent upon the input features. When there is only one feature, it is called **univariate or simple linear regression**.

Ordinary Least Squares (OLS) is one of many mathematical techniques to find the solution of the mathematical equations resulting from the regression model. The result of the solution is the unknown parameters of the regression model.

Suppose we have only one input variable, x , such as the dollar spend on the advertisement, as in Figure 6.1. We assume the linear regression model. Then, this model can be mathematically described by the equation of a line.

$$f(x) = mx + c$$

The parameter m is the slope of the line, and the parameter c is the intercept on the vertical axis. $f(x)$ is the function to be estimated that approximates the output variable, such as the sales in dollars. The parameters m and c of the linear model have to be learned to find the optimal line that passes through the data points such that the error between the line and the data points is minimum.

Consider the case when we have multiple input features $x_1, x_2, x_3 \dots$. These features can be different mediums of the advertisement: TV, news, social media, etc., for the example given in Figure 6.1. Suppose all of the input features have a linear relationship with the target variable y . A linear model can still be assumed as follows:

$$f(x) = m_1x_1 + m_2x_2 + m_3x_3 + \dots + c.$$

In this case, the parameters to be learned are c , m_1 , m_2 , m_3 , and so on. In this case, there are multiple features; thus, it is a multivariate parameter learning problem. In this case, the regression is known as **multiple linear regression**.

6.3 Fundamentals Concepts in Inference

6.3.1 Point Estimation

The process of point estimation makes use of the observed data to calculate a point or a single value related to an unknown population. The point is generally a parameter, for example, the mean or the standard deviation of the population.

In the following example, we estimate the population mean from the sample mean.

```

1. import numpy as np
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. import scipy.stats as stats
5.
6. np.random.seed(12345)
7. population = stats.poisson.rvs(loc=12, mu=24, size=10000)
8. print('The population mean is', population.mean())
9. pd.Series(population).hist(density=True, z-order = 1,
   alpha=1, label = 'Population Histogram')
10.
11. # We estimate mean by taking a sample out of original
    population
12. np.random.seed(12345)
13. sample = np.random.choice(a= population, size=500) # 
    Sample 500 values
14. print ('The sample mean is', sample.mean())

```

```

15. pd.Series(sample).hist(density=True, z-order = 2,
   alpha=0.5, label = 'Sample Histogram')
16. plt.legend()
17.
18. # We check the difference between two means
19. print("Difference of the means is ",population.mean() -
   sample.mean() ) # difference between means

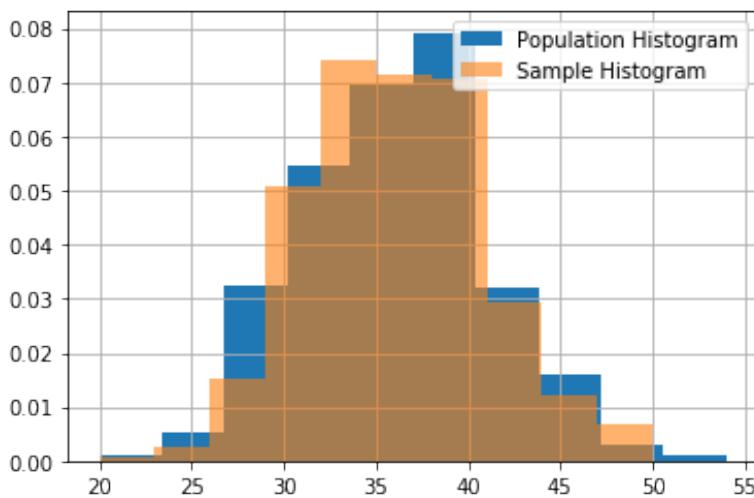
```

Output:

The population mean is 35.9784

The sample mean is 35.544

The difference of the means is 0.43440000000000367



Lines 6 and 12 of the code use a seed value of 12,345 to ensure similar results every time we run the code. We generate a Poisson random variable of 10,000 samples having an expected (average) rate of occurrence of 24, and the starting point is 12, as specified in line 7 of the code.

In line 13 of the code, we take 500 samples from the generated Poisson distribution using `random.choice()` function of the NumPy library. It can be observed from the output that the sample mean provides a very good approximation of the population mean because the difference between means is

very small. If our sample size is reduced, we may not get a good estimate of the population mean.

In a similar fashion, the point estimate for the variance or standard deviation of the population can be calculated from the sample variance or sample standard deviation.

6.3.2 Interval Estimation

Interval estimation, in contrast to point estimation, aims to find a range of values of interest of an unknown population.

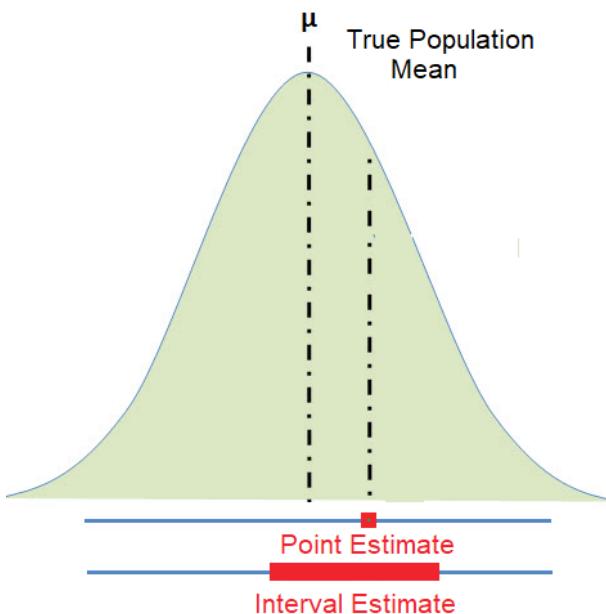


Figure 6.2: The point and interval estimates of a Normally distributed population whose true mean, μ , is given in the center of the distribution.

The accuracy of the estimation generally increases with large samples. However, it is very rare that the point estimate from a given sample exactly corresponds to the true population mean. Thus, in many cases, it is preferred to calculate an interval within which we expect our parameter of interest to

fall. The estimation of such an interval is called an interval estimate.

For instance, a random sample of scores for students in an exam may give us an interval from 65 to 75, within which we expect the true average of all scores. The endpoint values 65 and 75 depend on the samples and the computed sample mean. As the sample size increases and we take numerous samples, the variability between computed means for different samples decreases. Thus, the estimated mean, a parameter of the population μ , is likely to be closer to the true mean of the population of all students who appeared in that particular exam.

Let θ be the parameter to be estimated. It can be mean, standard deviation, or any other point. Let θ_L and θ_U be the lower and upper limits of the interval. The difference $\theta_U - \theta_L$ gives us the range within which we expect the true parameter of the population with a certain probability. The interval estimation techniques try to find this difference $\theta_U - \theta_L$. Interval estimates are either:

- confidence intervals (frequentist inference) or
- credible intervals (Bayesian inference).

We discuss these in chapters 7 and 8.

6.4 Hypothesis Testing

Sometimes, statisticians have to check certain assumptions, also known as **statistical hypotheses**, about the population under consideration. For instance, while analyzing data samples from patients, a researcher may have to study the origin of a certain disease.

Based on the experimental data and evidence, the researcher may have to decide, for example, whether COVID19 jumped to humans via pangolin. In this case, it is hypothesized that COVID19 is spread in humans via pangolin. This hypothesis or the assumption may or may not be true. Thus, it has to be tested for truth. The process of **hypothesis testing**, a method of statistical inference, is used by statisticians to accept or reject statistical hypotheses.

The data of the patients have to be collected to study the relationship between a certain food and the disease. For instance, a statistical hypothesis can be “*consuming too much caffeine increases the risk of cancer.*” Now, this hypothesis may or may not be true. Based on the analysis of data, we have to decide the truth of the said hypothesis.

To check if a hypothesis is true, we have to study the entire population. Since we do not usually have access to the data of the population, we analyze a random sample from the population. If the observed data is found inconsistent with the hypothesis, the hypothesis is rejected.

6.4.1 Null and Alternative Hypotheses

Hypothesis testing requires a statistical model to explain the data. In frequentist inference, the parameter to be estimated is assumed to be Normally distributed. Now, the observed data may be consistent or contradictory to the assumed data distribution. Based on the analysis of the data, we define two different statistical hypotheses:

1. **The null hypothesis**, denoted by H_0 , implies that the observed data is consistent with the assumed distribution. The null hypothesis usually describes that

the chance alone is responsible for the results. It means we have to check for the consistency of the data with our assumption. The null hypothesis is assumed true unless the observed data/evidence shows differently. If the observed data is consistent with the null hypothesis, then it is accepted.

2. **The alternative hypothesis**, denoted by H_1 , suggests that the observed data contradicts our assumptions. In this case, we reject the null hypothesis and accept the alternative hypothesis.

In a coin toss experiment, for example, we want to determine whether a coin is fair. A null hypothesis would be that both Heads and Tails would occur an equal number of times when this coin is flipped numerous times. The alternative hypothesis would be that the number of Heads and Tails would be different. We can express these hypotheses mathematically as:

$$\text{Null Hypothesis} = H_0: P = 0.5$$

$$\text{Alternative Hypothesis} = H_1: P \neq 0.5$$

Now, the coin is flipped 100 times, resulting in 70 Tails and 30 Heads. From this experimental data, we tend to reject the null hypothesis. Thus, based on the evidence from the observed data, we may decide that the coin is not fair.

Moreover, in the legal principle, *the presumption of innocence*, a suspect is assumed to be innocent until proven guilty. In this case, the null and the alternative hypotheses would be:

$$\text{Null Hypothesis} = H_0: \text{Suspect is innocent,}$$

$$\text{Alternative Hypothesis} = H_1: \text{Suspect is guilty.}$$

If the evidence is in favor of the suspect, they are declared innocent (null hypothesis is not rejected). On the other hand, the suspect is convicted based on the evidence against them (null is rejected, the alternative hypothesis is selected).

6.4.2 Procedure for Hypothesis Testing

In **hypothesis testing**, a procedure is followed to either reject or accept a null hypothesis.

1. Form the null and alternative hypotheses such that if one hypothesis is true, the other must be false.
2. Express the plan to use the observed data for the evaluation of the null hypothesis. This plan usually uses a single test statistic, such as the mean of the sample data.
3. Examine the observed data to compute the value of the test statistic, such as the mean.
4. Infer from the results whether to accept or reject the null hypothesis by applying the decision rules. An unlikely value of the test statistic from the sample data may result in the rejection of the null hypothesis.

6.5 Important Terms used in Hypothesis Testing

6.5.1 Sampling Distribution

Let us draw all possible samples of a specific size from a population of interest and compute a statistic such as the sample mean or the standard deviation for each drawn sample. Naturally, the computed statistic will vary from one sample to another. If we plot the probability distribution of this statistic,

we get a distribution called a **sampling distribution** of the statistic.

An important theorem in statistics is the **central limit theorem** that is concerned with the sampling distributions. Suppose we take sufficiently large random samples from a population of interest. Every time we have to take a sample, we replace the previous sample in the population. This is called sampling with replacement. The central limit theorem states that the resulting sampling distribution of the computed means of the individual samples will be approximately Normally distributed even though the original population may or may not be Normally distributed.

6.5.2 Errors in Hypothesis Testing

Performing a hypothesis test may result in either of two types of errors:

1. **Type I error** typically occurs when we reject a null hypothesis despite it being true. The probability of getting this error is known as the **significance level**, denoted by Alpha α . The value of α is set by the statistician according to the problem at hand before even investigating the data. Usually, it is set to 0.005, 0.01, or 0.05.
2. **Type II error** arises when we fail to reject a null hypothesis when it is false. The probability of picking up a Type II error is denoted by Beta β . The probability of not committing a Type II error or correctly rejecting the null hypothesis is called the **power of the test**.

Table 6.1: Summary of hypothesis testing.

Decision	Null Hypothesis H_0 is True	Null Hypothesis H_0 is False
Accept H_0	(1– α) Confidence level	β (Type II error)
Reject H_0	α (Type I error)	(1– β) Power of the test

6.5.3 Tests for Statistical Hypotheses

Two types of tests are common in statistical hypothesis testing. These are given as follows:

The **One-tailed test** has a rejection region on only one side of the sampling distribution. For instance, if the null hypothesis is that the mean is greater than or equal to 2, the alternative hypothesis is that the mean is lesser than 2. The rejection region is the range of numbers on the left side of 2, i.e., less than 2, in the sampling distribution.

The **Two-tailed test** has a rejection region on both sides of the sampling distribution. For instance, the null hypothesis is that the mean equals 2. The alternative hypothesis is that the mean is either less than or greater than 2. The rejection region is the range of numbers on both sides of the sampling distribution.

If we have sufficiently large samples available, we assume a Normal sampling distribution. In this case, we perform testing using z-scores. However, when we do not have access to large sample sizes, say more than 30 samples, we resort to t-distribution for testing purposes.

The t-distribution, also known as the Student's t-distribution, is similar to the Normal distribution but with tails heavier than the Normal distribution. It implies that a t-distribution has a

greater chance of observing extreme values than a Normal distribution because it has heavier tails. When our sample size is small and the variance of the population is unknown, we perform t-tests instead of z-tests for hypothesis testing.

6.5.4 z-value (z-score)

z-value, also known as z-score, is a measure of standard deviations. It measures how many standard deviations the observed value is away from the mean in a Normal distribution. For example, z-value = 2 means that the observed value is 2 standard deviations away from the mean. Below is the formula to calculate the z value:

$$z = \frac{(x - \mu)}{\sigma}$$

where x , μ , and σ are data points on the curve, mean, and the standard deviation, respectively. The z-score represents the area under the Normal distribution up to the value of z . Recall from Section 3.6.3 on the Normal distribution that the area under the curve from $-\sigma$ to $+\sigma$ is 68.3 percent of the overall area. Furthermore, the area under the curve from -2σ to $+2\sigma$ is 95.5 percent of the overall area.

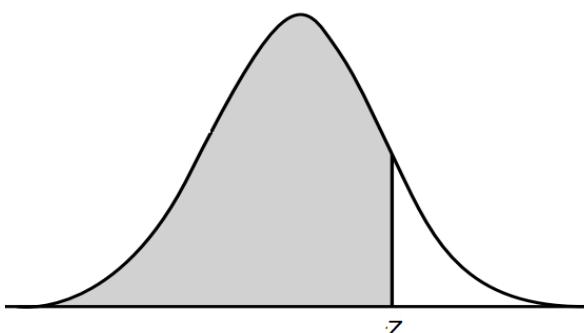


Figure 6.3: z-score from a Normal distribution.

The value of z is first computed by subtracting the mean μ of the distribution from the data point x . This operation centers the distribution around 0. Second, we divide the difference between x and μ by the standard deviation σ to make a standard Normal distribution having a unit standard deviation. The z -score is then calculated by computing the area under the Normal curve up to the value of z .

A z -score equal to, less than, and greater than 0 represents an element equal to, less than, and greater than the mean, respectively. A z -score equal to $-2 / 2$ represents a data point that is 2 standard deviations less than / greater than the mean, respectively.

When the number of sample data points is large, about 68 percent of the elements have a z -score between -1 and 1 ; about 95 percent have a z -score between -2 and 2 ; and about 99 percent have a z -score between -3 and 3 .

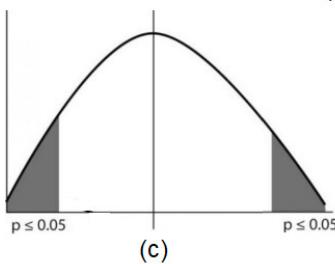
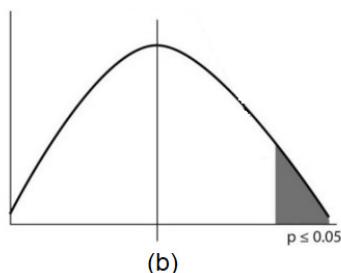
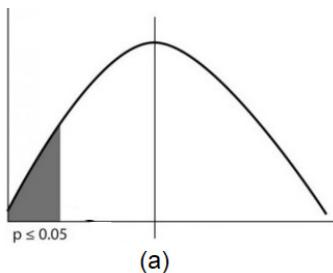
6.5.5 p-value

The examination of the observed data makes use of decision rules for rejecting or accepting the null hypothesis. The decision rules are described by using a probability value known as the p-value.

To understand the p-value, we first describe some important terminology.

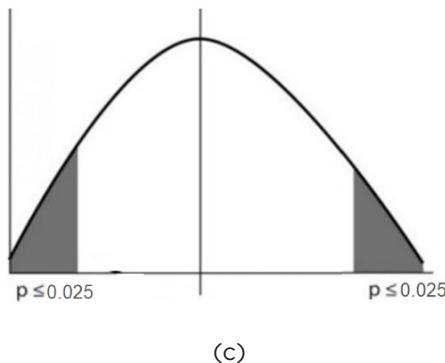
1. **The region of acceptance** describes a range of values. If the test statistic falls within this range, the null hypothesis is not rejected. The region of acceptance is chosen in such a way that the probability of making a Type I error is equal to the significance level.

- 2. The critical region** or the **region of rejection** comprises the set of values outside the region of acceptance. If the test statistic falls within the region of rejection, the null hypothesis is rejected at the α level of significance.
- 3. The significance level** is the probability of rejecting the null hypothesis when it is true. The significance level α is used in hypothesis testing to find out the hypothesis that the data support. For example, a significance level of 0.05 indicates a 5 percent probability that an actual difference between the sample and the population represented by our null hypothesis exists when there is no actual difference. A lower significance level means stronger evidence will be required to reject the null hypothesis.



(a)

(b)



(c)

Figure 6.4: For a 95% confidence interval, regions of acceptance (unshaded) and rejection (shaded) for (a) one-tailed (left-tailed), (b) one-tailed (right-tailed), and (c) a two-tailed test. If the p-value falls in shaded regions, it is considered significant to reject the null hypothesis.

The **p-value** is the probability of obtaining test results at least as extreme as the results actually observed, assuming a correct null hypothesis. It measures the strength of evidence in support of a null hypothesis. If the p-value is less than the significance level, we reject the null hypothesis.

A very small p-value implies that under the null hypothesis such an extreme observed outcome would not be likely. A p-value is compared against the significance level, α . If the p-value is less than α , we have a very unlikely observation. Thus, we say that the results are statistically significant because the sample data is giving us this evidence.

In this case, we reject the null hypothesis because there is very little chance of observing the data consistent with the sampling distribution of the test statistic under the null hypothesis.

In the tails of the Normal distribution, we get a high magnitude of z-values that correspond to small p-values. Getting these values is an indication that it is very unlikely that the

observed sample comes from a pattern represented by our null hypothesis, i.e., we reject the null hypothesis.

As an example, suppose the population mean is 10, and the range of means is from 9.5 to 10.5. If we find the sample average is outside the given range, the null hypothesis is rejected. Else, the difference is supposed to be justifiable by chance alone.

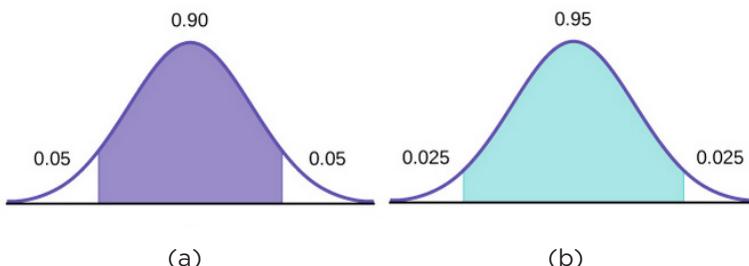


Figure 6.5: (a) A confidence level of 90% corresponding to $\alpha = 1 - 0.90 = 0.1$, $\alpha/2 = 0.1/2 = 0.05$ area of rejection is on either side, (b) A confidence level of 95%, corresponding to $\alpha = 1 - 0.95 = 0.05$, $\alpha/2 = 0.05/2 = 0.025$ area of rejection is on either side.

Let us use a 95% confidence level, the associated significance level α would be 0.05 (5%). The z-score corresponding to 0.025 (2.5%), 0.05 (5%), 0.95 (95%) and 0.975 (97.5%) would be -1.96, -1.64, 1.64, and +1.96, respectively.

Table 6.1: z-scores for a confidence level of 95% (significance level α of 5%).

Test type	Range of z-scores in the region of acceptance
Left-tailed	$-\infty$ (infinity) to +1.64
Right-tailed	-1.64 to ∞
Two-tailed	-1.96 to +1.96

For a 95 percent confidence interval in a two-tailed test, if a z-score is between -1.96 and +1.96, the p-value will be larger than 0.025. We do not reject the null hypothesis.

When the z-score is outside the $-1.96 - 1.96$ range, e.g., 2.8, the corresponding p-value will be smaller than 0.025. In this instance, the null hypothesis can be rejected straightaway.

Note that both z-value and p-value are associated with the standard Normal distribution. These methods do not work with other distributions. Thus, we assume our sampling distribution to be a Normal distribution due to the central limit theorem.

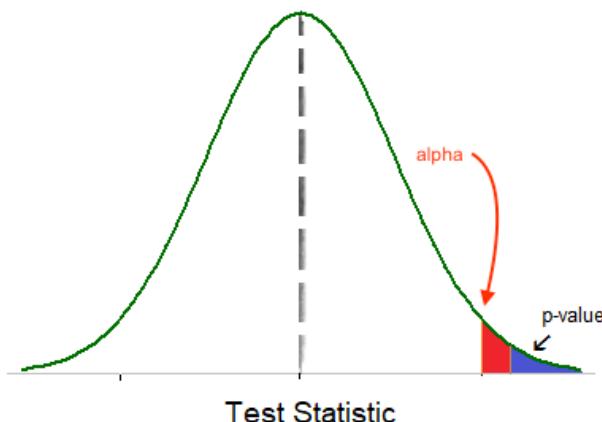
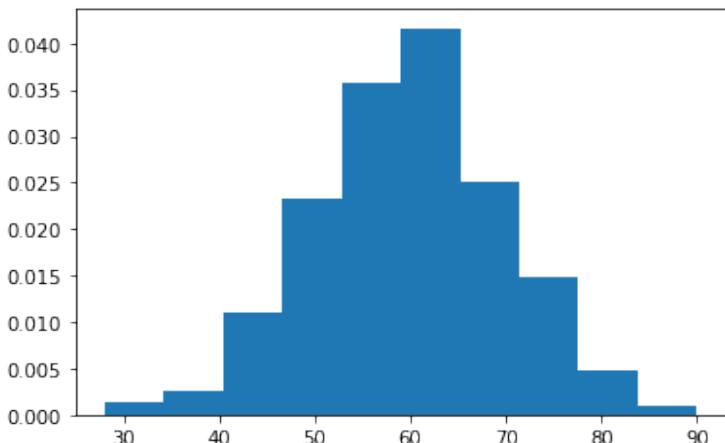


Figure 6.4: Comparison of p-value and α for hypothesis testing in the sampling distribution of the test statistic such as the mean. The area under the curve outside the observed data point is the p-value.

Suppose we have the scores of 500 students with an average score of 60. We would like to know the probability of getting a score, such as 85. Let us plot the student score as a histogram.

1. `np.random.seed(123)`
2. `student_score = np.random.normal(60, 10, 500).round()`
3. `plt.hist(student_score, bins = 10, density=True)`
4. `plt.show()`
5. `print('The mean student score is', student_score.mean())`
6. `print('The mean student score is', student_score.mean())`
- 7.

Output:

The mean student score is 59.626

Line 1 of the code uses a random seed to reproduce the results. Line 2 generates 500 Normally distributed random numbers rounded to an integer by the use of the round () function. The mean value of these numbers is set to 60, whereas a standard deviation of 10 is used.

The output shows an average student score of 59.626, and the histogram shows an approximate Normal distribution of the scores. The score of students can be normalized by using `scipy.stats` function `z-score ()` to get a standard Normal distribution with zero mean and unit standard deviation. Here, we use the formula:

$$z = \frac{(x - \mu)}{\sigma}$$

to get the z-value corresponding to a particular value. For example, we have been given a class of students whose average marks are 85. We want to find out the probability of getting 85 marks given we have another class of 500 students

whose mean marks are found to be 60. First, we compute the z-value corresponding to 85 marks.

```
1. z-score = ( 85 - student_score.mean() ) / student_score.std()
2. z-score
Output:
2.5215872335107994
```

Since the area under the probability distribution is 1, a probability of 0.95, let us say, corresponds to 95 percent of the area under the distribution. Thus, in a quantile function, also known as inverse cumulative distribution function or percent point function, we specify the value of the probability to get the percentage of points.

In a standard Normal distribution, 95 percent of points correspond to a z-value of 1.645 by using the flowing command. The Python code to find this z-value from the probability or percentage of points is given as follows:

```
1. stats.norm.ppf(0.95)
Output:
1.6448536269514722
```

If we have to find out, for instance, marks above which top 5 percent students lie, we type the following code:

```
1. stats.norm.ppf(0.95)*student_score.std()+student_score.mean()
Output:
76.17768434215024
```

This code converts a z-value to the corresponding x value in the Normal distribution using the following formula that comes directly from the formula of calculating the z-value.

$$x = z\sigma + \mu$$

The output of the code implies that if a student has marks above 76.178, they lie in the top 5 percent marks. We use the following Python script to find the p-value from the z-score corresponding to 85 mean marks of our second class.

```
1. marks_to_check = 85
2. z-score = ( marks_to_check - student_score.mean() ) /
   student_score.std()      # getting z-score
3. p_val = 1 - stats.norm.cdf(z-score)           # getting the
   p-value
4. print(f'The z-score = {z-score:0.3f} and the p-value = {p_
   val:0.3f}')
```

Output:

```
The z-score = 2.522 and the p-value = 0.006
```

Line 2 converts the x value to the corresponding z-score, whereas line 3 computes the p-value. `stats.norm.cdf(z-score)` calculates the area under the Normal curve to the left of the z-score. We subtract this area from 1 to get the area under the curve to the right of the z-score. Line 4 uses the f-string format, and `0.3f` displays the floating-point numbers to 3 decimal values.

A positive z-score of 2.522 means that 85 marks are about 2.5 standard deviations away from the mean. Furthermore, a p-value of 0.006 implies that there is about a 0.6 percent chance of getting these marks from a sample of 500 students having an average of 60 marks and a standard deviation of 10 marks.

Now, if we set a significance level of 5 percent, we can compare it against the computed p-value of 0.6 percent. If we hypothesize that getting 85 marks is common, we can reject our hypothesis because 0.6 percent is less than a 5 percent significance level.

We make use of z-values and p-values in Chapter 7 for hypothesis testing.

Further Readings

More information on the point and interval estimates can be found at:

<https://bit.ly/365cSNu>

For more information about hypothesis testing, visit:

<https://bit.ly/38bWyNm>

6.6 Exercise Questions

Question 1:

Which of the following testing is used to make decisions from the observed data?

- A. Probability
- B. Hypothesis
- C. Statistics
- D. None of the mentioned

Question 2:

Which of the values is mostly used to measure statistical significance?

- A. p
- B. μ
- C. σ
- D. All of the mentioned

Question 3:

The use of samples to estimate population parameters is called:

- A. Statistical interference
- B. Statistical inference
- C. Statistical application
- D. None of the mentioned

Question 4:

A large positive z-score, for example, 3, implies that the value under consideration is _____, and the corresponding p-value would be _____.

- A. Highly likely, small
- B. Highly likely, large
- C. Less likely, small
- D. Less likely, large

Question 5:

A small positive z-score, for example, 0.5, implies that the value under consideration is _____, and the corresponding p-value would be _____.

- A. Highly likely, small
- B. Highly likely, large
- C. Less likely, small
- D. Less likely, large

Question 6:

In a one-tailed test, the confidence level is set to 90 percent. If the p-value is lesser than _____, we reject the null hypothesis.

- A. 0.01
- B. 0.5
- C. 0.05
- D. 0.1

Question 7:

In a two-tailed test, the confidence level is set to 90 percent. If the p-value is lesser than _____, we reject the null hypothesis.

- A. 0.05
- B. 0.5
- C. 0.025
- D. 0.25

7

Frequentist Inference

It is pointed out in Chapter 6 that most techniques of statistical inference are either:

- (1) parameter(s) estimation (learning) or
- (2) hypothesis testing.

In parameter estimation, we have to estimate the parameters of a statistical model or a distribution. In hypothesis testing, we have to check certain assumptions about the population under consideration.

These assumptions have to be tested for truth. Methods of hypothesis testing are used to accept or reject these assumptions. In the following sections, we present parametric and non-parametric estimation methods as well as methods for hypothesis testing.

7.1 Parametric Inference

Parametric models belong to a class of statistical models through which we can represent the population using a **finite set** of fixed parameters. A parametric model can also be defined in terms of probability distributions, which can be represented by a finite number of parameters.

A linear model can be described by the parameters of the line: slope and intercept. To estimate a target variable corresponding to the unseen future test data, we need just the parameters of the model.

Furthermore, if a population is assumed to be Normally distributed, it can be defined in terms of two parameters of the Normal distribution: mean and standard deviation. Thus, the model that assumes the Normal distribution is said to be a parametric model. The examples of learning algorithms that utilize parametric models include:

- Linear **regression**,
- Logistic regression (**classification**), and
- Naïve Bayes' classification.

Linear regression uses input variables to estimate the output target variable when it is continuous, whereas logistic regression tries to estimate the discrete output variables. The problems where we estimate discrete output labels or classes are called classification problems.

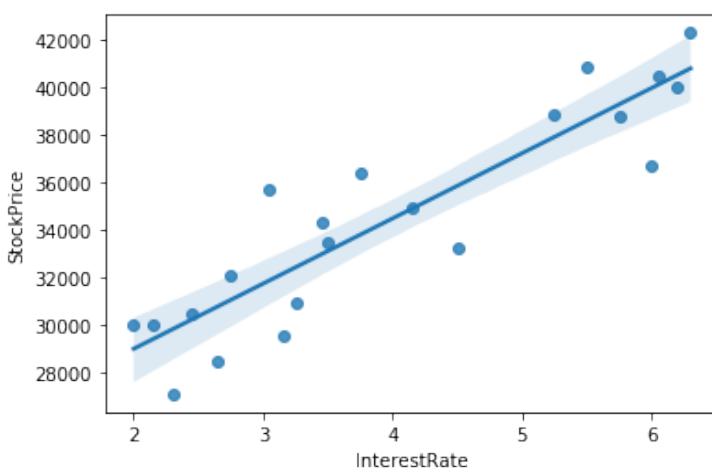
To implement linear regression, a parametric model, in Python, we use the following program:

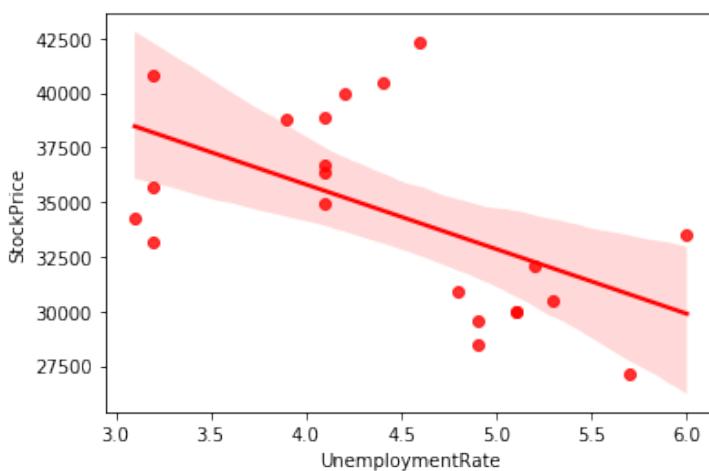
```
1. from pandas import DataFrame
2. import statsmodels.api as sm
3. import seaborn as sns
4. import matplotlib.pyplot as plt
5.
6. StockMarketData = {
7.     'Year': [2001,2002,2003,2004,2005,2006,2007,2008,2009,
8.             2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020,2021],
```

```
8.     'InterestRate': [2, 2.15, 2.45, 2.75, 2.3, 2.65, 3.5,
    3.15, 3.25, 4.15, 4.5, 3.45, 3.75, 5.25, 5.75, 5.5, 3.05, 6, 6.2, 6.05
    , 6.3],
9.     'UnemploymentRate': [5.1, 5.1, 5.3, 5.2, 5.7, 4.9,
    6, 4.9, 4.8, 4.1, 3.2, 3.1, 4.1, 4.1, 3.9, 3.2, 3.2,
    4.1, 4.2, 4.4, 4.6],
10.    'StockPrice': [30000, 30010, 30500, 32104, 27098, 28459, 33
    512, 29565, 30931, 34958, 33211, 34293, 36384, 38866, 38776, 40822,
    35704, 36719, 40000, 40500, 42300]
11. }
12.
13. df = DataFrame(StockMarketData,columns=[ 'Year',
    'InterestRate', 'UnemploymentRate', 'StockPrice'])
14. print(df)
15.
16. # here we have 2 variables for the multiple linear
   regression.
17. #If you just want to use one variable for simple linear
   regression,
18. #then use X = df[ 'Interest_Rate' ] or X =
   df[ 'UnemploymentRate' ]
19. X = df[[ 'InterestRate', 'UnemploymentRate' ] ]
20. Y = df[ 'StockPrice' ]
21.
22. sns.regplot(x=X[ 'InterestRate' ],y=Y)
23. plt.show()
24. sns.regplot(x=X[ 'UnemploymentRate' ],y=Y,color='r')
25. plt.show()
26.
27. X = sm.add_constant(X) # adding a constant for y-intercept
28.
29. model = sm.OLS(Y, X).fit()
30.
31. predictions = model.predict(X)
32. print("Predictions:\n",predictions)
33. print_model = model.summary()
34. print(print_model)
```

Output:

	Year	InterestRate	UnemploymentRate	StockPrice
0	2001	2.00	5.1	30000
1	2002	2.15	5.1	30010
2	2003	2.45	5.3	30500
3	2004	2.75	5.2	32104
4	2005	2.30	5.7	27098
5	2006	2.65	4.9	28459
6	2007	3.50	6.0	33512
7	2008	3.15	4.9	29565
8	2009	3.25	4.8	30931
9	2010	4.15	4.1	34958
10	2011	4.50	3.2	33211
11	2012	3.45	3.1	34293
12	2013	3.75	4.1	36384
13	2014	5.25	4.1	38866
14	2015	5.75	3.9	38776
15	2016	5.50	3.2	40822
16	2017	3.05	3.2	35704
17	2018	6.00	4.1	36719
18	2019	6.20	4.2	40000
19	2020	6.05	4.4	40500
20	2021	6.30	4.6	42300



**Predictions:**

```
0    28900.344332
1    29278.508818
2    29868.786739
3    30708.141239
4    29158.520145
5    30705.108162
6    31934.759459
7    31965.656451
8    32300.791636
9    35150.957245
10   36780.570789
11   34216.444907
12   34142.518613
13   37924.163482
14   39350.762825
15   39301.667368
16   33124.980749
17   39814.985916
18   40236.179705
19   39691.964165
20   40156.187256
dtype: float64
```

OLS Regression Results						
=====						
Dep. Variable:	StockPrice	R-squared:	0.837			
Model:	OLS	Adj. R-squared:	0.819			
Method:	Least Squares	F-statistic:	46.28			
Date:	Tue, 29 Sep 2020	Prob (F-statistic):	8.04e-08			
Time:	12:28:34	Log-Likelihood:	-186.95			
No. Observations:	21	AIC:	379.9			
Df Residuals:	18	BIC:	383.0			
Df Model:	2					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.975]

const	2.809e+04	3439.149	8.168	0.000	2.09e+04	3.53e+04
InterestRate	2521.0966	327.644	7.695	0.000	1832.742	3209.451
UnemploymentRate	-830.2553	583.165	-1.424	0.172	-2055.439	394.929
=====						
Omnibus:	2.076	Durbin-Watson:			1.924	
Prob(Omnibus):	0.354	Jarque-Bera (JB):			1.594	
Skew:	-0.509	Prob(JB):			0.451	
Kurtosis:	2.114	Cond. No.			51.0	
=====						
Warnings:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.						

In this program, we want to predict the Stock Price based upon two independent variables—Interest Rate and Unemployment Rate. First, we import Pandas, Statsmodels.api, Seaborn, and Matplotlib libraries. We save our data in a dictionary in lines 6 to 11. Next, we construct a Pandas DataFrame from the saved dictionary in line 13.

We display our complete data using line 14 of the code that prints the DataFrame df. There are four columns in the output: Year, Interest Rate, Unemployment Rate, and Stock Price.

The X and the Y variables for the linear regression are constructed in lines 19 and 20, respectively. Note that the variable X contains data for both independent variables—Interest Rate and Unemployment Rate. Since we use multiple (2) input variables to estimate the output variable Stock Price, this becomes a **multiple linear regression** problem.

Lines 22 to 25 plot the data as regression plots of the Seaborn library to find the general trend of the dependent variable against independent variables. The output plots are used to find the general trend of the data visually. For example, the trend of the first plot shows a positive relationship between Interest Rate and Stock Prince, whereas the second plot shows a negative relationship between Unemployment Rate and Stock Price.

Line 27 is used to add a constant column to the variable X. This constant is required by the statsmodel.api module to find the y-intercept term. Finally, line 29 fits the data using Ordinary Least Squares (OLS), a mathematical algorithm, to find the parameters (coefficients) of the multiple linear regression model.

The ordinary least squares (OLS) regression results show a lot of statistics and useful information about the learned model. The parameters of the model are given as the coef:

const = 2.809e+04,

Interest Rate = 2521.0966 and

Unemployment Rate = -830.2553.

Note that a positive parameter (slope) for the Interest Rate indicates that the dependent/target variable Stock Price has a positive relationship with this independent variable,

whereas the negative value—830.2553—indicates that the Unemployment Rate is inversely proportional to the Stock Price. This is also evident from the plots generated by the program.

Parametric models are simpler and easier to interpret. Due to the limited number of fixed parameters, they are usually faster than nonparametric models. A less amount of data is required to learn the parameters of these models. On the downside, they are generally applicable to the problems in which the structure of the underlying data is not very complex.

7.2 Confidence Intervals

Interval estimates are formed by specifying a value. A value of 0.95 indicates that if the experiment is run, sampling distributions of the sample statistic is drawn, then 95 percent of the time, these distributions would capture the true parameter within the interval of interest.

Since this value gives us a kind of surety about the true parameter, we call this value a **confidence level**. Since we deal with the confidence level, we refer to an interval of interest as a **confidence interval**. Preferably, we select a short interval with a high confidence level or degree of confidence. It is better to be 95 percent confident that certain electronic equipment has an average life of 4 to 5 years than to be 99 percent confident that its average life is 1 to 10 years.

Since we are assuming the repetition of the sampling process and drawing the sampling distributions accordingly, we are interpreting the probability as done in the **frequentist statistics**.

The Python implementation for the confidence interval is given as:

```
1. import numpy as np
2. import scipy.stats as stats
3.
4. N=1000      # sample size
5. gamma=0.95  # gamma = 1-alpha (also known as confidence
   level)
6. alpha = 1 - gamma
7.
8. # mean and standard deviation of the population
9. mu=5
10. sigma=3
11.
12. # Normally distributed sample data of size N
13. x=np.random.randn(N)*sigma+mu
14.
15. # Calculation of mean and standard deviation of the sample
   data
16. mu_sample=np.mean(x)
17. sigma_sample=np.std(x, ddof=1)    # delta degree of freedom
   ddof=1 means divide by N-1 instead of N for the sample
   STD.
18.
19. print('sample mean: mu_sample
   : %f' % mu_sample)
20. print('sample standard deviation: sigma_sample
   : %f' % sigma_sample)
21.
22.
23. # calculations using normal distribution for 95%
   confidence level
24. l1=stats.norm.ppf(0.025)      # lower percentile
25. u1=stats.norm.ppf(0.975)    # upper percentile
26. print('confidence interval of mu_sample using normal
   distribution : (%f, %f)' %
27.       (mu_sample+l1*sigma_sample/np.sqrt(N), mu_
   sample+u1*sigma_sample/np.sqrt(N)))
```

Output:

```
sample mean: mu_sample : 3.940637
sample standard deviation: sigma_sample : 2.758312
confidence interval of mu_sample using normal distribution
: (2.859398, 5.021875)
```

We specify 1,000 sample points and a confidence level of 0.95. The data is sampled in line 13 of the code. This data comes from a normal distribution having mean 5 and standard deviation 3, as specified in lines 9 and 10. The sample means and standard deviations are printed using lines 19 and 20 of the code. Since the percent point function or the quantile function is used to get the values of the random variables when the probability is given as the input to the function.

Thus, we use `stats.norm.ppf` and specify lower point $\alpha/2 = (1-0.95)/2 = 0.05/2 = 0.025$ or 2.5% and upper point $1-\alpha/2 = 1 - (1-0.95)/2 = 1 - 0.05/2 = 1 - 0.025 = 0.975$ or 97.5% on the normal curve to this function. In return, we get -1.96 and 1.96 as z-scores corresponding to 2.5% and 97.5%, respectively. This is done in lines 24 and 25 of the code. Finally, we use the formula in line 27 of the code to print the confidence interval.

$$\text{Confidence Interval} = \left(\bar{\mu} - z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}} \right) \text{ to } \left(\bar{\mu} + z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}} \right)$$

where $\bar{\mu}$ represents the point estimate of the mean, $z_{\alpha/2}$ is the z-score at the junction between the region of acceptance and rejection, and σ/\sqrt{n} is the standard deviation of the sampling distribution of the mean, which is also referred to as standard error (SE).

7.3 Nonparametric Inference

In **nonparametric models**, the structure of the model is not defined in advance. Rather, it depends upon the data. As more and more data are gathered, the number of parameters of the model varies to fit the data accordingly. It is not the case that the nonparametric models do not have any parameters. The number of parameters is not fixed instead.

The examples of learning algorithms that utilize nonparametric models include:

- Histogram
- Kernel density estimation (KDE)
- Nonparametric regression and
- K-nearest neighbors (KNN).

Both histograms and the kernel density estimation (KDE) methods provide us the estimates of the probability density function (PDF) of a random variable. In nonparametric regression, the parameter size varies depending upon the available data. A KNN looks K points in the available data that are nearest to the test data point for estimation purposes. It can be used for both regression and classification problems.

Nonparametric methods such as KDE make inferences about the population from a finite data sample. These methods make use of one of many kernels that are non-negative symmetric mathematical functions. Examples of kernel functions include uniform, triangle, Gaussian, and cosine functions.

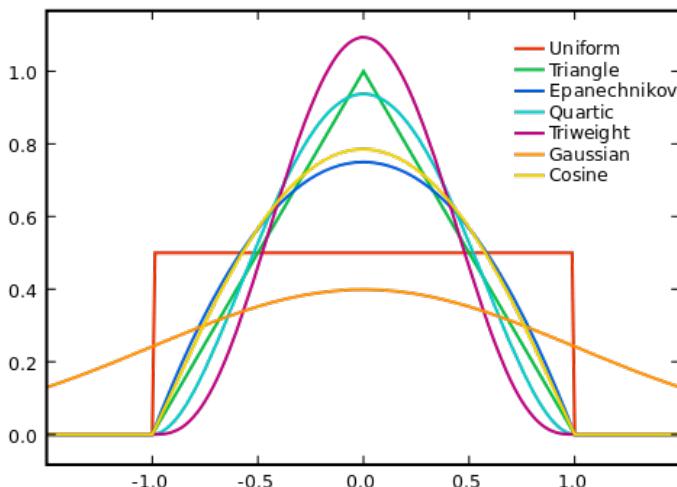


Figure 7.1: Examples of kernel functions.

A histogram counts the number of data points in different regions to estimate PDF, whereas KDE is a function defined as the sum of a kernel function on every data point. KDE gives a better estimate of the PDF as compared to a histogram.

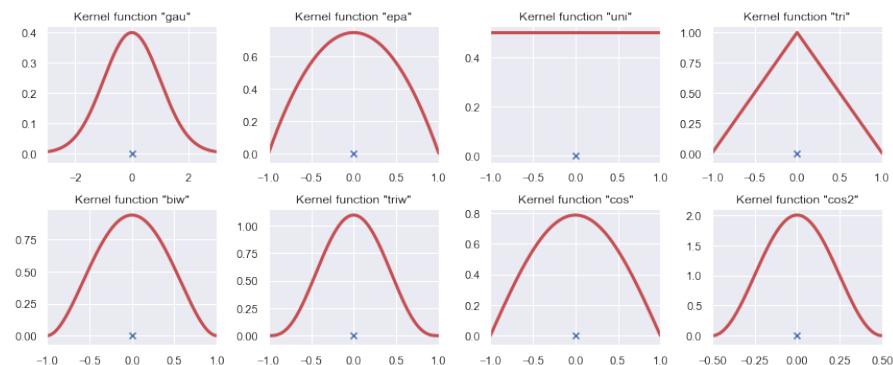


Figure 7.2: Kernel functions available in Statsmodels library.

To estimate the PDF using KDE for a univariate case, we use **sm.nonparametric.KDEUnivariate**.

To implement nonparametric kernel density estimation in Python, we first import the necessary packages.

```

1. %matplotlib inline
2. import numpy as np
3. from scipy import stats
4. import statsmodels.api as sm
5. import matplotlib.pyplot as plt
6. from statsmodels.distributions.mixture_rvs import mixture_
   rvs
7.
8. # Seed the random number generator for reproducible
   results
9. np.random.seed(12345)

```

We use the function **%matplotlib inline** that renders the figures such that they become part of the notebook.

When we mention seed() with a particular number as input, say 0 or 12345, we make sure to generate the same set of random numbers always. Thus, it is used to reproduce results.

We sample from known distributions as follows:

```

1. # Location (center), scale (spread) and weight for the two
   distributions
2. pdf1_loc, pdf1_scale, pdf1_weight = -1 , .5, .25
3. pdf2_loc, pdf2_scale, pdf2_weight = 1 , .5, .75
4.
5. # Sample from a mixture of distributions
6. sample_dist = mixture_rvs(prob=[pdf1_weight, pdf2_weight],
   size=250,
7. dist=[stats.norm, stats.norm],
8. kwargs = (dict(loc=pdf1_loc, scale=pdf1_scale),
   dict(loc=pdf2_loc, scale=pdf2_scale)))

```

First, we specify the centers, spread, and probability weight for the two distributions in lines 2 and 3. A weight of .25 indicates that we take 25% of the samples from the first PDF. To get samples from a mixture of distributions, we use the function **mixture_rvs** that take probability weights, size, and types of distributions as the input in lines 6 to 8 of the code.

We specify Normal distributions for both cases and take 250 samples as specified in line 6. The option kwargs (keyworded arguments), like the option args, allows us to pass a variable number of arguments to a function as input. Usually, we use a dictionary object with kwargs.

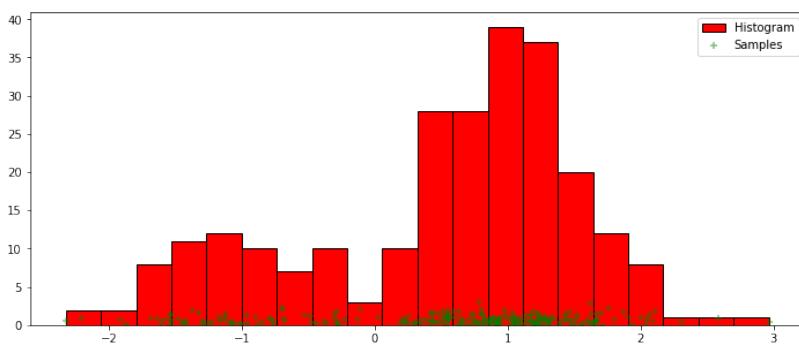
Next, we plot the samples and their histograms by the following piece of code:

```

1. fig = plt.figure(figsize=(12, 5))
2. ax = fig.add_subplot(111)
3.
4. # Scatter plot of data samples and histogram
5. ax.scatter(sample_dist, np.abs(np.random.randn(sample_
    dist.size)),
6.             z-order=2, color='g', marker ='+' , alpha=0.5,
    label='Samples')
7.
8. lines = ax.hist(sample_dist, bins=20, color = 'r',
    edgecolor='k', label='Histogram', z-order=1)
9.
10. ax.legend(loc='best')
11. plt.show()

```

Output:



The first line of the program adjusts the size of the figures, whereas the second line specifies how many subplots we need. We have specified 111 in line 2. The first two digits, 11, indicate

a matrix of subplots having one row and one column, whereas the last digit 1 means the subplot number 1. Lines 5 and 6 are used to plot the samples of distributions as green markers +, while the histogram is drawn by counting the number of samples corresponding to each bin in line 8. It can be seen that approximately 25 percent and 75 percent of the samples belong to the first and the second distribution, respectively.

```
1. kde = sm.nonparametric.KDEUnivariate(sample_dist)
2. kde.fit() # Estimate the densities
```

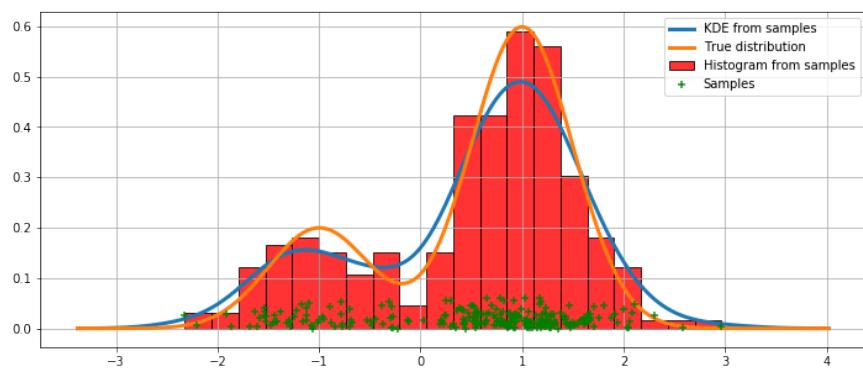
The abovementioned two lines of code are used to estimate the PDFs of a mixture of two distributions. Finally, we plot the results as follows:

```
1. fig = plt.figure(figsize=(12, 5))
2. ax = fig.add_subplot(111)
3.
4. # Plot the histogram of samples
5. ax.hist(sample_dist, bins=20, density=True,
   label='Histogram from samples',
   z-order=1, color='r', edgecolor='k', alpha=0.8)
6.
7.
8. # Plot the KDE as fitted using the default arguments
9. ax.plot(kde.support, kde.density, lw=3, label='KDE from
   samples', z-order=2)
10.
11.
12. # Plot the true distribution
13. true_values = (stats.norm.pdf(loc=pdf1_loc, scale=pdf1_
   scale, x=kde.support)*pdf1_weight
14.           + stats.norm.pdf(loc=pdf2_loc, scale=pdf2_
   scale, x=kde.support)*pdf2_weight)
15. ax.plot(kde.support, true_values, lw=3, label='True
   distribution', z-order=3)
16.
```

```

17. # Plot the samples
18. ax.scatter(sample_dist, np.abs(np.random.randn(sample_
    dist.size))/40,
19.             marker='+', color='g', z_order=4,
    label='Samples', alpha=0.9)
20.
21. ax.legend(loc='best')
22. ax.grid(True)
Output:

```



The output of this program shows the actual samples (green '+' markers), the histogram (red bars), estimated KDE from the samples (blue), and the true distribution (orange).

The option `density = True` normalizes the histogram in line 5 of the code.

The option `z-order` in all four plots defines the order of the appearance of the plots. For example, a plot having a greater number of `z-order` appears in front of the plot that has a smaller number of `z-order`.

The option `kde.support` provides the values of `x-axis` against which the estimated density `kde.density` is plotted in line 9 of the code.

To plot the true distribution that is a mixture of two normal distributions of weights 25 percent and 75 percent, we mix both true distributions together in lines 13 and 14 of the code. Note that there is a slight difference between the true distributions and the estimated densities from KDE. This is partly because we use a limited number of observations for kernel density estimation.

The benefits of nonparametric models include their flexibility to learn very complex forms of relationships between the input and the output data because they do not assume the form of function that maps input to the output.

The negative aspects of these models include the requirement of a lot of training data to estimate the mapping function. Furthermore, these models are slower to learn because of the presence of a lot of parameters.

Further Readings

For the official documentation of the stable release of the Statsmodel package and the kernel density estimation, visit the following webpage:

<https://bit.ly/32avvhC>

<https://bit.ly/3mLm4gy>

7.4 Hypothesis Testing using z Tests

As mentioned in Chapter 6, when we have a sample size greater than 30 observations, and the standard deviation of the population is known, we perform z tests for hypothesis testing. Otherwise, we perform t tests. In this section, we present one-tailed and two-tailed z tests.

7.4.1 One-tailed z Test

Suppose the hypotheses to be tested are as follows:

Null Hypothesis : $H_0: \text{sample_mean} < 19$

Alternate Hypothesis : $H_1: \text{sample_mean} > 19$

In this case, the alternative hypothesis is on only one side of the null hypothesis, i.e., the mean of the sample is greater than 19. The test that can be performed to test these hypotheses is called a one-tailed z test.

We assume a confidence interval of 95 percent. The corresponding z-value is computed. The p-value is compared against the significance level α , 0.05 (5%). If the z-value is found greater than the critical z-value that corresponds to the z-value at the boundary of the region of acceptance and the region of rejection, we accept the null hypothesis.

To implement the one-tailed z test, type the following Python script:

```
1. import numpy as np
2. from numpy import random
3. import math
4. import scipy.stats as st
5.
6.
7. population = np.array([20,12,22,32,52,1,22,30,40,50,6,12,
   32,52,1,22,3,7,12,32,52,62,12,32,52,11,22,3])
8. sample_size = 12
9. sample = random.choice(population, sample_size)
10.
11. population_mean = population.mean()
12. print("Population Mean: ", population_mean)
13.
14. sample_mean = sample.mean()
15. print("Sample Mean: ", sample_mean)
```

```
16.  
17. # Null Hypothesis  
18. # sample_mean < 19  
19. # Alternate Hypothesis  
20. # sample_mean > 19  
21.  
22. pop_stdev = population.std()  
23. print("Population Standard Deviation:", pop_stdev)  
24. z_test = (sample_mean - population_mean) / (pop_stdev/math.  
    sqrt(sample_size))  
25. print("Z test value: ", z_test)  
26.  
27. confidence_level = 0.95  
28. z_critical_val = st.norm.ppf(confidence_level)  
29. print("Z critical value: ", z_critical_val)  
30.  
31. if(z_test > z_critical_val):  
32.     print("Null Hypothesis is rejected.")  
33. else:  
34.     print("Null Hypothesis is accepted.")  
Output:  
Population Mean: 25.214285714285715  
Sample Mean: 24.083333333333332  
Population Standard Deviation: 18.020538169498916  
Z test value: -0.21740382738025502  
Z critical value: 1.6448536269514722  
Null Hypothesis is accepted.
```

In line 9, a sample of size 12 is taken from the population defined on line 7 of the code. The sample mean is compared against the population (true) mean to get the z-value in line 24 of the code. The critical value of z that corresponds to a particular p-value is computed using SciPy.stats function norm.ppf () .

We reject the null hypothesis if the z critical value is less than the z test value. Since z test value, -0.217, is less than z critical value, 1.645, we accept the null hypothesis.

7.4.2 Two-tailed z Test

Suppose the hypotheses to be tested are as follows:

Null Hypothesis : $H_0: \text{sample_mean} = 23$

Alternate Hypothesis : $H_1: \text{sample_mean} \neq 23$

In this case, the alternative hypothesis is on both sides of the null hypothesis, i.e., the mean of the sample is not equal to 23. The test that can be performed to test these hypotheses is called a two-tailed z test.

To implement the two-tailed z test, type the following Python script:

```
1. import numpy as np
2. from numpy import random
3. import math
4. import scipy.stats as st
5.
6. population = np.array([20,12,22,32,52,1,22,30,40,50,6,12,
   32,52,1,22,3,7,12,32,52,62,12,32,52,11,22,3])
7. sample_size = 12
8. sample = random.choice(population,12)
9.
10. population_mean = population.mean()
11. print("Population Mean: ", population_mean)
12.
13. sample_mean = sample.mean()
14. print('Sample Mean:', sample_mean)
15.
16. # Null Hypothesis
17. # sample_mean = 23.5
18. # Alternate Hypothesis
19. # sample_mean != 23.5
20.
21. pop_stdev = population.std()
22. print('Population Standard Deviation: ', pop_stdev)
```

```
23. z_test = (sample_mean - population_mean)/ (pop_stdev/math.sqrt(sample_size))
24. print('Z test value:', z_test)
25.
26. # calculation of p-value
27. if (z_test>0):
28.     p_val = 1 - st.norm.cdf(z_test)
29. else:
30.     p_val = st.norm.cdf(z_test)
31. print('p-value:' ,p_val)
32.
33. # Right Tail
34. confidence_level = 0.95
35. alpha = 1-confidence_level
36. z_critical_val1 = st.norm.ppf(confidence_level+alpha/2)
37.
38. # Left Tail
39. z_critical_val2 = st.norm.ppf(alpha/2)
40.
41. print("Z critical value 1: ", z_critical_val1)
42. print("Z critical value 2: ", z_critical_val2)
43.
44. #if(z_test > z_critical_val1 or z_test < z_critical_val2):
    either use this line or the line following it
45. if(p_val < alpha/2):
46.     print("Null Hypothesis is rejected.")
47. else:
48.     print("Null Hypothesis is accepted.")

Output:
Population Mean: 25.214285714285715
Sample Mean: 29.75
Population Standard Deviation: 18.020538169498916
Z test value: 0.871903770861863
p-value: 0.19163043818841108
Z critical value 1: 1.959963984540054
Z critical value 2: -1.959963984540054
Null Hypothesis is accepted.
```

In line 8, a sample of size 12 is taken from the population defined on line 6 of the code. The sample mean is compared against the population (true) mean to get the z-value in line 23 of the code. A p-value is computed in lines 27 to 30 by using if-else conditions. This is because if the z-value is positive, it has to be compared to the p-value on the right side of the sampling distribution.

However, a negative z-value should be compared to the p-value on the left side of the distribution. Since it is a two-tailed test, two critical values of z are computed using SciPy.stats function `norm.ppf()` in lines 36 and 39 of the code. Finally, we can either test our hypothesis using `z_test` or `p_val`. Here, in line 45, we test if the p-value is less than half of the significance level, we reject the null hypothesis.

Since the p-value, 0.1916 is greater than $\alpha/2$, we accept the null hypothesis.

7.5 Exercise Questions

Question 1:

The probability of rejecting the null hypothesis when it is false corresponds to:

- A. α
- B. β
- C. Type I error
- D. Type II error

Question 2:

Since α corresponds to the probability of Type I error, then $1-\alpha$ corresponds to:

- A. Probability of rejecting H_0 when H_0 is true
- B. Probability of accepting H_0 when H_0 is true
- C. Probability of accepting H_0 when H_1 is true
- D. Probability of rejecting H_0 when H_1 is true

Question 3:

In a hypothesis testing, if β is type II error, and $1-\beta$ is the power of the test, then which statement corresponds to $1-\beta$?

- A. probability of rejecting H_0 when H_1 is true
- B. probability of failing to reject H_0 when H_1 is true
- C. probability of failing to reject H_0 when H_0 is true
- D. probability of rejecting H_0 when H_0 is true.

Question 4:

In a hypothesis, what is the effect on the region of rejection when the level of significance α is reduced?

- A. The rejection region is reduced in size
- B. The rejection region is increased in size
- C. The rejection region is unaltered
- D. The answer depends on the alternative hypothesis

Question 5:

Which statement(s) is true?

- A. A very small p-value indicates that the actual data differs from the expected under the null hypothesis
- B. p-value measures the probability that the hypothesis is true
- C. p-value measures the probability of Type II error
- D. A large p-value indicates that the data is consistent with the alternative hypothesis

Question 6:

The average growth of a specific type of tree is 5.3 inches in a year. A researcher hypothesizes that a new variety of that tree should have greater yearly growth. A random sample of 100 new trees results in average yearly growth of 5.9 inches and a standard deviation of 1.5 inches. The appropriate null and alternative hypotheses to test the hypothesis are:

- A. $H_0: \mu=5.9$ against $H_1: \mu>5.9$
- B. $H_0: \mu=5.9$ against $H_1: \mu\neq5.9$
- C. $H_0: \mu=5.3$ against $H_1: \mu>5.3$
- D. $H_0: \mu=5.3$ against $H_1: \mu\neq5.3$

8

Bayesian Inference

The Bayesian statistics, in contrast to the frequentist statistics, interprets probability as a degree of belief as to the prior knowledge. The use of priors about the happening of events makes this a subjective view of probability because the prior knowledge may vary from one expert to another.

The prior knowledge is also used to associate a probability with the parameters to be estimated or the hypotheses to be tested. Our degree of belief changes as new evidence/data appears. The belief can be defined as a probability distribution. For instance, the belief about average heights of European males can be expressed as “there is an 80 percent probability that the average height is between 175 and 185 cm.” Bayes’ theorem is employed by Bayesian inference to update the probability of a hypothesis based on the evidence or the availability of data.

Bayesian statistics mainly finds its application when we have to update some parameters based on dynamic data or a sequence of data. In the subsequent sections, we revise the Bayes’ rule and present methods based upon Bayesian inference.

8.1 Conditional Probability

The events occurring around us can be independent such as the tossing of a single coin. However, many events are dependent upon other events in such a way that they are affected by previous events. Conditional probability is a way to describe the dependency of the events. The probability of an event that is based upon / conditional to another event is called a conditional probability.

For instance, suppose we have 5 balls in a bag, out of which 2 are blue, and 3 are red. We define the following events:

1. Event A: Getting a blue ball

Chances of getting a blue ball are 2 in 5:

$$P(A) = 2/5$$

2. Event B: Getting a red ball

Chances of getting a red ball are 3 in 5:

$$P(B) = 3/5.$$

What happens when we draw a ball from the bag and then draw another ball? The second draw now depends upon (conditional to) the previous draw. Thus, it is a dependent event. Suppose we get a blue ball in the first draw, i.e., $P(A) = 2/5$, and a red ball in the second draw. The probability of getting the second red ball is not $3/5$ anymore. It will be $3/4$ because we do not replace the first ball. The total number of balls present in the bag is 4 for the second draw.

If we have to compute the probability of getting the first blue ball and the second red ball, we have to combine the probabilities of both events happening together. It is known as the **joint probability** of A and B and denoted as $P(A \cap B)$:

$$P(A \cap B) = 2/5 \times 3/4 = 6/20 = 3/10.$$

The symbol \cap is for the intersection of the events. The expression for $P(A \cap B)$ combines the probability of event A, $P(A)$, and the probability of event B given A has occurred before, $P(B|A)$.

$$P(A \cap B) = P(A) \cdot P(B|A)$$

$$P(B|A) = P(A \cap B) / P(A)$$

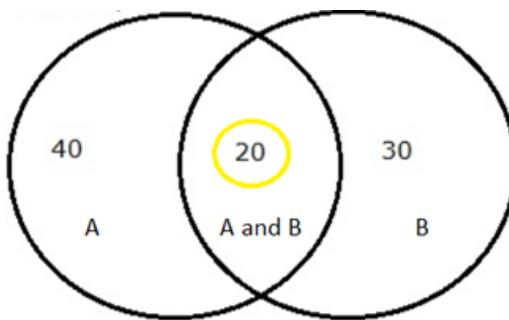


Figure 8.1 Explanation of the joint probability $P(A \cap B)$.

Thus, the conditional probability can be defined as the likelihood of occurrence of an event based on the occurrence of a previous event. Conditional probability is calculated by multiplying the probability of the preceding event by the updated probability of the conditional event. As another example:

- Event A is that it is raining outside, and it has a 40 percent chance of raining today. The probability of event A is $P(A) = 0.4$.
- Event B is that a person needs to go outside, and it has a chance of 30 percent, i.e., $P(B) = 0.3$.

Let the probability that both events happen together is $P(A \cap B) = 0.2$ or 20 percent.

Now, we are interested to know the probability or chances of occurrence of rain given the person has come out. The probability of rain given the person went out is the conditional probability $P(A|B)$ that can be given as,

$$P(A|B) = P(A \cap B)/P(B) = 0.2/0.3 = 0.66 = 66.6\%.$$

8.2 Bayes' Theorem and the Bayesian Philosophy

In many situations, the events or experiments are not repeatable. For example, find the probability of the candidates winning in elections to be held next month? Obviously, the event of the election will happen only once next month. It is not repeatable. The frequentist approach fails in such scenarios. The Bayesian approach will make use of prior knowledge about the candidates, such as the popularity of the candidates, the result of previous elections, etc., to make inferences from the available data.

In many cases of scientific experiments, typically, some prior knowledge of the experiment is available. Discarding this prior information may affect the results of the inference. The Bayesian statistics makes use of this already available information about the process of making decisions. This prior information is combined with the present information using Bayes' theorem.

Frequentist inference works with the probability of the data given the hypothesis $P(D|H)$ is true. Bayesian inference, contrary to the frequentist approach, concentrates on $P(H|D)$, i.e., the probability of the hypothesis, given the data. This implies that the data is treated as fixed, not a random variable. However, hypotheses are considered as random variables.

Thus, probabilities are associated with hypotheses to be tested.

The Bayesian probability is interpreted as a **degree of belief**. Suppose, in a rainy season, it rains most of the days of the month in some geographical area. The natives of that area believe that the chances of having a rainy day are 80 percent or 0.8. This becomes the prior probability that is based on the degree of belief of the natives. We write:

$$P(\text{rainy day} = \text{true}) = 0.8,$$

where a degree of belief of 80 percent that a randomly chosen day receives rain is the prior probability of having a rainy day in the absence of any other evidence. It is important to point out that the degree of belief denotes the probability of happening of a particular event before we make an actual observation of the event.

Obviously, the priors of the events can change when we observe the actual outcomes or events. In other words, the presence of evidence may cause our degree of belief in the event to change.

As a practical example, we might want to calculate the probability that a patient has heart disease, given they are obese. We define event A as “patient has a heart disease.” From previous experience and the data collected from different hospitals, it is known as a **prior** belief that 15 percent of patients have heart disease, i.e., $P(A) = 0.15$. Furthermore, we define event B as “patient is obese.” From the past collected data, 10 percent of the patients are obese, i.e., $P(B) = 0.1$.

Now, suppose we know from hospital tests data that 20 percent of the patients diagnosed with heart disease are obese, i.e.,

$P(B|A) = 0.2$. The probability that a patient is obese, given that they have heart disease, is 20 percent. $P(B|A)$ is referred to as a **likelihood** function. Now, we are interested in finding out the probability that a patient has heart disease if they are obese, i.e., $P(A|B)$. This new conditional probability in the presence of **evidence**, event B: obesity, is called a **posterior** probability.

Bayes' theorem computes posterior from the likelihood, evidence, and prior as follows:

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

$$P(A|B) = \frac{(0.2)(0.15)}{0.1} = 0.3$$

This implies that if a patient is obese, their chances of having heart disease are 0.3 or 30 percent, in contrast to 15 percent chances in the absence of the evidence, as suggested by the past data. Thus, the presence of evidence of one event alters the posterior probability of the other event.

In terms of hypothesis testing, some terminology related to Bayes' theorem is given as follows:

- **P(A):** the probability of hypothesis A being true, in spite of the data. This is the **prior** probability of A or the unconditional probability.
- **P(B):** the probability of the data, regardless of the hypothesis. This is known as **evidence**.
- **P(B|A):** the probability of data B given that hypothesis A is true. This is known as the **likelihood** of data B conditional on hypothesis A.
- **P(A|B):** the probability of hypothesis A given the data B. This is known as the **posterior** probability.

8.3 Computations in Bayesian Inference

Suppose we want to estimate the average height of adults. We assume that height has a Normal distribution. The frequentist approach to estimate adult height could be as follows.

The height is assumed to be a fixed number, not a random number. Thus, we cannot assign probabilities to the average height being equal to a specific value. Next, we shall collect a sample of the population consisting of adults. The population mean is then estimated as the sample mean that is consistent with the data. The most likely/consistent value of the average height is obtained by a method called **maximum likelihood estimate**. The maximum likelihood estimate of the population mean is usually equal to the sample mean in frequentist statistics.

The Bayesian approach to estimate adult height could be as follows. The average or the mean, a fixed value, is described as having a probability distribution over the possible values of the average heights. The sample data is obtained from the population. This data is used to update this distribution. The updated distribution of the parameter becomes narrower as new sample data is used by Bayes' rule to update the estimate of the average height of adults.

8.3.1 Computing Evidence: Total Probability

The computation of the posterior probability in the Bayes' theorem involves the computation of prior, likelihood, and evidence. The evidence is usually calculated by the law of total probability.

Suppose we have a sample space represented as a rectangle in Figure 8.2 (a). Since three mutually exclusive events B_1 , B_2 , and B_3 , cover the whole sample space, these are exhaustive. Now, any event A can be considered as the intersection of the event with all three mutually exclusive exhaustive events.

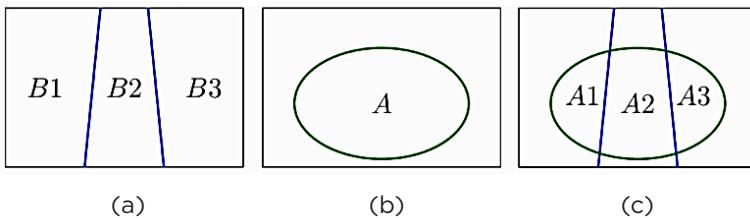


Figure 8.2: The law of total probability. To compute the probability of an event A (given as an oval) in the sample space (rectangular), we have to compute the probability of event A with respect to mutually exclusive and exhaustive events B_1 , B_2 , and B_3 .

To find the probability of A , we can compute and add the probabilities of mutually exclusive parts of event A : A_1 , A_2 , and A_3 as follows:

$$P(A) = P(A_1) + P(A_2) + P(A_3)$$

where

$$A_1 = A \cap B_1, \quad A_2 = A \cap B_2 \text{ and } A_3 = A \cap B_3.$$

Thus,

$$P(A) = P(A \cap B_1) + P(A \cap B_2) + P(A \cap B_3)$$

Now, from the definition of conditional probability,

$$P(A \cap B) = P(A|B)P(B).$$

The probability of event A in terms of probabilities conditional to the mutually exclusive events is now given as:

$$P(A) = P(A|B_1)P(B_1) + P(A|B_2)P(B_2) + P(A|B_3)P(B_3)$$

This can be compactly written as:

$$P(A) = \sum_i P(A \cap B_i) = \sum_i P(A|B_i)P(B_i)$$

The last equation is known as “**the law of total probability**” in the sense that the total probability of any event A can be computed by its parts A_1, A_2 , and A_3 , which in turn can be computed from the conditional probabilities. This equation demonstrates that the computation of the evidence requires computations comparable to the number of partitions of the sample space. This amounts to a lot of computations when the partitions become large.

In the case of a continuous random variable B , the summation is replaced by its continuous counterpart, the integral. Most Bayesian calculations of the posterior become intractable due to the large computational overhead due to the calculation of the evidence in the denominator. Therefore, instead of computing the exact sum (discrete case) or integral (continuous case), we take samples from the posterior in such a way that its good approximation is obtained. Monte Carlo methods, discussed in Section 8.4, are used to sample the posterior.

8.3.2 Steps to Follow for Bayesian Inference

We have to compute the posterior from prior, evidence, and likelihood functions. We follow these steps to estimate the posterior in the Bayesian inference:

- Establish a belief about the prior before observing the data. Furthermore, assume a likelihood function, as well. This is equivalent to assuming some probability distributions for the prior and the likelihood.

- Use the data to update our belief about the model. This corresponds to computing the posterior from the available data.
- If data is available sequentially, we keep on updating our belief based upon new evidence.

8.4 Monte Carlo Methods

Monte Carlo methods are algorithms that depend on repeated random sampling to obtain approximate results. The basic concept behind Monte Carlo sampling is to employ randomness to approximate solutions to the problems for which an exact solution is difficult to compute. These methods are widely used to find the approximate (numerical) integration and to generate random draws from a probability distribution.

Monte Carlo sampling is particularly useful to compute an approximate solution to the posterior in a reasonable time that is hard to compute otherwise. Monte Carlo methods can be used to predict the expected outcome of an experiment by calculating the outcome multiple times with different random inputs.

Monte Carlo methods rely on randomness and the *law of large numbers* that can be explained as follows. The sample statistic such as mean for a sample size n converges to the population mean as n approaches infinity. In other words, there would be a very high probability that the sample mean is very close to the population mean when n is very large.

Suppose we want to find the probability of tails in a coin flip experiment. We call this probability p . Initially, we do not have any idea or any prior information about this number p . We start to flip a coin and record the outcomes in each flip. This

becomes our observed data. Our inference of the probability p might change when we repeat the coin flip experiment again and again. To implement this in Python, type the following script:

```
1. #Import required libraries
2. import random
3. import matplotlib.pyplot as plt
4.
5. #Let Heads = 0 and Tails = 1
6. #Definition of function that randomly gives us either 0 or
  1 as the output.
7. def coin_flip():
8.     return random.randint(0,1)
9.
10. #Check the return value of coin_flip()
11. coin_flip()
12.
13. #Monte Carlo Simulation
14. #Empty list to store the probability values.
15. list1 = []
16.
17.
18. def monte_carlo(n):
19.     results = 0
20.     for i in range(n):
21.         flip_result = coin_flip()
22.         results = results + flip_result
23.
24.         #Calculating probability value:
25.         prob_value = results/(i+1)
26.
27.         #Append the probability values to the list:
28.         list1.append(prob_value)
29.
30.         #Plot the results:
31.         plt.axhline(y=0.5, color='r', linestyle='--')
32.         plt.xlabel("Iterations")
33.         plt.ylabel("Probability")
```

```

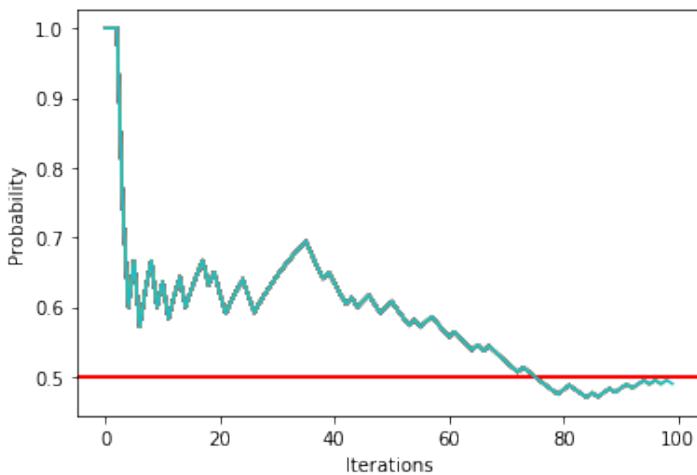
34.         plt.plot(list1)
35.
36.     return results/n
37.
38. #Calling the function:
39.
40. answer = monte_carlo(100)
41. print("Final value : ",answer)

```

Output:

The flipped coin value is 1

Final value : 0.49



This code imports libraries random and Matplotlib for generating random numbers and for plotting the results, respectively. We define a function coin_flip () at line 7 of the code. This function uses another function random.randint () to generate a random integer, either 0 or 1.

Next, to save the outcomes of each coin flip, we create an empty list in line 15 of the code. This list is updated in another function monte_carlo (n) that takes the number of repetitions n of the experiment as its input. We use a for loop that iterates n number of times, each time the function coin_flip () is called,

and its random outcome, either 0 or 1, is appended to the list. Thus, the size of the list grows until it is equal to the number of repetitions of the experiment.

The value of the probability of getting Tails is calculated in line 25 of the code. Note that we divide the results that holds the sum of 1's (Tails) by $(i+1)$ instead of i . This is because the for loop starts from 0 and goes to $n-1$ (a total of n times).

Next, we plot the results using lines 31 to 34 of the code. Finally, the function returns the final value of the probability after n trials of the experiment in line 36 of the code.

The designed function `monte_carlo (n)` is called in line 40 of the code for an input of 100. The output shows that the initial outcome of the `coin_flip` was 1. This is depicted in the output plot. As more and more flips are made, the computed probability approaches 0.5, as per our expectation.

8.5 Maximum a Posteriori (MAP) Estimation

The maximum a posteriori (MAP) estimation of a model is the mode of the posterior distribution, where the mode is the value of the random variable at which the probability mass (or density) function takes its maximum value.

The mode of distribution is found by numerical optimization methods. PyMC3 allows us to compute the MAP of distribution with the `find_MAP ()` function. The MAP is returned as a parameter point, which is always represented by a Python dictionary of variable names: NumPy arrays of parameter values as key:value pairs.

In the following code, we find the MAP estimate of our model.

```
1. import pymc3 as pm
2. import numpy as np
3.
4. # True parameter values
5. alpha, sigma = 1, 1
6. beta = [1, 2.5]
7.
8. # Size of dataset
9. size = 100
10.
11. # Predictor variable
12. X1 = np.random.randn(size)
13. X2 = np.random.randn(size) * 0.2
14.
15. # Simulate outcome variable
16. Y = alpha + beta[0] * X1 + beta[1] * X2 + np.random.
   randn(size) * sigma
17.
18. # Initialize a PyMC model
19. basic_model = pm.Model()
20.
21. # Define model parameters
22. with basic_model:
23.
24.     # Priors for unknown model parameters, create Normal
      variables
25.     # These are stochastic variables
26.     alpha = pm.Normal("alpha", mu=0, sigma=10)
27.     beta = pm.Normal("beta", mu=0, sigma=10, shape=2)
28.     sigma = pm.HalfNormal("sigma", sigma=1)
29.
30.     # Expected value of outcome, completely deterministic
      variable
31.     mu = alpha + beta[0] * X1 + beta[1] * X2
32.
33.     # Likelihood of observations in the model
34.     Y_obs = pm.Normal("Y_obs", mu=mu, sigma=sigma,
   observed=Y)
```

```
35.  
36.  
37. # Use find_MAP to find maximum a posteriori from a pymc  
    model  
38. map_estimate = pm.find_MAP(model=basic_model)  
39. print(map_estimate)  
Output:  
100.00% [19/19 00:00<00:00 logp = -163.64, ||grad|| = 11.014]  
{'alpha': array(1.03540327), 'beta': array([0.85459263,  
2.29026671]), 'sigma_log__': array(0.0619924), 'sigma':  
array(1.06395426)}
```

The output shows the estimated values of parameters, alpha, beta, and sigma. It can be observed that the estimated MAP values are closer to the actual values defined in lines 5 and 6 of the code. We generate input (predictor) variables X1 and X2 in lines 12 and 13 of the code. The output is assumed to be a linear combination of the input variables with some noise as defined in line 16. The basic PyMC model is initialized using `pm.Model()` function. Line 22 uses the keyword “with” to specify the model parameters.

We define parameters alpha and beta to be Normally distributed, whereas the parameter sigma to be half Normal. The half Normal distribution has just half of the Normal distribution. For example, if any parameter such as the standard deviation assumes positive values only and follows a Gaussian distribution, we can model it as a half Normal. The likelihood of the observations is also modeled as a Normal distribution in line 34. Finally, function `find_MAP()` in line 38 finds the MAP estimate of the parameters.

Sometimes, the MAP estimate of the model is not acceptable, particularly when the distribution is either multimodal (having multiple modes that are meaningfully different), or at an extreme.

PyMC3's function `find_MAP()` is used less commonly in many situations. If we wish a point estimate such as the mode, we usually get it from the posterior using Monte Carlo sampling methods.

8.6 Credible Interval Estimation

The credible interval is the Bayesian counterpart of the confidence interval for the frequentist approach to the interval estimates.

Remember, in the frequentist statistics, we defined the probability by frequent runs of the experiment. It is assumed that the population parameters, such as average height and the standard deviation of the height, are fixed. This parameter, for example, a sample mean is computed from the sample, and its sampling distribution, is used for inference of the population mean. Since it is often difficult to conduct many trials of the experiment, we resort to the *central limit theorem* that tells us that taking large random samples from the population with replacement results in a Normal distribution for the sample means. The frequentist approach usually chooses an interval in which the population parameter, such as the mean of the distribution, lies 95 percent of the time.

In the Bayesian inference of the credible interval, sampling distributions are not used. The degree of belief about the population parameter is modeled as a posterior distribution. For instance, in the coin toss experiment presented in Section 8.4, our degree of belief or the probability of getting a tails or a heads is continuously updated as more and more flips of the coin are presented to us.

For instance, we want to estimate the posterior probability of getting heads in a coin toss experiment using Bayesian inference. Recall from Chapter 3, this type of experiment is called a binomial experiment.

8.6.1 Beta Distribution as a Prior

In Bayesian statistics, we have to assume a prior distribution for the parameters to be estimated. The uniform distribution can be used as a prior (flat prior) when the probability of occurrence of every possible outcome is equally likely. In case the coin is fair, the probability of getting heads equals the probability of getting tails, and a uniform distribution can be assumed for both heads and tails. However, it may not be the case, especially when the occurrence of some events is more likely than other events. In this case, the **Beta distribution** is usually used as a conjugate prior for the Bernoulli trials and the binomial model. The word conjugate means that both prior and posterior would result in a similar type of distribution, i.e., a beta prior results in a beta posterior.

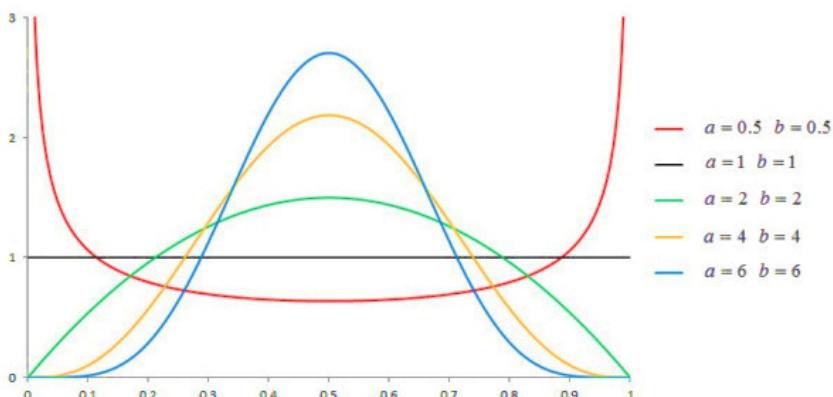


Figure 8.3: Beta distribution for different values of α and β .

Mathematically, the Beta distribution is given as,

$$\text{Beta}(\alpha, \beta) : \text{Prob}(x|\alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \quad 0 < x < 1$$

where B is the beta function that normalizes the Beta distribution to make it a valid probability distribution. It can be observed from the aforementioned equation, and Figure 8.3 that for varying values of α and β , the shape of the Beta distribution varies. If both α and β are set to 1 in the Beta distribution, we get a uniform distribution that is a special case of the Beta distribution. The greater the difference between α and β , the more *skewed* the resulting Beta distribution is.

Thus, the Beta distribution is able to capture the probability of x successes in repeated trials of a binomial experiment even when the outcomes have different probabilities.

Suppose our initial prior belief about the flipping of a coin is that both heads and tails are equally likely. Thus, we can assume that priors for both parameters α and β are 1. However, when we flipped the coin 100 times, we observed that heads occurred more than tails. The following Python code shows the simulation of the coin flip experiment.

```
1. ### Estimating the posterior probability in a coin flip
   experiment done multiple times
2. import numpy as np
3. from scipy.stats import beta, t, norm
4. from scipy.special import btdtri
5. import matplotlib.pyplot as plt
6.
7. p = 0.7
8. n = 100
9. np.random.seed(5) # to reproduce results
10. num_heads = np.random.binomial(p=p, n=n)
11. num_tails = n - num_heads
```

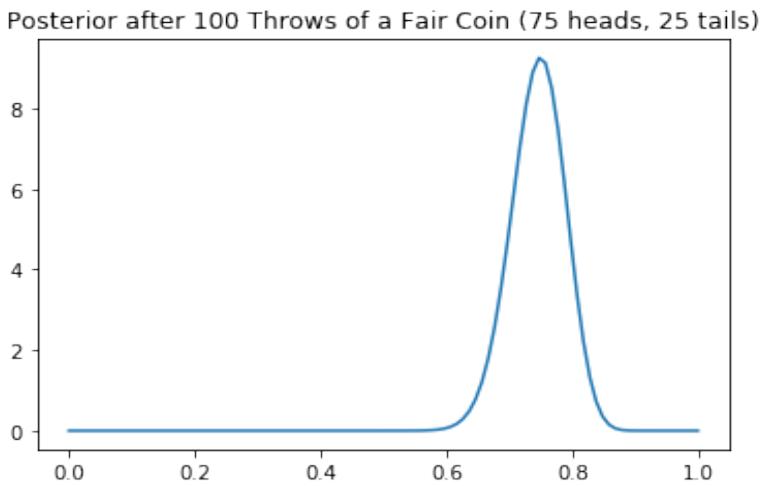
```
12. print("successes (heads) = %i, failures (tails) =  
    %i"%(num_heads, num_tails))  
Output:  
successes (heads) = 72, failures (tails) = 28
```

In this code, we have assumed that success corresponds to the heads, and the failure corresponds to the tails. We have modeled the occurring of the heads as $p = 0.7$ in line 7 of the code. When a binomial experiment was run for $n=100$ times, we observed 72 heads and 28 tails. Line 9 uses `np.random.seed(5)` to ensure the same outcome in each run of the code to reproduce results.

Our initial belief (prior) about a fair coin is updated. Since Beta distribution is used as a prior for a binomial model, the following Python code shows the updating of our belief as the posterior distribution.

```
1. ## initially assumed priors for heads and tails before the  
   experiment  
2. prior_a = 1  
3. prior_b = 1  
4.  
5. a = prior_a + num_heads  
6. b = prior_b + num_tails  
7. rv = beta(a, b)  
8.  
9. x = np.linspace(0, 1, 100)  
10. plt.plot(x, rv.pdf(x))  
11. plt.title("Posterior after 100 Throws of a Fair Coin (%i  
    heads, %i tails)" %(num_heads,num_tails))  
12. plt.show()
```

Output:



Lines 2 and 3 of the code show our initial belief before the run of the experiment. We update our belief in lines 5 and 6 of the code by simply adding the number of heads to α and the number of tails to β . Lines 9 to 12 are used to plot the posterior distribution. It can be seen that the distribution is skewed toward the right because the value of α is greater than β .

To find the credible interval, we type the following code:

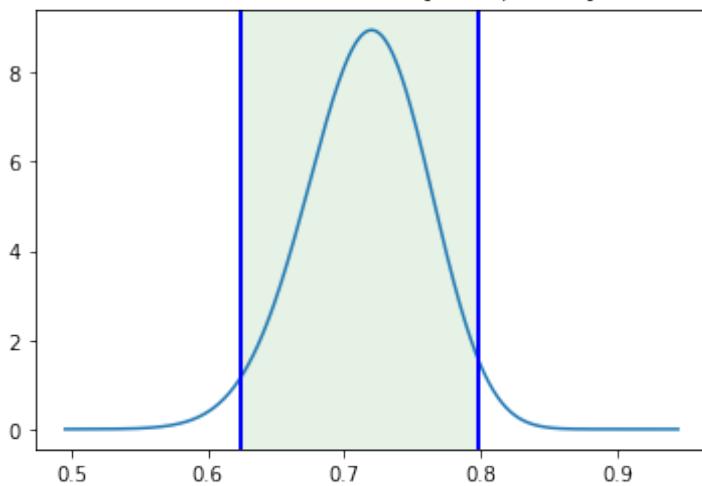
```

1. # The p-th quantile of the beta distribution.
2. b_up = btdtri(a, b, 0.975)
3. b_lo = btdtri(a, b, 0.025)
4.
5. plt.plot(x, rv.pdf(x))
6.
7. right_line = plt.axvline(b_up, lw=2, color='blue')
8. left_line = plt.axvline(b_lo, lw=2, color='blue')
9. fill = plt.axvspan(b_lo, b_up, alpha=0.1, color='green')
10.
11. plt.title("95% credible interval: [% .3f, % .3f]"%(b_lo,
   b_up))
12. plt.show()

```

Output:

95% credible interval: [0.625, 0.799]



Lines 2 and 3 of the code uses the `btdtri()` function from the `SciPy.special` package. This function computes the quantiles of the Beta distribution specified by the inputs `a` and `b` (corresponding to α and β). It means that it uses the inverse cumulative distribution function (the quantile function) of the Beta distribution. Details of the quantile function are given in Chapter 5. For example, 0.025 and 0.975 values in these lines compute the points on the Beta distribution that corresponds to the 97.5 percentile and 2.5 percentile of the distribution. The difference $97.5 - 2.5 = 95$ gives the 95 percent credible interval. The 95 percent credible interval means that there is a 95 percent probability that our parameter (probability of success or heads) falls within the range of 0.625 and 0.799.

8.6.2 Gamma Distribution as a Prior

A Gamma distribution $\Gamma(\alpha, \beta)$ with a positive shape parameter α and a positive rate parameter β can be given as:

$$\Gamma(\alpha, \beta) : Prob(x|\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} \quad \alpha > 0, \beta > 0$$

The Gamma distribution is defined for positive values of the random variable x . The rate parameter β can be considered as the inverse of the scale parameter θ , i.e., $\theta = 1/\beta$. $\Gamma(\alpha)$ is called the Gamma function. This is given as:

$$\Gamma(\alpha) = \int_0^{\infty} t^{\alpha-1} e^{-t} dt$$

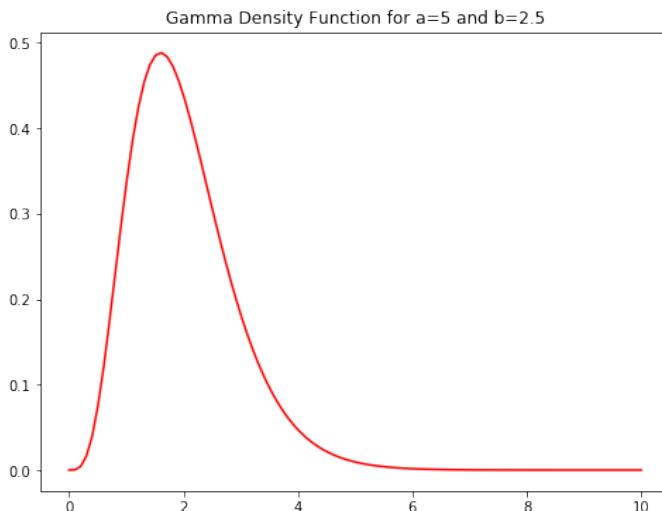
The Gamma function computes the factorial of the positive real numbers. Remember, factorial is only defined for integers. However, the integral given in the Gamma function interpolates the factorial for positive non-integers as well. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

The factorial of floating-point (non-integers) such as 5.2! is undefined. However, $\Gamma(5.2)$ is defined due to the Gamma function that computes the integration.

The following Python code plots a Gamma distribution for shape or alpha parameter of 5 and the rate or beta parameter of 2.5.

```
1. # Parameters of the prior gamma distribution.
2. a = 5 # shape or alpha
3. b = 2.5 # rate = 1/scale or beta
4.
5. x = np.linspace(start=0, stop=10, num=100)
6. plt.figure(figsize=(8, 6))
7. plt.plot(x, stats.gamma.pdf(x,a=a,scale=1/b), 'r-')
8. plt.title('Gamma Density Function for a={} and b={}'.format(a,b))
```

Output:

The presence of an exponential function, shape, and rate parameters enable a Gamma distribution to serve as a conjugate prior in Bayesian statistics for the exponential family of distributions such as a **Poisson distribution**.

Suppose it is believed that the average number of goals in a football match is 2.5. We want to find out the probability of exact k goals in a match. This can be modeled as a Poisson distribution, which can be given as:

$$f(k|\lambda) : P(X = k|\lambda) = \frac{e^{-\lambda} \lambda^k}{k!}$$

where λ represents the rate, e is the Euler number ($e=2.71828\dots$), and k represents the number of occurrences of the event. For example, to find out exactly 1, 2, and 5 goals in a match, we can perform calculations as follows:

$$P(X = 1|2.5) = \frac{e^{-2.5} 2.5^1}{1!} = 0.205$$

$$P(X = 2|2.5) = \frac{e^{-2.5} 2.5^2}{2!} = 0.256$$

$$P(X = 5|2.5) = \frac{e^{-2.5} 2.5^5}{5!} = 0.067$$

Now, suppose we observe the results of 100 football matches and print the average number of goals that occurred in all the 100 matches. The following Python script simulates 100 samples.

```

1. import numpy as np
2. import scipy.stats as stats
3.
4. np.random.seed(5)          # using seed to reproduce the
   results
5. n = 100                  # number of samples.
6. actual_lambda = 2.5       # true parameter.
7. # sample array.
8. y = np.random.poisson(lam=actual_lambda, size=n)
9. print('The generated Poisson samples are:', y)
10. print('\nThe mean of samples is %.3f' %y.mean())
Output:
The generated Poisson samples are: [2 5 1 2 3 3 1 1 3 0 3 1 4
 2 4 2 1 4 5 1 2 1 2 5 2 3 0 2 4 4 2 4 2 8 2 1 2 3 1 2 4 6
 3 0 3 5 3 3 2 2 1 4 5 3 1 1 1 2 2 5 4 4 2 1 3 5 3 2 2 3 3
 3 3 6 3 1 2 1 3 0 1 0 3 5 3 2 2 2 0 1 1 6 7 1 2 4 5 2 3 4]

The mean of samples is 2.640

```

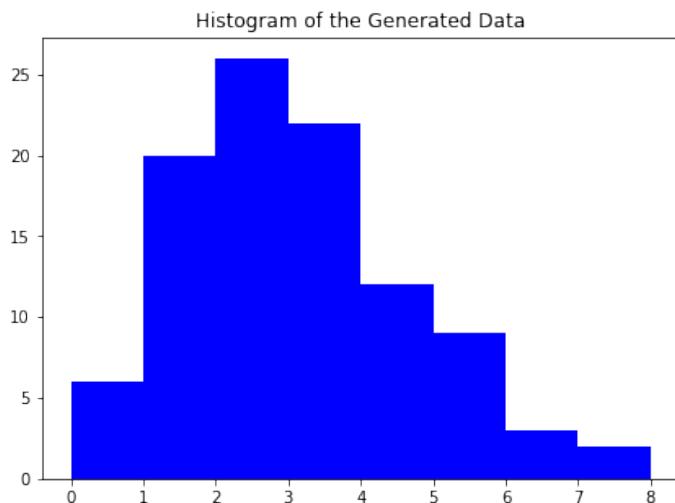
We use `np.random.seed(5)` to reproduce the results. The average number of goals from 100 matches equals 2.64 that is quite close to the `actual_lambda`, 2.5, of the Poisson distribution. We plot the histogram of the 100 samples as follows:

```

1. import matplotlib.pyplot as plt
2. %matplotlib inline
3. # Histogram of the sample data
4. plt.figure(figsize=(7, 5))
5. plt.hist(y, bins=8, color='b')
6. plt.title('Histogram of the Generated Data')
7. plt.show()

```

Output:



A peak can be observed between 2 and 3 in the histogram that indicates 2 to 3 goals in more than 25 matches.

Since each match is independent of other matches, the observations or the number of goals scored in each match are independent of each other. Thus, the likelihood function given the λ parameter of the Poisson model is given as:

$$f(X|\lambda) = \prod_{i=1}^n \frac{e^{-\lambda} \lambda^{X_i}}{X_i!} = \frac{e^{-n\lambda} \lambda^{\sum_{i=1}^n X_i}}{\prod_{i=1}^n X_i!}$$

where the symbol \prod means the product of individual probabilities. For instance, if we want to find the probability

of the event of observing 2 goals in the first 5 matches, the aforementioned equation becomes

$$\begin{aligned} P(X = 2, 2, 2, 2, 2|2.5) &= \left(\frac{e^{-2.5} 2.5^2}{2!}\right) \left(\frac{e^{-2.5} 2.5^2}{2!}\right) \left(\frac{e^{-2.5} 2.5^2}{2!}\right) \\ &\quad \left(\frac{e^{-2.5} 2.5^2}{2!}\right) \left(\frac{e^{-2.5} 2.5^2}{2!}\right) = 0.256^5 = 0.001 \end{aligned}$$

Thus, the chances of observing exactly 2 goals in 5 matches are very low at 0.001. If we have to find the probability of observing 2 goals in all 100 matches, we would have a very small number $0.256^{100} \sim 0$. Due to this reason, we mostly use **log likelihood** instead of simple likelihood due to very small probabilities. The log function amplifies small numbers. Thus, we get rid of this problem of dealing with very small numbers.

The computation of the probabilities in a Bayesian model becomes enormous when we have a lot of observations. Furthermore, the computation of the evidence, the denominator, is usually not easy. Thus, instead of computing the posterior directly from the evidence, likelihood, and the prior, we construct the posterior by taking its samples using **Monte Carlo techniques**.

We assume a Gamma distribution as the prior and the likelihood of the aforementioned Poisson process related to the average number of goals in a football match.

The following code shows how to perform this using Python's PyMC3 package.

```

1. import pymc3 as pm
2. model = pm.Model()
3. with model:
4.
5.     # Define the prior of the parameter lambda.
6.     lam = pm.Gamma('lambda', alpha=a, beta=b)
7.     # Define the likelihood function.
8.     y_obs = pm.Poisson('y_obs', mu=lam, observed=y)
9.     # Consider 1000 draws and 2 chains.
10.    trace = pm.sample(draws=1000, chains=2)

```

Output:

Auto-assigning NUTS sampler...
 Initializing NUTS using jitter+adapt_diag...
 Multiprocess sampling (2 chains in 4 jobs)
 NUTS: [lambda]

100.00% [4000/4000 00:02<00:00 Sampling 2 chains, 0
 divergences]
 Sampling 2 chains for 1_000 tune and 1_000 draw iterations
 (2_000 + 2_000 draws total) took 12 seconds.

Line 2 of the code initiates a PyMC model, whereas the parameters of the Bayesian model are specified using the keyword “with” with the model name. We define the Gamma prior for our process and the Poisson likelihood since the observed data is assumed to be coming from a Poisson distribution.

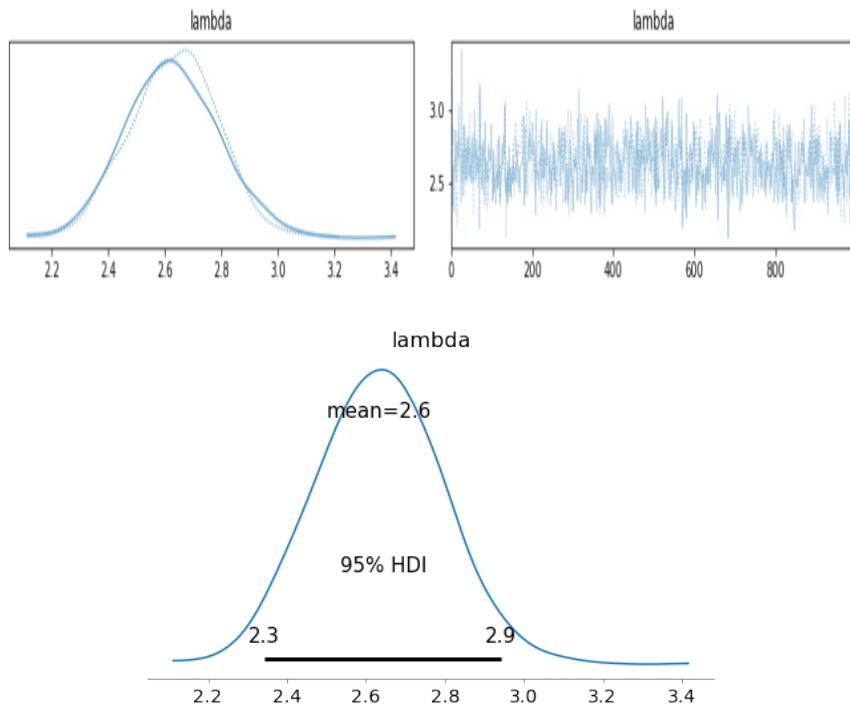
The parameters of the Gamma distribution in line 6 of the code are specified by variables a and b, whereas the parameter mu of the Poisson distribution corresponds to the λ , and is assumed to be Gamma distributed because, in Bayesian inference, the parameter is considered a random variable. This is in contrast to the frequentist inference, where the parameter of the model is considered fixed and is not assigned any probability.

The PyMC function sample () in line 10 takes the samples of the posterior based upon defined prior and the likelihood. The number of samples to draw is specified by the option draws, whereas the option chains specifies the number of Markov chains to take samples from. These samples are referred to as trace in PyMC. Markov chains is a process usually employed by Monte Carlo methods to get samples from the posterior. Since the number of chains is 2, in the output of the code, we see 2,000 draws, of which 1,000 good draws are returned to us.

To plot the output trace and the posterior, we may type the following commands:

```
1. pm.traceplot(trace)
2.
3. pm.plot_posterior(trace, hdi_prob=.95)
```

Output:



Line 1 of the code plots the posterior of the lambda parameter, which is assumed to be Gamma distributed in the PyMC model. The option hdi_prob plots the **Highest Density Interval (HDI)** of the posterior for the chosen percentage of density. HDI refers to the region in which the posterior has high density. In our case, we plot 95 percent HDI that corresponds to the 0.95 credible interval. It can be observed that the mean posterior lambda is 2.6, and the 0.95 credible interval lies between 2.3 and 2.9.

8.7 Naïve Bayes' Classification

The classification is the problem of estimating the output target variable when it is discrete. The difference between regression and classification is that in the former, the output variable is continuous, whereas, in the latter, the output variable is discrete that represents the classes or categories.

Naive Bayes' is one of the simplest classification algorithms that is based on Bayes' Theorem of probability. A basic assumption that Naive Bayes' classifier uses is the independence of input variables/features. This assumption is considered as naive which simplifies computations. In terms of probability, this assumption is called class conditional independence.

Suppose, based upon weather conditions, we want to predict whether to play or not on a particular day. We have access to some previous data along with the output label corresponding to whether the game was played or not. We use the Naïve Bayes' algorithm to predict /classify the output variable.

To implement the Naïve Bayes' algorithm in Python, we may write the following script:

```
1. #Import Gaussian Naive Bayes model
2. from sklearn.naive_bayes import GaussianNB
3.
4. from sklearn import preprocessing
5.
6. # Assigning features and label variables
7. weather=['Sunny','Sunny','Overcast','Rainy','Rainy',
   'Rainy','Overcast','Sunny','Sunny',
8. 'Rainy','Sunny','Overcast','Overcast','Rainy']
9. temp=['Hot','Hot','Hot','Mild','Cool','Cool','Cool',
   'Mild', 'Cool','Mild','Mild','Mild','Hot','Mild']
10.
11. play=['No','No','Yes','Yes','Yes','No','Yes','No','Yes',
   'Yes','Yes','Yes','Yes','No']
12.
13. #creating label Encoder
14. le = preprocessing.LabelEncoder()
15. # Converting string labels into numbers.
16. weather_encoded=le.fit_transform(weather)
17. print ("Weather:",weather_encoded)
18.
19. # Encode temp and play columns to convert string labels
   into numbers
20. temp_encoded=le.fit_transform(temp)
21. label=le.fit_transform(play)
22. print ("Temp:",temp_encoded)
23. print ("Play:",label)
24.
25. #Combining features weather and temp in a single variable
   (list of tuples).
26.
27. features=np.column_stack((weather_encoded,temp_encoded))
28. print ("Combined feature:",features)
29.
30. # Generate a model using naive Bayes' classifier in the
   following steps:
31. # 1. Create naive Bayes' classifier
32. # 2. Fit the dataset on classifier
33. # 3. Perform prediction
```

```
34.  
35. #Create a Gaussian Classifier  
36. model = GaussianNB()  
37.  
38. # Train the model using the training sets  
39. model.fit(features,label)  
40.  
41. #Predict Output for input 0:Overcast, 2:Mild  
42.  
43. predicted= model.predict([[0,2]])  
44. print ("Predicted Value:", predicted)  
Output:  
Weather: [2 2 0 1 1 1 0 2 2 1 2 0 0 1]  
Temp: [1 1 1 2 0 0 0 2 0 2 2 2 1 2]  
Play: [0 0 1 1 1 0 1 0 1 1 1 1 1 0]  
Combined feature: [[2 1]  
[2 1]  
[0 1]  
[1 2]  
[1 0]  
[1 0]  
[0 0]  
[2 2]  
[2 0]  
[1 2]  
[2 2]  
[0 2]  
[0 1]  
[1 2]]  
Predicted Value: [1]
```

This program uses the sci-kit learn library that contains numerous machine learning algorithms. Lines 7 to 11 are used to enter the input and output variables. Since strings are difficult to work with, we convert the string feature values and output labels into integers using the label encoding method `preprocessing.LabelEncoder()`. The function `fit_transform()` in lines 16, 20, and 21 converts the string labels into numbers.

We create a Gaussian Naïve Bayes' classifier model in line 36, and train this model using the available observations/training data using `model.fit()` in line 39. To predict a new test point corresponding to the input 0: overcast weather and 2: mild temperature, we use the `predict()` function that results in a predicted value of 1: play the game.

Naïve Bayes' is easy to implement and interpret. It performs better compared to other similar models when the input features are independent of each other. It requires a small amount of training data to estimate the test data.

The main limitation of Naïve Bayes' is the assumption of independence between the independent variables. If features are dependent, this algorithm cannot be applied. Dimensionality reduction techniques can be used to transform the features into a set of independent features before applying the Naïve Bayes' classifier.

8.8 Comparison of Frequentist and Bayesian Inferences

The frequentist inference assumes that the sample data is random and incomplete. It has come from an unknown population whose parameters have to be estimated. The sample data is used to make the inference about unknown population parameters. The frequentist inference assumes that there is a single fixed true value of the unknown parameter.

The frequentist inference uses confidence statements instead of direct probability about the parameters. Parameters are usually specified in terms of the confidence level we have about their true range of values. It is important to realize that

a confidence level of say 0.95 or 95 percent does not mean a probability of 0.95.

The main strength of the frequentist inference is that it does not need any prior knowledge of the process or experiment from which the inference has to be made about the population. The main weakness of the frequentist approach lies in its interpretation of the probability and the frequency of occurrence of a particular event in a long run of trials.

In the frequentist statistics, we cannot assign probabilities to those events which cannot be repeated. However, it is normal to assign probabilities to non-repeatable events in Bayesian statistics.

The Bayesian inference employs a single tool, Bayes' theorem, in contrast to the frequentist inference that has to use many different tools such as z-value, p-value, and critical regions, etc., to infer from the data.

In Bayesian statistics, it is easy to get rid of difficult parameters by marginalizing them out of the joint posterior distribution. Furthermore, Bayesian statistics provides us the means to determine the distribution of future observations. This is not an easy task if we do things in a frequentist way.

The main drawback of the Bayesian inference is the subjective view of probability. The prior about an event or a process may vary from one expert to another, which would result in different estimates of the posteriors.

8.9 Exercise Questions

Question 1:

Which distribution is usually used as a prior for Bernoulli trials and binomial experiments?

- A. Normal distribution
- B. Poisson distribution
- C. Beta distribution
- D. Gamma distribution

Question 2:

Which distribution is usually used as a prior for the distributions involving a parameter related to the average rate of occurrence of a certain event?

- A. Normal distribution
- B. Poisson distribution
- C. Beta distribution
- D. Gamma distribution

Question 3:

Which one of the following credible intervals for a specific parameter of a distribution would have minimum width?

- A. 99%
- B. 90%
- C. 95%
- D. 100%

Question 4:

Suppose we have been given the following credible intervals.
Which one would be better?

- A. 95% credible interval with the range [2, 4]
- B. 90% credible interval with the range [2, 4]
- C. 95% credible interval with the range [3, 3.5]
- D. 90% credible interval with the range [3, 3.5]

Question 5:

The classification is the problem of estimating the _____ variable when it is _____.

- A. output target, discrete.
- B. output target, continuous.
- C. input target, discrete.
- D. input target, continuous.

Question 6:

Log likelihood is preferred instead of simple likelihood because of?

- A. Avoiding very high values of the probability
- B. Avoiding very small values of the probability
- C. Avoiding the computation of the evidence
- D. Sampling of the posterior probability

Question 7:

Maximum Likelihood Estimation (MLE) is used for _____ whereas Maximum a Posteriori (MAP) estimation is used for _____ inference?

- A. Bayesian, Frequentist
- B. Frequentist, Bayesian
- C. Prior, Posterior
- D. Posterior, Prior

Question 8:

Monte Carlo methods are used to _____ the _____ distribution?

- A. Randomize, Likelihood
- B. Randomize, Posterior
- C. Sample, Posterior
- D. Sample, Likelihood

Question 9:

Suppose we have 10 students, out of which 7 are female and 3 are male. We define the following events:

1. Event A: Selecting a female student for a sports event;
2. Event B: Selecting a male student for a sports event.

What is the probability of selecting a female and a male student together?

Hint: find the probability of the happening of events A and B together.

- A. 21/90
- B. 70/90
- C. 7/10
- D. 21/100

Question 10:

Suppose we have 10 students, out of which 7 are female and 3 are male. We define the following events:

1. Event A: Selecting a female student for a sports event;
2. Event B: Selecting a male student for a sports event.

What is the probability of selecting a female student if we have already selected a male student?

Hint: find the probability of event A, given B has already occurred?

- A. 21/90
- B. 70/90
- C. 7/10
- D. 3/9

9

Hands-on Projects

This chapter presents two projects to give the reader a better understanding of the concepts introduced in previous chapters.

The first project performs A/B testing, which is a statistical experiment where we test two or more alternative choices against each other to decide the one that performs better than the other. It is a classical hypothesis testing problem where we state the null and the alternative hypothesis. Confidence intervals are analyzed to reject or accept the null hypothesis in favor of or against the alternative hypothesis, respectively.

The second project deals with linear regression using both frequentist and Bayesian methods. For frequentist inference of linear regression parameters, we use ordinary least squares (OLS), whereas, for Bayesian inference, we make use of Monte Carlo simulation to sample posterior probabilities and use credible intervals to make decisions.

We give the details of these projects in the following sections.

9.1 Project 1: A/B Testing Hypothesis – Frequentist Inference

A/B testing is used to test changes in features of a product or its new features. We split the product users into two categories:

- **experiment or test group** that consists of those users, which are introduced to the new features of the product;
- **control group** that uses the product without new features.

We conduct a comparison between these two groups to decide whether new features of the product are better than the original old features.

A/B testing allows us to select the better option based on the statistical tests. A/B testing may not be appropriate to test major changes such as completely new products. To design an A/B test, we may need to look at the external historical data to get insights about similar experiments.

Suitable metrics should be chosen for A/B testing. These metrics include sums and counts, e.g., how many users visit a particular webpage daily. Other metrics include probability or ratio of probabilities, mean, median, and quartiles, etc. For instance, what are the average or mean number of users of a particular product or its features. Another example may be the percentage of visitors to a web page who follow the links to other specific websites.

In the first step, we decide on a particular metric, for example, the average number of active users who visit an online product or technology. Here, active may be defined as a user who visits the web page at least twice a week. The next step is to decide on the minimum number of observations to conduct the A/B

testing. The following test parameters should be kept in mind to enable the calculation of a suitable sample size:

- **Baseline conversion rate** is the current conversion rate for the online web page we are testing. The conversion rate can be defined as the number of conversions (to premium content, for example) divided by the total number of visitors. This rate corresponds to the time before any change is made in the product or web page.
- **Significance level** is the minimum change to the baseline rate. For example, an increase in the conversion rate of 2 percent might be good for the business.
- **Confidence level** is a range of reasonable values in which we expect to have the true population parameter.

The baseline conversion rate is estimated from the historical data. The significance level depends upon the needs of the business, whereas the confidence level is commonly set at 95 percent.

Once we finalize the aforementioned steps, it is time to implement A/B testing in Python. We download a publicly available dataset from *Kaggle* that is used to test the conversion rates of control and treatment (test) groups related to the old and the new web pages, respectively. The dataset can be downloaded from <https://www.kaggle.com/zhangluyuan/ab-testing>.

This dataset contains 294,478 rows of observations. We define the null and the alternative hypotheses as follows:

$$\begin{aligned}H_0 &: p = p_0 \\H_a &: p \neq p_0\end{aligned}$$

where p_0 and p are the conversion rates of the old and the new designs, respectively. The null hypothesis H_0 can be interpreted as the probabilities of conversion in the control and the test groups are equal. The alternative hypothesis H_a states that the probabilities of conversion in the control and the test groups are not equal.

We set a confidence level of 95 percent, i.e.,

$$\alpha = (1 - 0.95) = 0.05.$$

The user-defined threshold α implies that if the p-value is lower than α , we reject the null hypothesis.

In other words, the conversion rate we observe for the test group should be statistically different from the conversion rate of the control group in such a way that our confidence is 95 percent. If it is the case, we decide to reject the null hypothesis H_0 , i.e., we accept the new design.

Effect size is a measure to evaluate the strength of a statistical claim. It usually refers to a statistic calculated from sample data, such as the correlation between two variables and the mean difference. Effect sizes are used in statistical hypothesis testing to calculate the power of a statistical test. The [standard deviation](#) of the effect size indicates the amount of uncertainty in the measurement.

The effect size is directly related to the difference we expect in conversion rates between the control and the treatment groups. In this project, we set the effect size to 2.5 percent. Thus, we compute the effect size using a baseline conversion rate of, for example, 13 percent and an expected conversion rate of 15.5 percent in the treatment group ($15.5 - 13 = 2.5\%$).

The minimum sample size required for the A/B testing is estimated using **the power of the test**. Recall from Section 6.5.2 that the power of the test is defined as $(1-\beta)$. This corresponds to the probability of discovering a statistical difference between the control and the treatment groups when the difference actually exists. A typical value of this is generally set at 0.8. The value of Alpha α is set earlier at 0.05 from a 95 percent confidence interval.

After importing the necessary packages and libraries, the following Python code computes the effect size, and then the sample size required using the power of the test.

```
1. # Importing Packages and Libraries
2. import numpy as np
3. import pandas as pd
4. import scipy.stats as stats
5. import statsmodels.stats.api as sms
6. import matplotlib.pyplot as plt
7. import seaborn as sns
8. %matplotlib inline
9.
10. # Calculating effect size based on our expected rates
11. effect_size = sms.proportion_effectsize(0.13, 0.155)
12. # Calculation of the required sample size
13. required_sample_size = sms.NormalIndPower().solve_
    power(effect_size, alpha=0.05, power=0.8, ratio=1)
14. required_sample_size = np.ceil(required_sample_size)
15. print(required_sample_size)
Output:
3064.0
```

The function `proportion_effectsize()` from the `Statsmodel.stats.api` specified in line 11 of the code takes two conversion rates as the input and computes the effect size to be used in the power of the test. The function `NormalIndPower().solve_`

power () takes any three of the following parameters as its input and computes the fourth parameter:

- **Effect_size** is related to the difference we expect in conversion rates between the control and the treatment groups.
- **Alpha (α)** is the probability of a type I error, that is, wrong rejections if the Null Hypothesis is true.
- **Nobs1** is the number of observations of the sample.
- **Power of the test ($1-\beta$)** is one minus the probability of a type II error. It is the probability that the test correctly rejects the null hypothesis if the alternative hypothesis is true.
- **Ratio** is the number of observations in sample 2 relative to sample 1. It is usually set to 1.

Here, we intend to find the number of observations of sample 1. The output of the program indicates that at least 3,064 observations of each group are required.

The aforementioned settings of the parameters specify that with a power parameter set to 0.8, if there is an actual difference in two conversion rates, i.e., 2.5% (15.5%–13%), we have about 80% chance to detect it as statistically significant in our test with the calculated sample size.

We load our downloaded dataset as a Pandas DataFrame as follows:

```
1. df = pd.read_csv('ab_data.csv')
2. df.info()
3. df.head(10)
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 294478 entries, 0 to 294477
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
---  --          -----          ---    
 0   user_id     294478 non-null   int64  
 1   timestamp   294478 non-null   object  
 2   group       294478 non-null   object  
 3   landing_page 294478 non-null   object  
 4   converted    294478 non-null   int64  
dtypes: int64(2), object(3)
memory usage: 11.2+ MB
```

	user_id	timestamp	group	landing_page	converted
0	851104	11:48.6	control	old_page	0
1	804228	01:45.2	control	old_page	0
2	661590	55:06.2	treatment	new_page	0
3	853541	28:03.1	treatment	new_page	0
4	864975	52:26.2	control	old_page	1
5	936923	20:49.1	control	old_page	0
6	679687	26:46.9	treatment	new_page	1
7	719014	48:29.5	control	old_page	0
8	817355	58:09.0	treatment	new_page	1
9	839785	11:06.6	treatment	new_page	1

The first line reads our dataset, which is saved as a comma separated value file (.csv extension). We check the information of the loaded dataset in the second line. Finally, the third line of the code shows the first 10 row entries (each row is one observation) of the dataset. It can be observed that there are a total of 294,478 observations, each having five columns or

feature variables. The columns **user_id** and the **timestamp** show the user ID of each session and the time of the session, respectively.

The column **group** shows two different types of user categories: the user assigned to the control or the treatment group. The **landing_page** column shows which design of the web page, old or new, the user landed on. The column named **converted** shows the status of the users, whether a conversion is made or not where 0=not converted and 1=converted.

We check the cross-tabulation of the control and the treatment groups by Pandas crosstab () function.

```
1. # Computing cross-tabulation or frequency table of groups.
2. pd.crosstab(df['group'], df['landing_page'])
```

Output:

		landing_page	new_page	old_page
		group		
		control	1928	145274
		treatment	145311	1965

It can be observed that some users in the control group are incorrectly exposed to the new_page, and the users of the treatment group are incorrectly exposed to the old_page.

We check if some users appear multiple times to check duplicate entries in the dataset.

```
1. # Checking users who appear multiple times in the dataset.
2. num_sessions = df['user_id'].value_counts()
3. user_mul_times = num_sessions[num_sessions > 1].count()
4.
5. print('There are %s users that appear multiple times in
       the dataset' %user_mul_times)
```

Output:

There are 3894 users that appear multiple times in the dataset.

We get rid of these duplicate entries to avoid sampling of the same user multiple times.

```
1. remove_users = num_sessions[num_sessions > 1].index  
2.  
3. df = df[~df['user_id'].isin(remove_users)]  
4. print(f'''The updated dataset has {df.shape[0]}  
observations  
5.           after we have removed the users who appear multiple  
times.'''')
```

Output:

```
The updated dataset has 286,690 observations after we have  
removed the users who appear multiple times.
```

The first line gets the indices of the users who appear in multiple sessions. These indices are used in the isin () function that returns the values in df['user_id'], which are in the given list, and the ~ at the beginning is a not operator.

df['user_id'].isin(remove_users) returns those indices which corresponds to the IDs to be removed. However, ~ df['user_id'].isin(remove_users) is used to find those indices which corresponds to the IDs to keep. Thus, we get 286,690 observations after we have removed the users that appear multiple times in the dataset.

Since we have already calculated the required sample size to be used for A/B testing, we do not need to work with all the observations of the dataset. We sample both the control and the treatment groups for the required sample size in the following Python script.

```
1. control_sample = df[df['group'] == 'control'].  
    sample(n=int(required_sample_size), random_state=5)  
2. treatment_sample = df[df['group'] == 'treatment'].  
    sample(n=int(required_sample_size), random_state=5)  
3.  
4. ab_test = pd.concat([control_sample, treatment_sample],  
    axis=0)  
5. ab_test.reset_index(drop=True, inplace=True)  
6.  
7.  
8. ab_test.info()  
9. print('\n \n')  
10. ab_test['group'].value_counts()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 6128 entries, 0 to 6127  
Data columns (total 5 columns):  
 #   Column      Non-Null Count  Dtype     
 ---  --          --          --          --  
 0   user_id     6128 non-null   int64    
 1   timestamp   6128 non-null   object   
 2   group       6128 non-null   object   
 3   landing_page 6128 non-null   object   
 4   converted    6128 non-null   int64    
dtypes: int64(2), object(3)  
memory usage: 239.5+ KB
```

```
treatment    3064  
control      3064  
Name: group, dtype: int64
```

In lines 2 and 3, we select the observations corresponding to the control and the treatment groups, respectively. The function `sample()` is used to take a random sample equal to the size of the variable `required_sample_size`. Note that we have converted this variable to an integer because the `sample()` requires an integer input rather than a float.

In lines 4 and 5, we first concatenate both groups as a single DataFrame namely ab_test, then the indices of the DataFrame are reset. This is done in place to update the DataFrame without creating a new dummy object. The output shows that there are a total of 6,128 entries out of which half the entries are for the control, and the remaining half is for the treatment group.

Once we are done with the sampling, we visualize the results, i.e., the conversion rates for both groups.

```

1. conversion_rates = ab_test.groupby('group')[‘converted’]
2. # Std. deviation of the proportion
3. std_p = lambda x: np.std(x, ddof=0)
4. # Std. error of the proportion
5. se_p = lambda x: stats.sem(x, ddof=0)
6.
7. conversion_rates = conversion_rates.agg([np.mean, std_p,
   se_p])
8.
9. conversion_rates.columns = [‘conversion_rate’, ‘std_
   deviation’, ‘std_error’]
10.
11. conversion_rates.style.format(‘{:.4f}’)
Output:
```

	conversion_rate	std_deviation	std_error
group			
control	0.1237	0.3292	0.0059
treatment	0.1270	0.3329	0.0060

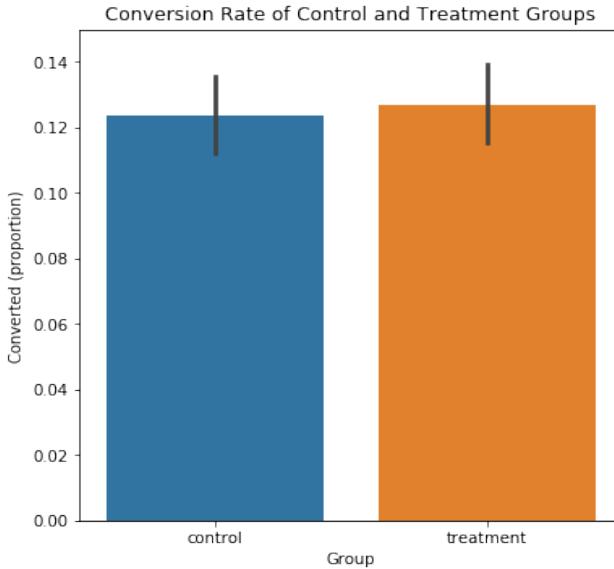
We compute the standard deviation and the standard error [σ/\sqrt{n}] of both groups. Furthermore, these results are aggregated based on the mean, standard deviation, and the standard error in line 7 of the code.

These results show that the performance of the two designs of the web pages is quite similar—12.37 percent and 12.7 percent

conversion rates for the control and the treatment groups, respectively. We plot these results.

```
1. # plotting the results
2. plt.figure(figsize=(6,6))
3. sns.barplot(x=ab_test['group'], y=ab_test['converted'])
4.
5. plt.ylim(0, 0.15)
6. plt.title('Conversion Rate of Control and Treatment
Groups')
7. plt.xlabel('Group')
8. plt.ylabel('Converted (proportion)')
```

Output:



The black vertical line in both bars represents the 95 percent confidence interval. The treatment group has a slightly greater conversion rate than the control group. We check if this small increment is statistically significant.

Finally, we test our hypothesis. We use a Normal approximation to compute p-value in a z-test. We import statsmodels.stats.proportion module to get the p-value and confidence intervals.

We also find the number of conversions in both control and the treatment groups as follows.

```

1. from statsmodels.stats.proportion import proportions_
   ztest, proportion_confint
2. converted_control = ab_test[ab_test['group'] == 'control']
   ['converted']
3. converted_treatment = ab_test[ab_test['group'] ==
   'treatment']['converted']
4.
5.
6. n_control = converted_control.count()
7. n_treatment = converted_treatment.count()
8. successes = [converted_control.sum(), converted_treatment.
   sum()]
9. nobs = [n_control, n_treatment]
10.
11. z_stat, pval = proportions_ztest(successes, nobs=nobs)
12. (lower_con, lower_treat), (upper_con, upper_treat) =
   proportion_confint(successes, nobs=nobs, alpha=0.05)
13.
14. print(f'z statistic: {z_stat:.2f}')
15. print(f'p-value: {pval:.3f}')
16. print(f'95% confidence interval for control group: [{lower_\
   con:.3f}, {upper_con:.3f}]')
17. print(f'95% confidence interval for treatment group: \
   [{lower_treat:.3f}, {upper_treat:.3f}]')

Output:
z statistic: -0.39
p-value: 0.700
95% confidence interval for control group: [0.112, 0.135]
95% confidence interval for treatment group: [0.115, 0.139]
```

In lines 2 and 3, we separate those control and treatment observations, which result in conversion from the DataFrame ab_test. Lines 6 and 7 of the code compute the number of conversions for both groups. Line 11 of the code computes the z-statistic and the p-value from the successes. Line 12 uses

function `proportion_confint()` to get the **confidence intervals for a binomial proportion**.

The results show that the p -value = 0.700 is greater than the set significance level $\alpha = 0.05$. This implies that the probability of observing extreme results is small. Thus, we shall not reject the null hypothesis H_0 . In conclusion, the new web page (new design) does not perform significantly better than the old design.

The limits in a 95 percent confidence interval for the treatment group ($[0.115, 0.139]$) contains the baseline 13 percent conversion rate. However, it does not include the 15.5 percent target. Thus, the new design is unable to meet our expectations.

9.2 Project 2: Linear Regression using Frequentist and Bayesian Approaches

There are mainly two types of supervised learning algorithms: classification and regression. If the output labels have a continuous range of values, it is a regression problem. For example, the prediction of house prices from a given dataset is a regression problem.

If the relationship between the input features and the output target variable is assumed to be linear, the regression would be linear. Otherwise, it would be non-linear. Here, we discuss linear regression that is more prevalent than its non-linear counterpart.

The key objective of a linear regression model is to find a relationship between one or more independent input features and a continuous target variable, which is dependent upon the input features. When there is only one feature, it is called

a univariate or simple linear regression problem, whereas the case of multiple input features is known as **multiple linear regression**. The following equation illustrates the linear regression model:

$$f(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_n x_n + \varepsilon$$

where $f(x)$ is the predicted value, θ_0 is the bias term, $[\theta_1 \theta_2 \theta_3 \dots]$ are model parameters, $[x_0 \ x_1 x_2 x_3 \dots x_n]$ are $(n+1)$ input features, $x_0 = 1$, and ε represents the noise in the measurements. This regression model can be compactly represented as:

$$f(x) = \Theta^T \mathbf{x} + \varepsilon$$

where $\Theta^T = [\theta_0 \theta_1 \theta_2 \theta_3 \dots]$ is the vector that contains all the parameters of the model and $\mathbf{x} = [x_0 x_1 x_2 x_3 \dots]$ is the vector of features. The learned function $f(x)$ serves as an estimate of the output target variable y . The solution to the linear regression problem can be given as:

$$\Theta = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T \mathbf{y}$$

This solution is obtained using a mathematical technique known as Ordinary Least Squares (OLS) that gives us the maximum likelihood estimate of the parameter vector Θ .

If we have one feature, the parameters θ_0 and θ_1 correspond to the y-intercept and the slope of the line. However, in the case of more than one input feature, as in the aforementioned model, the concept of a line is extended to a plane or a hyperplane in more than two dimensions.

To implement the linear regression in Python, we first load our dataset in a Pandas DataFrame.

```

1. # Importing packages
2. import pandas as pd
3. import numpy as np
4. import matplotlib.pyplot as plt
5. %matplotlib inline
6. import seaborn as sns
7. import scipy           # Scipy for statistics
8. # PyMC3 for Bayesian Inference
9. import pymc3 as pm
10.
11. # Loading and displaying datasets
12. exercise = pd.read_csv('exercise.csv')      # give path of
   the dataset files
13. calories = pd.read_csv('calories.csv')
14. df = pd.merge(exercise, calories, on = 'User_ID')
15. df = df[df['Calories'] < 300]
16. df = df.reset_index()
17. df['Intercept'] = 1
18. df.head(10)

```

Output:

index	User_ID	Gender	Age	Height	Weight	Duration	Heart_Rate	Body_Temp	Calories	Intercept
0	0	14733363	male	68	190.0	94.0	29.0	105.0	40.8	231.0
1	1	14861698	female	20	166.0	60.0	14.0	94.0	40.3	66.0
2	2	11179863	male	69	179.0	79.0	5.0	88.0	38.7	26.0
3	3	16180408	female	34	179.0	71.0	13.0	100.0	40.5	71.0
4	4	17771927	female	27	154.0	58.0	10.0	81.0	39.8	35.0
5	5	15130815	female	36	151.0	50.0	23.0	96.0	40.7	123.0
6	6	19602372	female	33	158.0	56.0	22.0	95.0	40.5	112.0
7	7	11117088	male	41	175.0	85.0	25.0	100.0	40.7	143.0
8	8	12132339	male	60	186.0	94.0	21.0	97.0	40.4	134.0
9	9	17964668	female	26	146.0	51.0	16.0	90.0	40.2	72.0

After importing the required packages, we load the already downloaded datasets **exercise.csv** and **calories.csv**. These two datasets can be downloaded from <https://www.kaggle.com/fmendes/exercise-and-calories>.

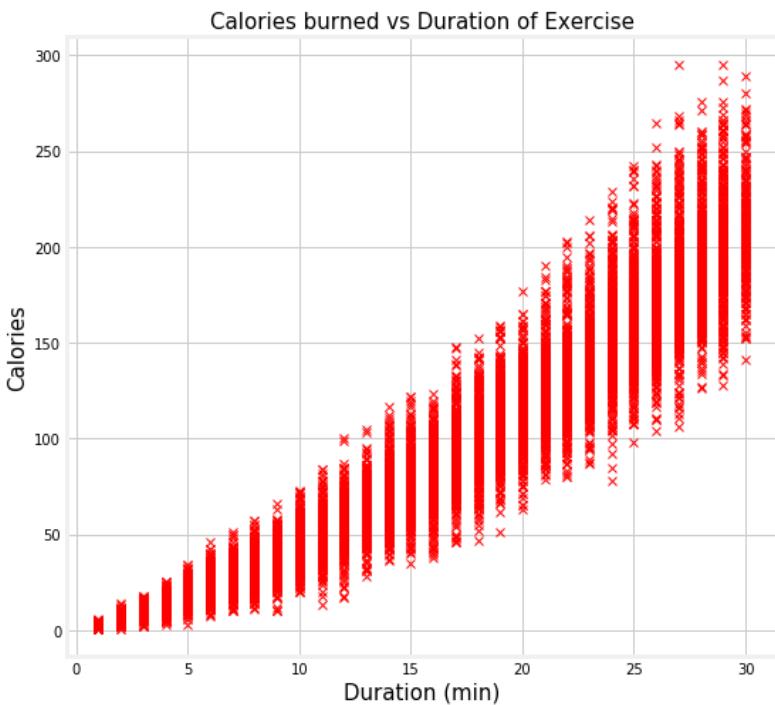
The **exercise** dataset has the following 8 columns: User_ID, Gender, Age, Height, Weight, Duration, Heart_Rate, and Body_

Temp, whereas the **calorie** dataset contains only 2 columns: User_ID and Calories. We merge these two datasets together in line 14 of the code to find the relationship between the duration of the exercise and the calories burnt.

The following code plots the calories burnt against the time spent in exercise.

```
1. plt.figure(figsize=(8, 8))
2.
3. plt.plot(df['Duration'], df['Calories'], 'rx');
4. plt.xlabel('Duration (min)', size = 15); plt.
   ylabel('Calories', size = 15);
5. plt.title('Calories burned vs Duration of Exercise', size
   = 15);
```

Output:



Each red color x mark on this graph shows one data point (observation) in the combined dataset. The time is measured

in integer values of minutes. Therefore, we do not see a continuous plot here. Instead, we observe discrete points on the plot corresponding to integer values of the duration of exercise in minutes.

9.2.1 Frequentist Approach

We create X features and y output target variable, as follows:

```
1. # Create the features and response
2. X = df.loc[:, ['Intercept', 'Duration']]
3. y = df.loc[:, 'Calories']
4. X.head(8)
```

Output:

	Intercept	Duration
0	1	29.0
1	1	14.0
2	1	5.0
3	1	13.0
4	1	10.0
5	1	23.0
6	1	22.0
7	1	25.0

Note that an intercept = 1 is required in X because of the presence of the term $\theta_0 x_0$. The intercept corresponds to x_0 in our model.

We find the OLS solution to the linear regression model between feature duration and the response calories burnt as follows.

```

1. # Ordinary Least Squares calculations
2. ols_coefs = np.matmul(np.matmul(
   inv(X.T.dot(X)) , X.T), y)
3. print(f'Intercept calculated: {ols_coefs [0]}')
4. print(f'Slope calculated: {ols_coefs[1]}')

Output:
Intercept calculated: -21.828102526050735
Slope calculated: 7.16978334958786

```

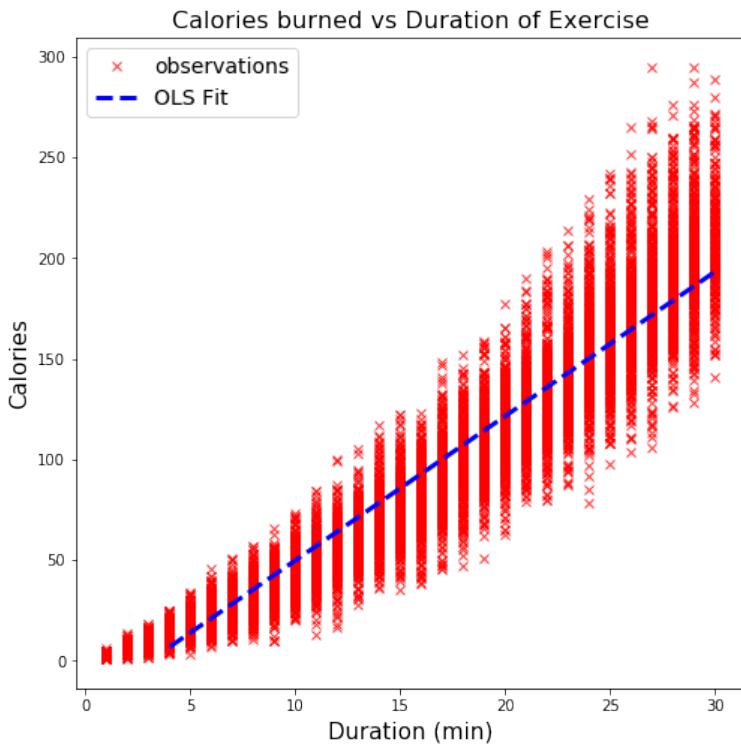
X.T in line 2 of the code computes the transpose of the matrix, X.T.dot(X) computes the term $X^T X$, the NumPy linear algebra function np.linalg.inv () computes the matrix inverse, and np.matmul () performs matrix multiplication. The output shows two estimated parameters: the intercept (θ_0) and the slope (θ_1).

We plot the estimated parameters, along with the data points, as follows.

```

1. xs = np.linspace(4, 30, 1000)
2. ys = ols_coefs[0] + ols_coefs[1] * xs
3.
4. plt.figure(figsize=(8, 8))
5.
6. plt.plot(df['Duration'], df['Calories'], 'rx', label =
   'observations', alpha = 0.8)
7. plt.xlabel('Duration (min)', size = 15); plt.
   ylabel('Calories', size = 15)
8. plt.plot(xs, ys, 'b--', label = 'OLS Fit', linewidth = 3)
9. plt.legend(prop={'size': 14})
10. plt.title('Calories burned vs Duration of Exercise', size
   = 16);

```

Output:

The blue color line that passes almost in the middle through the observed data points is the OLS fit to our data. Note that a line in a 2-dimensional space is described by two parameters as in the simple linear regression model. Had we used multiple input features, we would have to estimate more than two parameters of the linear regression model in higher dimensions.

Now to check the estimated output (calories burnt) against a specific input point (duration), we type the following code:

```

1. #specifying a point against which the OLS estimate is to
   be calculated.
2. specific_point = 20
3.
4. OLS_estimate = ols_coefs[0] + ols_coefs[1] * specific_point
5.
6. print('Exercising for {:.0f} minutes will burn an
   estimated {:.2f} calories.')
7.     format(specific_point, OLS_estimate))
Output:
Exercising for 20 minutes will burn an estimated 121.57
calories.

```

Line 4 of the code uses a specific_point as input to the estimated model. The OLS estimated against a 20-minute duration is found to be 121.57 calories. This is the way we perform testing once we have learned the parameters of our model. Every future test point is passed through an equation given in line 4 of the code to find its estimated output target variable.

9.2.2 Bayesian Approach

The aforementioned linear regression model is computed using the **frequentist** approach. The OLS gave us the maximum likelihood estimate (MLE) of the model parameters. To find the Bayesian solution, we have to specify a prior along with the likelihood function.

We resort to the PyMC module for specifying parameters of a Bayesian model. Let's suppose we have access to only a few observations. Let the number of observations be 500. The following code specifies a Bayesian linear regression model with 500 observations using PyMC.

```

1. with pm.Model() as linear_model_500:
2.     # Intercept modeled as Normally distributed
3.     intercept = pm.Normal('Intercept', mu = 0, sd = 10)
4.     # Slope modeled as Normally distributed
5.     slope = pm.Normal('slope', mu = 0, sd = 10)
6.     # Standard deviation modeled as half Normally
    distributed
7.     sigma = pm.HalfNormal('sigma', sd = 10)
8.
9.     # Estimate of mean
10.    mean = intercept + slope * X.loc[0:499, 'Duration']
11.
12.    # Observed values
13.    Y_obs = pm.Normal('Y_obs', mu = mean, sd = sigma,
    observed = y.values[0:500])
14.
15.    # Sampler
16.    step = pm.NUTS()
17.
18.    # Posterior distribution
19.    linear_trace_500 = pm.sample(1000, step)

Output:

```

```

Multiprocess sampling (2 chains in 2 jobs)
NUTS: [sigma, slope, Intercept]

100.00% [4000/4000 00:23<00:00 Sampling 2 chains, 0
    divergences]
Sampling 2 chains for 1_000 tune and 1_000 draw iterations
    (2_000 + 2_000 draws total) took 71 seconds.
The acceptance probability does not match the target. It
    is 0.886405005570454 but should be close to 0.8. Try to
    increase the number of tuning steps.

```

In Bayesian inference, the parameters of the model are not considered fixed; thus, we model both parameters intercept and the slope as Normally distributed with a zero mean and standard deviation 10. The standard deviation can only be positive. Thus, it is modeled as a half Normal having only

positive values. Line 10 estimates the mean value of the target variable from the linear regression model using the intercept and the slope defined in lines 3 and 5, respectively.

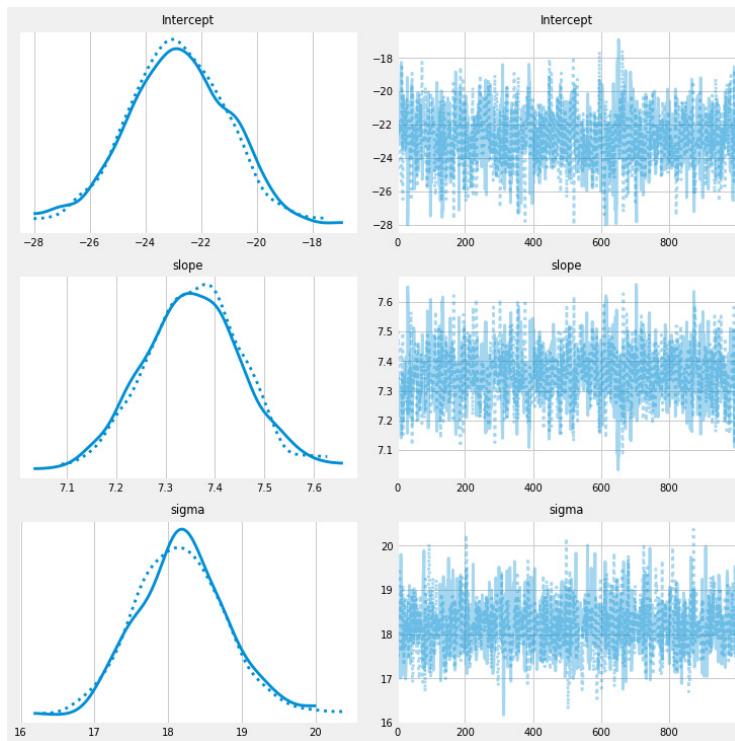
Line 13 of the code models the observed variable as a Normal with the mean estimated in line 10, and the standard deviation modeled as half Normal in line 7 of the code. It uses 500 observed values of the output variable.

Line 19 of the code takes 1,000 samples of the posterior distribution using a built-in No U-Turn Sampler (NUTS) that is specified as an option to the function sample () .

The sample posterior is known as trace in PyMC. We plot the trace as follows:

```
1. pm.traceplot(linear_trace_500, figsize = (10, 10))
```

Output:

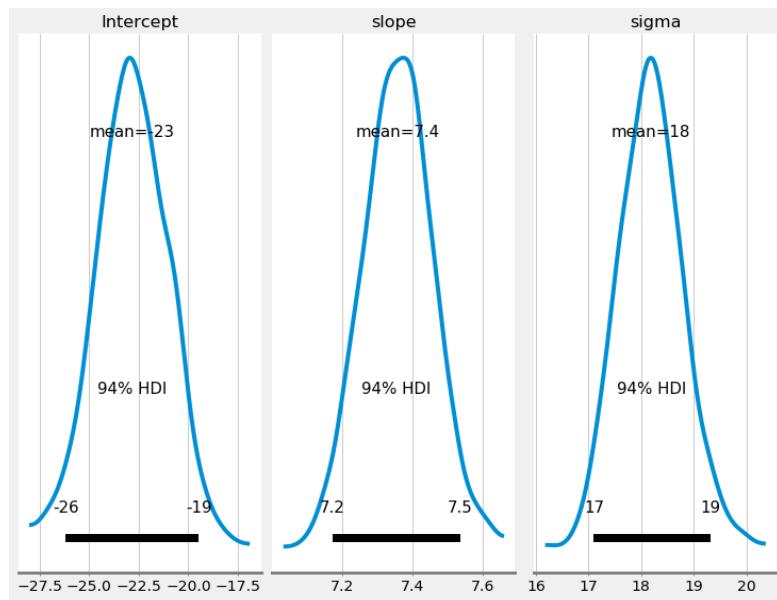


Three plots in the output correspond to the intercept, slope, and standard deviation. The right-hand sides of the figures show samples, whereas the left sides show the histogram of these samples. It can be seen in the output plots given on the left side of the figure that all three posteriors are Normally distributed.

Furthermore, the peaks of these distributions correspond to the mode of the distribution, i.e., the Maximum A Posteriori (MAP) estimates of the parameters of the model. For instance, the slope is peaked between 7.3 and 7.4.

To get a better idea about the posteriors, we plot the posterior distributions as follows:

```
1. pm.plot_posterior(linear_trace_500, figsize = (10, 8))
Output:
```



The three plots show the mean values of the estimated posterior distributions along with the default 94 percent credible interval.

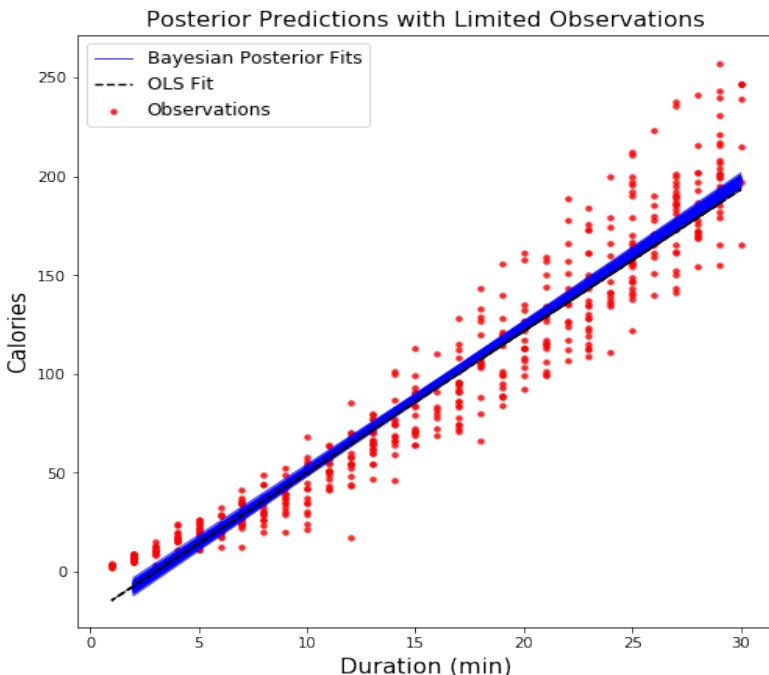
We plot these estimates along with the frequentist estimates on the plot of the original dataset.

```

1. plt.figure(figsize = (8, 8))
2. pm.plot_posterior_predictive_glm(linear_trace_500, samples
   = 100, eval=np.linspace(2, 30, 100), linewidth = 1,
   color = 'blue', alpha =
   0.8, label = 'Bayesian Posterior Fits',
4.           lm = lambda x, sample:
   sample['Intercept'] + sample['slope'] * x)
5. plt.scatter(X['Duration'][:500], y.values[:500], s = 12,
   alpha = 0.8, c = 'red', label = 'Observations')
6. plt.plot(X['Duration'], ols_coefs[0] + X['Duration'] *
   ols_coefs[1], 'k--', label = 'OLS Fit', linewidth = 1.4)
7. plt.title('Posterior Predictions with Limited
   Observations', size = 15); plt.xlabel('Duration (min)',
   size = 15)
8. plt.ylabel('Calories', size = 15)
9. plt.legend(prop={‘size’}: 12)

```

Output:



To evaluate the Bayesian posterior fit, we plot posterior predictive regression lines by taking the regression parameters from the posterior distribution and plotting a regression line for each. The function `plot_posterior_predictive_glm()` plots the predicted posterior lines of a Generalized Linear Model (GLM).

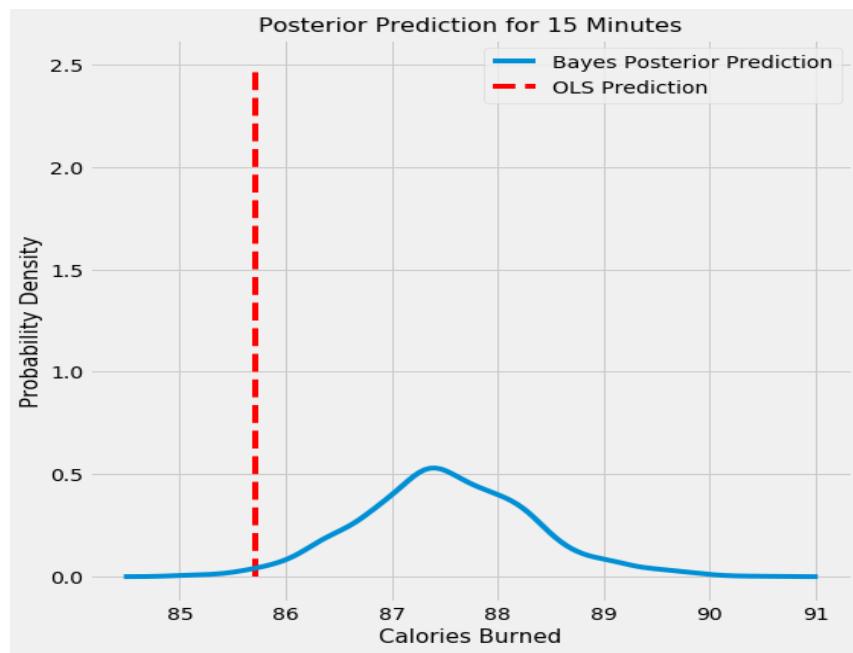
The black line corresponds to the OLS fit, whereas the blue lines correspond to the Bayesian posterior fit lines. It can be seen that the Bayesian estimate of the parameters correctly corresponds to the frequentist estimates.

To check the posterior distribution of the target variable, `calories`, against a specified input point, say 20-minute duration, we use the following Python script:

```

1. specific_point = 20    # this corresponds to the value in
   terms of minutes exercised
2. bayes_prediction = linear_trace_500['Intercept'] + linear_
   trace_500['slope'] * specific_point
3.
4.
5. # plotting the results
6. plt.figure(figsize = (8, 8))
7. plt.style.use('fivethirtyeight')
8. sns.kdeplot(bayes_prediction, label = 'Bayes Posterior
   Prediction')
9. plt.vlines(x = ols_coefs[0] + ols_coefs[1] * specific_
   point,
10.             ymin = 0, ymax = 2.5,
11.             label = 'OLS Prediction',
12.             colors = 'red', linestyles='--')
13. plt.legend();
14. plt.xlabel('Calories Burned', size = 15), plt.
   ylabel('Probability Density', size = 15);
15. plt.title(f'Posterior Prediction for {specific_point}
   Minutes', size = 16)

```

Output:

The output plot shows the Bayesian posterior prediction as a Normal curve along with the frequentist estimate for comparison. The peak of the Bayesian prediction is a bit far from the frequentist estimate. This is because we have used 500 samples instead of 1,000. The posterior distribution by the Bayesian inference reports the spread (standard deviation) of the target variable to indicate the variability of our estimates. This is a feature not offered by the frequentist estimates because the estimated mean is considered fixed and reported as a single point in the frequentist inference.

Answers to Exercise Questions

Chapter 1

- Question 1: A
- Question 2: C
- Question 3: B
- Question 4: A
- Question 5: B
- Question 6: C
- Question 7: B
- Question 8: C
- Question 9: C
- Question 10: C

Chapter 2

- Question 1: B
- Question 2: A
- Question 3: C
- Question 4: C
- Question 5: C
- Question 6: D

Question 7: B

Question 8: C

Question 9: D

Question 10: D

Chapter 3

Question 1: C

Question 2: A

Question 3: B

Question 4: B

Question 5: C

Question 6: C

Question 7: A

Question 8: C

Question 9: B

Question 10: B

Question 11: B

Question 12: B

Chapter 4

Question 1: D

Question 2: B

Question 3: B

Question 4: C

Question 5: C

Question 6: D

Question 7: C

Question 8: A

Chapter 5

- Question 1: C
- Question 2: D
- Question 3: B
- Question 4: A
- Question 5: D
- Question 6: B
- Question 7: D
- Question 8: C

Chapter 6

- Question 1: B
- Question 2: A
- Question 3: B
- Question 4: C
- Question 5: B
- Question 6: D
- Question 7: A

Chapter 7

- Question 1: A
- Question 2: B
- Question 3: A
- Question 4: A
- Question 5: A
- Question 6: C

Chapter 8

Question 1: C

Question 2: D

Question 3: B

Question 4: C

Question 5: A

Question 6: B

Question 7: B

Question 8: C

Question 9: A

Question 10: B

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.