Hands-On Simulation Modeling with Python

Develop simulation models to get accurate results and enhance decision-making processes

Packt

www.packt.com

Giuseppe Ciaburro

Hands-On Simulation Modeling with Python

Develop simulation models to get accurate results and enhance decision-making processes

Giuseppe Ciaburro

Packt

BIRMINGHAM—MUMBAI

Hands-On Simulation Modeling with Python

Commissioning Editor: Sunith Shetty
Acquisition Editor: Devika Battike
Senior Editor: David Sugarman
Content Development Editor: Joseph Sunil
Technical Editor: Sonam Pandey
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Joshua Misquitta

First published: July 2020

Production reference: 1160720

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83898-509-7

`www.packt.com`

Contributors

About the author

Giuseppe Ciaburro holds a PhD in environmental technical physics, along with two master's degrees. His research was focused on machine learning applications in the study of urban sound environments. He works at the Built Environment Control Laboratory at the Università degli Studi della Campania Luigi Vanvitelli, Italy. He has over 18 years' professional experience in programming (Python, R, and MATLAB), first in the field of combustion, and then in acoustics and noise control. He has several publications to his credit.

About the reviewers

Greg Walters has been involved with computers and computer programming since 1972. He is well versed in Visual Basic, Visual Basic.NET, Python, and SQL, and is an accomplished user of MySQL, SQLite, Microsoft SQL Server, Oracle, C++, Delphi, Modula-2, Pascal, C, 80x86 Assembler, COBOL, and Fortran. He is a programming trainer and has trained numerous individuals in many pieces of computer software, including MySQL, Open Database Connectivity, Quattro Pro, Corel Draw!, Paradox, Microsoft Word, Excel, DOS, Windows 3.11, Windows for Workgroups, Windows 95, Windows NT, Windows 2000, Windows XP, and Linux. He is currently retired and, in his spare time, is a musician and loves to cook. He is also open to working as a freelancer on various projects.

Yoon Hyup Hwang is a seasoned data scientist in the marketing and finance industries and is the author of two applied machine learning books. He has almost a decade of experience building numerous machine learning models and data science products. He holds an M.S.E. in Computer and Information Technology from the University of Pennsylvania and a B.A. in Economics from the University of Chicago. He enjoys training various martial arts, snowboarding, and roasting coffee. He currently lives in the Greater New York Area with his artist wife, Sunyoung, and a playful dog, Dali (named after Salvador Dali).

Table of Contents

4

Exploring Monte Carlo Simulations

5

Simulation-Based Markov Decision Processes

Section 3: Real-World Applications

Preface

Simulation modeling helps you to create digital prototypes of physical models to analyze how they work and predict their performance in the real world. With this comprehensive guide, you'll learn about various computational statistical simulations using Python.

Starting with the fundamentals of simulation modeling, you'll learn about concepts such as randomness and explore data generating processes, resampling methods, and bootstrapping techniques. You'll then cover key algorithms such as Monte Carlo simulations and the Markov Decision Process, which are used to develop numerical simulation models, and discover how they can be used to solve real-world problems. As you make progress, you'll develop simulation models to help you get accurate results and enhance decision-making processes. Using optimization techniques, you'll learn to modify the performance of a model to improve results and make optimal use of resources. The book will guide you through creating a digital prototype using practical use cases for financial engineering, prototyping project management to improve planning, and simulating physical phenomena using neural networks.

By the end of this book, you'll be able to construct and deploy simulation models of your own to solve real-world challenges.

Who this book is for

Hands-On Simulation Modeling with Python is for simulation developers and engineers, model designers, and anyone already familiar with the basic computational methods that are used to study the behavior of systems. This book will help you explore advanced simulation techniques such as Monte Carlo methods, statistical simulations, and much more using Python. Working knowledge of the Python programming language is required.

What this book covers

Chapter 1, *Introduction,* analyzes the basics of numerical simulation and highlights the difference between modeling and simulation and the strengths of simulation models such as defects. The different types of models are analyzed, and we study practical modeling cases to understand how to elaborate a model starting from the initial considerations.

Chapter 2, Understanding Randomness and Random Numbers, defines stochastic processes and explains the importance of using them to address numerous real-world problems. The main methods for generating random numbers with practical examples in Python code, and the generation of uniform and generic distributions, are both explored. It also explains how to perform a uniformity test using the chi-square method.

Chapter 3, Probability and the Data Generating Process, shows how to distinguish between the different definitions of probabilities and how they can be integrated to obtain useful information in the simulation of real phenomena.

Chapter 4, Monte Carlo Simulations, explores techniques based on Monte Carlo methods for process simulation. We will first learn the basic concepts, and then we will see how to apply them to practical cases.

Chapter 5, Simulation-Based Markov Decision Process, shows how to deal with decision-making processes with Markov chains. We will analyze the concepts underlying Markovian processes and then analyze some practical applications to learn how to choose the right actions for the transition between different states of the system.

Chapter 6, Resampling Methods, shows how to apply resampling methods to approximate some characteristics of the distribution of a sample in order to validate a statistical model. We will analyze the basics of the most common resampling methods and learn how to use them by solving some practical problems.

Chapter 7, Use of Simulation to Improve and Optimize Systems, shows how to use the main optimization techniques to improve the performance of our simulation models. We will see how to use the gradient descent technique, the Newton-Raphson method, and stochastic gradient descent. We will also see how to apply these techniques with practical examples.

Chapter 8, Simulation Models for Financial Engineering, shows practical cases of using simulation methods in a financial context. We will learn how to use Monte Carlo methods to predict stock prices and how to assess the risk associated with a portfolio of shares.

Chapter 9, Simulating Physical Phenomena with Neural Networks, shows how to develop models based on artificial neural networks to simulate physical phenomena. We will start by exploring the basic concepts of neural networks, and we will examine their architecture and its main elements. We will see how to train a network to update its weights.

Chapter 10, *Modeling and Simulation for Project Management*, deals with practical cases of project management using the tools we learned how to use in the previous chapters. We will see how to evaluate in advance the results of the actions undertaken in the management of a forest using Markov processes, and then move on to evaluating the time required for the execution of a project using the Monte Carlo simulation.

Chapter 11, *What's Next?*, provides a better understanding of the problems associated with building and deploying simulation models and additional resources and technologies to learn how to hone your machine learning skills.

To get the most out of this book

Working knowledge of Python programming language is required.

Software/Hardware covered in the book	OS Requirements
Python 3.6 and above	Windows, Mac OS X, or Linux

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

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781838985097_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
import random
import statistics
import matplotlib.pyplot as plt
```

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

```
import random
import statistics
import matplotlib.pyplot as plt
```

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

```
$ python jakknife_estimator.py
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

> **Tips or important notes**
>
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

Section 1: Getting Started with Numerical Simulation

In this section, the basic concepts of simulation modeling are addressed. This section helps you to understand the fundamental concepts and elements of numerical simulation.

This section contains the following chapters:

1
Introducing Simulation Models

A simulation model is a tool capable of processing information and data and predicting the responses of a real system to certain inputs, thus becoming an effective support for analysis, performance evaluation, and decision-making processes. The term simulation refers to reproducing the behavior of a system. In general, we speak of simulation both in the case in which a concrete model is used and in the case in which an abstract model is used that reproduces reality using a computer. An example of a concrete model is a scale model of an airplane that is then placed in a wind tunnel to carry out simulated tests to estimate suitable performance measures.

Although, over the years, physicists have developed theoretical laws that we can use to obtain information on the performance of dynamic systems, often, the application of these laws to a real case takes too long. In these cases, it is convenient to construct a numerical simulation model that allows us to simulate the behavior of the system under certain conditions. This elaborated model will allow us to test the functionality of the system in a simple and immediate way, saving considerable resources in terms of time and money.

In this chapter, we're going to cover the following main topics:

- Introducing simulation models
- Classifying simulation models
- Approaching a simulation-based problem
- Dynamical systems modeling

Important Note

In this chapter, an introduction to simulation techniques will be discussed. In order to deal with the topics at hand, it is necessary that you have a basic knowledge of algebra and mathematical modeling.

Introducing simulation models

Simulation uses abstract models built to replicate the characteristics of a system. The operation of a system is simulated using probability distributions to randomly generate system events, and statistical observations are obtained from the simulated system. It plays a very important role, especially in the design of a stochastic system and in the definition of its operating procedures.

By not working directly on the real system, many scenarios can be simulated simply by changing the input parameters, thus limiting the costs that would occur if this solution were not used and, ultimately, reducing the time it would take. In this way, it is possible to quickly try alternative policies and design choices and model systems of great complexity by studying their behavior and evolution over time.

Important Note

Simulation is used when working on real systems is not convenient due to high costs, technical impossibility, and the non-existence of a real system. Simulation allows you to predict what happens to the real system if certain inputs are used. Changing these input parameters simulates different scenarios that allow us to identify the most convenient one from various points of view.

Decision-making workflow

In a decision-making process, the starting point is identifying the problematic context that requires a change and therefore a decision. The context that's identified is then analyzed in order to highlight what needs to be studied for the decisions that need to be made; that is, those elements that seem the most relevant are chosen, the relationships that connect them are highlighted, and the objectives to be achieved are defined. At this point, a formal model is constructed, which allows us to simulate the identified system in order to understand its behavior and to arrive at identifying the decisions to be made. The following diagram describes the workflow that allows us to make a decision, starting from observing the problematic context:

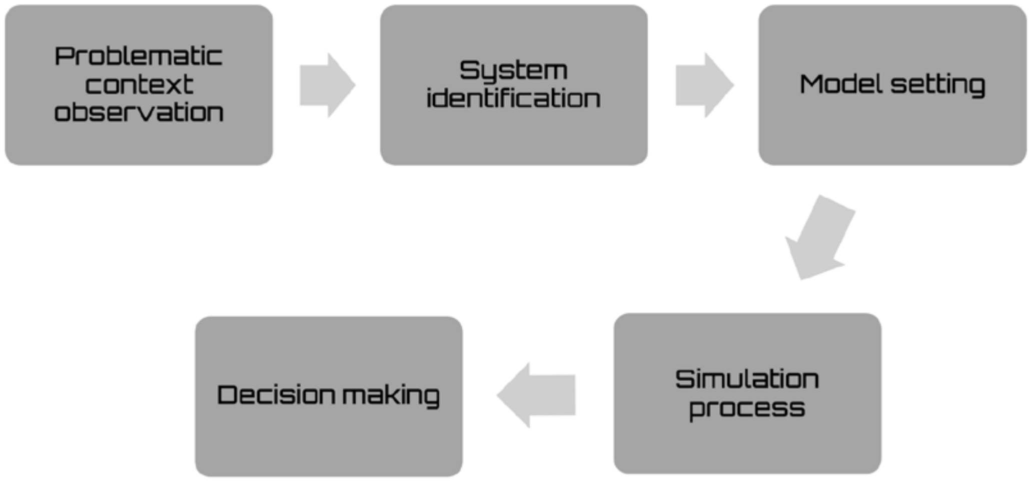


Figure 1.1 – Decision-making workflow

This represents a way of spreading knowledge and involves various actors. Constructing a model is a two-way process:

- Definition of conceptual models
- Continuous interaction between the model and reality by comparison

In addition, learning also has a participatory characteristic: it proceeds through the involvement of different actors. The models also allow you to analyze and propose organized actions so that you can modify the current situation and produce the desired solution.

Comparing modeling and simulation

To start, we will clarify the differences between modeling and simulation. A model is a representation of a physical system, while simulation is the process of seeing how a model-based system would work under certain conditions.

Modeling is a design methodology that is based on producing a model that implements a system and represents its functionality. In this way, it is possible to predict the behavior of a system and the effects of the variations or modifications that are made on it. Even if the model is a simplified representation of the system, it must still be close enough to the functional nature of the real system, but without becoming too complex and difficult to handle.

Important Note

Simulation is the process that puts the model into operation and allows you to evaluate its behavior under certain conditions. Simulation is a fundamental tool for modeling because, without necessarily resorting to physical prototyping, the developer can verify the functionality of the modeled system with the project specifications.

Simulation allows us to study the system through a wide spectrum of conditions so that we can understand how representative the model is of the system that it refers to.

Pros and cons of simulation modeling

Simulation is a tool that's widely used in a variety of fields, from operational research to the application industry. This technique can be made successful by it overcoming the difficulties that each complex procedure contains. The following are the pros and cons of simulation modeling. Let's start with the concrete advantages that can be obtained from the use of simulation models (**pros**):

- It reproduces the behavior of a system in reference to situations that cannot be directly experienced.
- It represents real systems, even complex ones, while also considering the sources of uncertainty.
- It requires limited resources in terms of data.
- It allows experimentation in limited time frames.
- The models that are obtained are easily demonstrable.

As anticipated, since it is a technique capable of reproducing complex scenarios, it has some limitations (**cons**):

- The simulation provides indications of the behavior of the system, but not exact results.
- The analysis of the output of a simulation could be complex and it could be difficult to identify which may be the best configuration.
- The implementation of a simulation model could be laborious and, moreover, it may take long calculation times to carry out a significant simulation.
- The results that are returned by the simulation depend on the quality of the input data: it cannot provide accurate results in the case of inaccurate input data.
- The complexity of the simulation model depends on the complexity of the system it intends to reproduce.

Nevertheless, simulation models represent the best solution for the analysis of complex scenarios.

Simulation modeling terminology

In this section, we will analyze the elements that make up a model and those that characterize a simulation process. We will give a brief description of each so that you understand their meaning and the role they play in the numerical simulation process.

System

The context of an investigation is represented through a system; that is, the set of elements that interact with each other. The main problem linked to this element concerns the system boundaries, that is, which elements of reality must be inserted in the system that represents it and which are left out and the relationships that exist between them.

State variables

A system is described in each instant of time by a set of variables. These are called state variables. For example, in the case of a weather system, the temperature is a state variable. In discrete systems, the variables change instantly at precise moments of time that are finite. In continuous systems, the variables vary in terms of continuity with respect to time.

Events

An event is defined as any instantaneous event that causes the value of at least one of the status variables to change. The arrival of a blizzard for a weather system is an event, as it causes the temperature to drop suddenly. There are both external events and internal events.

Parameters

Parameters represent essential terms when building a model. They are adjusted during the model simulation process to ensure that the results are brought into the necessary convergence margins. They can be modified iteratively through sensitivity analysis or in the model calibration phase.

Calibration

Calibration represents the process by which the parameters of the model are adjusted in order to adapt the results to the data observed in the best possible way. When calibrating the model, we try to obtain the best possible accuracy. A good calibration requires eliminating, or minimizing, errors in data collection and choosing a theoretical model that is the best possible description of reality. The choice of model parameters is decisive and must be done in such a way as to minimize the deviation of its results when applied to historical data.

Accuracy

Accuracy is the degree of correspondence of the simulation result that can be inferred from a series of calculated values with the actual data, that is, the difference between the average modeled value and the true or reference value. Accuracy, when calculated, provides a quantitative estimate of the quality expected from a forecast. Several indicators are available to measure accuracy. The most used are **mean absolute error** (**MAE**), **mean absolute percentage error** (**MAPE**), and **mean squared error** (**MSE**).

Sensitivity

The sensitivity of a model indicates the degree to which the model's outputs are affected by changes in the selected input parameters. A sensitivity analysis identifies the sensitive parameters for the output of the model. It allows us to determine which parameters require further investigation so that we have a more realistic evaluation of the model's output values. Furthermore, it allows us to identify which parameters are not significant for the generation of a certain output and therefore can possibly be eliminated from the model. Finally, it tells us which parameters should be considered in a possible and subsequent analysis of the uncertainty of the output values provided by the model.

Validation

This is the process that verifies the accuracy of the proposed model. The model must be validated to be used as a tool to support decisions. It aims to verify whether the model that's being analyzed corresponds conceptually to our intentions. The validation of a model is based on the various techniques of multivariate analysis, which, from time to time, study the variability and interdependence of attributes within a class of objects.

Classifying simulation models

Simulation models can be classified according to different criteria. The first distinction is between static and dynamic systems. So, let's see what differentiates them.

Comparing static and dynamic models

Static models are the representation of a system in an instant of time, or representative models of a system in which the time variable plays no role. An example of a static simulation is a Monte Carlo model.

Dynamic models, on the other hand, describe the evolution of the system over time. In the simplest case, the state of the system at time *t* is described by a function *x (t)*. For example, in population dynamics, *x (t)* represents the population present at time *t*. The equation that regulates the system is dynamic: it describes the instantaneous variation of the population or the variation in fixed time intervals.

Comparing deterministic and stochastic models

A model is deterministic when its evolution, over time, is uniquely determined by its initial conditions and characteristics. These models do not consider random elements and lend themselves to be solved with exact methods that are derived from mathematical analysis. In deterministic models, the output is well determined once the input data and the relationships that make up the model have been specified, despite the time required for data processing being particularly long. For these systems, the transformation rules univocally determine the change of state of the system. Examples of deterministic systems can be observed in some production and automation systems.

Stochastic models, on the other hand, can be evolved by inserting random elements into the evolution. These are obtained by extracting them from statistical distributions. Among the operational characteristics of these models, there is not just one relationship that fits all. There's also probability density functions, which means there is no one-to-one correspondence between the data and system history.

A final distinction is based on how the system evolves over time: this is why we distinguish between continuous and discrete simulation models.

Comparing continuous and discrete models

Continuous models represent systems in which the state of the variables changes continuously as a function of time. For example, a car moving on a road represents a continuous system since the variables that identify it, such as position and speed, can change continuously with respect to time.

In discrete models, the system is described by an overlapping sequence of physical operations, interspersed with inactivity pauses. These operations begin and end in well-defined instances (events). The system undergoes a change of state when each event occurs, remaining in the same state throughout the interval between the two subsequent events. This type of operation is easy to treat with the simulation approach.

Important Note

The stochastic or deterministic, or continuous or discrete, nature of a model is not its absolute property and depends on the observer's vision of the system itself. This is determined by the objectives and the method of study, as well as by the experience of the observer.

Now that we've analyzed the different types of models in detail, we will learn how to develop a numerical simulation model.

Approaching a simulation-based problem

To tackle a numerical simulation process that returns accurate results, it is crucial to rigorously follow a series of procedures that partly precede and partly follow the actual modeling of the system. We can separate the simulation process workflow into the following individual steps:

1. Problem analysis
2. Data collection
3. Setting up the simulation model
4. Simulation software selection

5. Verification of the software solution
6. Validation of the simulation model
7. Simulation and analysis of results

To fully understand the whole simulation process, it is essential to analyze the various phases that characterize a study based on simulation in depth.

Problem analysis

In this initial step, the goal is to understand the problem by trying to identify the aims of the study and the essential components, as well as the performance measures that interest them. Simulation is not simply an optimization technique and therefore there is no parameter that needs to be maximized or minimized. However, there is a series of performance indices whose dependence on the input variables must be verified. If an operational version of the system is already available, the work is simplified as it is enough to observe this system to deduce its fundamental characteristics.

Data collection

This represents a crucial step in the whole process since the quality of the simulation model depends on the quality of the input data. This step is closely related to the previous one. In fact, once the objective of the study has been identified, data is collected and subsequently processed. Processing the collected data is necessary to transform it into a format that can be used by the model. The origin of the data can be different: sometimes, the data is retrieved from company databases, but more often than not, direct measurements in the field must be made through a series of sensors that, in recent years, have become increasingly smart. These operations weigh down the entire study process, thus lengthening their execution times.

Setting up the simulation model

This is the crucial step of the whole simulation process; therefore, it is necessary to pay close attention to it. To set up a simulation model, it is necessary to know the probability distributions of the variables of interest. In fact, to generate various representative scenarios of how a system works, it is essential that a simulation generates random observations from these distributions.

For example, when managing stocks, the distribution of the product being requested and the distribution of time between an order and the receipt of the goods is necessary. On the other hand, when managing production systems with machines that can occasionally fail, it will be necessary to know the distribution of time until a machine fails and the distribution of repair times.

If the system is not already available, it is only possible to estimate these distributions by deriving them, for example, from the observation of similar, already existing systems. If, from the analysis of the data, it is seen that this form of distribution approximates a standard type distribution, the standard theoretical distribution can be used by carrying out a statistical test to verify whether the data can be well represented by that probability distribution. If there are no similar systems from which observable data can be obtained, other sources of information must be used: machine specifications, instruction manuals for the machines, experimental studies, and so on.

As we've already mentioned, constructing a simulation model is a complex procedure. Referring to simulating discrete events, constructing a model involves the following steps:

1. Defining the state variables
2. Identifying the values that can be taken by the state variables
3. Identifying the possible events that change the state of the system
4. Realizing a simulated time measurement, that is, a simulation clock, that records the flow of simulated time
5. Implementing a method for randomly generating events
6. Identifying the state transitions generated by events

After following these steps, we will have the simulation model ready for use. At this point, it will be necessary to implement this model in a dedicated software platform; let's see how.

Simulation software selection

The choice of the software platform that you will perform the numerical simulation with is fundamental for the success of the project. In this regard, we have several solutions that we can adopt. This choice will be made based on our knowledge of programming. Let's see what solutions are available:

- **Simulators**: These are application-oriented packages for simulation. There are numerous interactive software packages for simulation, such as MATLAB, COMSOL Multiphysics, Ansys, SolidWorks, Simulink, Arena, AnyLogic, and SimScale. These pieces of software represent excellent simulation platforms whose performance differs based on the application solutions provided. These simulators allow us to elaborate on a simulation environment using graphic menus without the need to program. They are easy to use but many of them have excellent modeling capabilities, even if you just use their standard features. Some of them provide animations that show the simulation in action, which allows you to easily illustrate the simulation to non-experts. The limitations presented by this software solution are the high costs of the licenses, which can only be faced by large companies, and the difficulty in modeling solutions that have not been foreseen by the standards.
- **Simulation languages**: A more versatile solution is offered by the different simulation languages available. There are solutions that facilitate the task of the programmer who, with these languages, can develop entire models or sub-models with a few lines of code that would otherwise require much longer drafting times, with a consequent increase in the probability of error. An example of a simulation language is the **general-purpose simulation system** (**GPSS**). This is a generic programming language that was developed by IBM in 1965. In it, a simulation clock advances in discrete steps, modeling a system as transactions enter the system and are passed from one service to another. It is mainly used as a process flow-oriented simulation language and is particularly suitable for application problems. Another example of a simulation language is SimScript, which was developed in 1963 as an extension of Fortran. SimScript is an event-based scripting language, so different parts of the script are triggered by different events.

- **GPSS**: General-purpose programming languages are designed to be able to create software in numerous areas of application. They are particularly suitable for the development of system software such as drivers, kernels, and anything that communicates directly with the hardware of a computer. Since these languages are not specifically dedicated to a simulation activity, they require the programmer to work harder to implement all the mechanisms and data structures necessary in a simulator. On the other hand, by offering all the potential of a high-level programming language, they offer the programmer a more versatile programming environment. In this way, you can develop a numerical simulation model perfectly suited to the needs of the researcher. In this book, we will use this solution by devoting ourselves to programming with Python. This software platform offers a series of tools that have been created by researchers from all over the world that make the elaboration of a numerical modeling system particularly easy. In addition, the open source nature of the projects written in Python makes this solution particularly inexpensive.

Now that we've made the choice of the software platform we're going to use and have elaborated on the numerical model, we need to verify the software solution.

Verification of the software solution

In this phase, a check is carried out on the numerical code. This is known as debugging, which consists of ensuring that the code correctly follows the desired logical flow, without unexpected blocks or interruptions. The verification must be provided in real time during the creation phase because correcting any concept or syntax errors becomes more difficult as the complexity of the model increases.

Although verification is simple in theory, debugging large-scale simulation code is a difficult task due to virtual competition. The correctness or otherwise of executions depends on time, as well as on the large number of potential logical paths. When developing a simulation model, you should divide the code into modules or subroutines in order to facilitate debugging. It is also advisable to have more than one person review the code, as a single programmer may not be a good critic. In addition, it can be helpful to perform the simulation when considering a large variety of input parameters and checking that the output is reasonable.

Important Note

One of the best techniques that can be used to verify a discrete-event simulation program is one based on tracking. The status of the system, the content of the list of events, the simulated time, the status variables, and the statistical counters are shown after the occurrence of each event and then compared with handmade calculations to check the operation of the code.

A track often produces a large volume of output that needs to be checked event by event for errors. Possible problems may arise, including the following:

- There may be information that hasn't been requested by the analyst.
- Other useful information may be missing, or a certain type of error may not be detectable during a limited debugging run.

After the verification process, it is necessary to validate the simulation model.

Validation of the simulation model

In this step, it is necessary to check whether the model that has been created provides valid results for the system in question. We must check whether the performance measurements of the real system are well approximated by the measurements generated by the simulation model. A simulation model of a complex system can only approximate it. A simulation model is always developed for a set of objectives. A model that's valid for one purpose may not be valid for another.

Important Note

Validation is a where the level of accuracy between the model and the system is respected. It is necessary to establish whether the model adequately represents the behavior of the system. The value of a model can only be defined in relation to its use. Therefore, validation is a process that aims to determine whether a simulation model accurately represents the system for the set objectives.

In this step, the ability of the model to reproduce the real functionality the system is ascertained; that is, it is ensured that the calibrated parameters, relative to the calibration scenario, can be used to correctly simulate other system situations. Once the validation phase is over, the model can be considered transferable and therefore usable for the simulation of any new control strategies and new intervention alternatives. As widely discussed in the literature on this subject, it is important to validate the model parameters that were previously calibrated on the basis of data other than that used to calibrate the model, always with reference to the phenomenon specific to the scenario being analyzed.

Simulation and analysis of results

A simulation is a process that evolves during its realization and where the initial results help lead the simulation toward more complex configurations. Attention should be paid to some details. For example, it is necessary to determine the transient length of the system before reaching stationary conditions if you want performance measures of the system at full capacity. It is also necessary to determine the duration of the simulation after the system has reached equilibrium. In fact, it must always be kept in mind that a simulation does not produce the exact values of the performance measures of a system since each simulation is a statistical experiment that generates statistical observations regarding the performance of the system. These observations are then used to produce estimates of performance measures. Increasing the duration of the simulation can increase the accuracy of these estimates.

The simulation results return statistical estimates of a system's performance measures. A fundamental point is that each measurement is accompanied by the confidence interval, within which it can vary. These results could immediately highlight a better system configuration than the others, but more often, more than one candidate configuration will be identified. In this case, further investigations may be needed to compare these configurations.

Dynamical systems modeling

In this section, we will analyze a real case of modeling a production process. In this way, we will learn how to deal with the elements of the system and how to translate the production instances into the elements of the model. A model is created to study the behavior of a system over time. It consists of a set of assumptions about the behavior of the system being expressed using mathematical logical-symbolic relationships. These relationships are between the entities that make up the system. Recall that a model is used to simulate changes in the system and predict the effects of these changes on the real system. Simple models are resolved analytically, using mathematical methods, while complex models are numerically simulated on the computer, where the data is treated as the data of a real system.

Managing workshop machinery

In this section, we will look at a simple example of how a discrete event simulation of a dynamic system is created. A discrete event system is a dynamic system whose states can assume logical or symbolic values, rather than numerical ones, and whose behavior is characterized by the occurrence of instantaneous events within an irregular timing sequence not necessarily known a priori. The behavior of these systems is described in terms of states and events.

In a workshop, there are two machines, which we will call A_1 and A_2. At the beginning of the day, five jobs need to be carried out: W_1, W_2, W_3, W_4, and W_5. The following table shows how long we need to work on the machines in minutes:

	A_1	A_2
W_1	30	50
W_2	0	40
W_3	20	70
W_4	30	0
W_5	50	20

Figure 1.2 – Table showing work time on the machines

A zero indicates that a job does not require that machine. Jobs that require two machines must pass through A_1 and then through A_2. Suppose that we decide to carry out the jobs by assigning them to each machine so that when they become available, the first executable job is started first, in the order from 1 to 5. If, at the same time, more jobs can be executed on the same machine, we will execute the one with a minor index first.

The purpose of modeling is to determine the minimum time needed to complete all the works. The events in which state changes can occur in the system are as follows:

1. A job becomes available for a machine.
2. A machine starts a job.
3. A machine ends a job.

Based on these rules and the evaluation times indicated in the previous table, we can insert the sequence of the jobs, along with the events scheduled according to the execution times, into a table:

Time (minute)	**A1**		**A2**	
	Start	End	Start	End
0	W_1		W_2	
30	W_3	W_1		
40			W_1	W_2
50	W_4	W_3		
80	W_5	W_4		
90			W_3	W_1
130		W_5		
160			W_5	W_3
180				W_5

Figure 1.3 – Table of job sequences

This table shows the times of the events in sequence, indicating the start and end of the work associated with the two machines available in the workshop. At the end of each job, a new job is sent to each machine according to the rules set previously. In this way, the deadlines for the work and the subsequent start of another job are clearly indicated. This is just as easy as it is to identify the time interval in which each machine is used and when it becomes available again. The table solution we have proposed represents a simple and immediate way of simulating a simple dynamic discrete system.

The example we just discussed is a typical case of a dynamic system in which time proceeds in steps, in a discrete way. However, many dynamic systems are best described by assuming that time passes continuously. In the next section, we will analyze the case of a continuous dynamic system.

Simple harmonic oscillator

Consider a mass *m* resting on a horizontal plane, without friction, and attached to a wall by an ideal spring, of elastic constant *k*. Suppose that, when the horizontal coordinate *x* is zero, the spring is at rest. The following diagram shows the scheme of a simple harmonic oscillator:

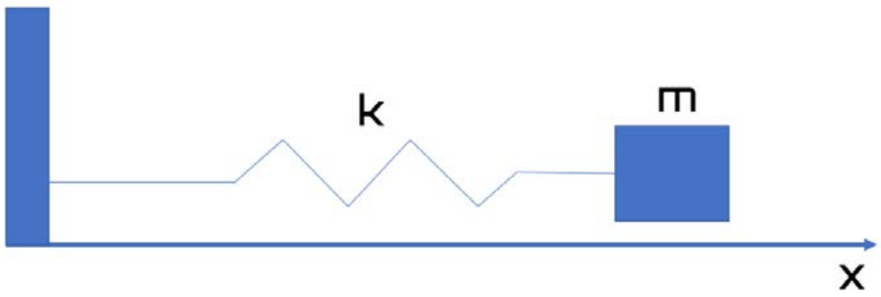


Figure 1.4 – The scheme of a harmonic oscillator

If the block of mass m is moved to the right with respect to its equilibrium position ($x > 0$), the spring, being elongated, calls it to the left. Conversely, if the block is placed to the left of its equilibrium position ($x < 0$), then the spring is compressed and pushes the block to the right. In both cases, we can express the component along the x-axis of the force due to the spring according to the following formula:

$$F_x = -k * x$$

Here, we have the following:

- F_x is the force.
- k is the elastic constant.
- x is the horizontal coordinate that indicate the position of the mass m.

From the second law of dynamics, we can derive the component of acceleration along x as follows:

$$a_x = -\frac{k}{m} * x$$

Here, we have the following:

- a_x is the acceleration.
- k is the elastic constant.
- m is the mass of the block.
- x is the horizontal coordinate that indicate the position of the mass m.

If we indicate with $\frac{dv}{dt} = a_x$ the rate of change of the speed and with $\frac{dx}{dt} = v$ the speed, we can obtain the evolution equations of the dynamic system, as follows:

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega^2 * x \end{cases}$$

Here, we have the following:

- $\omega^2 = -\frac{k}{m}$

For these equations, we must associate the initial conditions of the system, which we can write in the following way:

$$\begin{cases} x(0) = x_0 \\ v(0) = v_0 \end{cases}$$

The solutions to the previous differential equations are as follows:

$$\begin{cases} x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t) \\ v(t) = v_0 \cos(\omega t) - x_0 \omega \sin(\omega t) \end{cases}$$

In this way, we obtained the mathematical model of the analyzed system. In order to study the evolution of the oscillation phenomenon of the mass block *m* over time, it is enough to vary the time and calculate the position of the mass at that instant and its speed.

In decision-making processes characterized by high levels of complexity, the use of analytical models is not possible. In these cases, it is necessary to resort to models that differ from those of an analytical type for the use of the calculator as a tool not only for calculation, such as in mathematical programming models, but also for representing the elements that make up reality why studying the relationships between them.

Predator-prey model

In the field of simulations, simulating the functionality of production and logistic processes is considerably important. These systems are, in fact, characterized by high complexity, numerous interrelationships between the different processes that pass through them, segment failures, unavailability, and the stochasticity of the system parameters.

To understand how complex the analytical modeling of some phenomena is, let's analyze a universally widespread biological model. This is the predator-prey model, which was developed independently by the Italian researcher Vito Volterra and the American biophysicist Alfred Lotka.

On an island, there are two populations of animals: prey and predators. The vegetation of the island provides the prey with nourishment in quantities that we can consider as unlimited, while the prey is the only food available for the predators. We can consider the birth rate of the prey constant over time; this means that in the absence of predators, the prey would grow by exponential law. Their mortality rate, on the other hand, depends on the probability they have of falling prey to a predator and therefore on the number of predators present per unit area.

As for the predators, the mortality rate is constant, while their growth rate depends on the availability of food and therefore on the number of prey per unit area present on the island. We want to study the trend of the size of the two populations over time, starting from a known initial situation (number of prey and predators).

To carry out a simulation of this biological system, we can model it by means of the following system of finite difference equations, where *x(t)* and *y(t)* are the number of prey and predators at time *t*, respectively:

$$\begin{cases} \dfrac{dx}{dt} = \alpha * x(t) - \beta * x(t) * y(t) \\ \dfrac{dy}{dt} = \gamma * y(t) - \delta * x(t) * y(t) \end{cases}$$

Here, we have the following:

- α, β, γ, δ are positive real parameters related to the interaction of the two species
- $\frac{dx}{dt}$ is the instantaneous growth rates of the prey.
- $\frac{dy}{dt}$ is the instantaneous growth rates of the predators.

The following hypotheses underlie in this model:

- In the absence of predators, the number of prey increases according to an exponential law, that is, with a constant rate.
- Similarly, in the absence of prey, the number of predators decreases at a constant rate.

This is a deterministic and continuous simulation model. In fact, the state of the system, represented by the size of the two populations in each instance of time, is univocally determined by the initial state and the parameters of the model. Furthermore, at least in principle, the variables – that is, the size of the populations – vary continuously over time.

The differential system is in normal form and can be solved with respect to the derivatives of maximum order but cannot be separated into variables. It is not possible to solve this in an analytical form, but with numerical methods, it is solved immediately (the Runge-Kutta method). The solution obviously depends on the values of the four constants and the initial values.

Summary

In this chapter, we learned what is meant by simulation modeling. We understood the difference between modeling and simulation, and we discovered the strengths of simulation models, such as defects. To understand these concepts, we clarified the meaning of the terms that appear most frequently when dealing with these topics.

We then analyzed the different types of models: static versus dynamic, deterministic versus stochastic, and continuous versus discrete. We then explored the workflow connected to a numerical simulation process and highlighted the crucial steps. Finally, we studied some practical modeling cases to understand how to elaborate on a model starting from the initial considerations.

In the next chapter, we will learn how to approach a stochastic process and understand the random number simulation concepts. Then, we will explore the differences between pseudo and non-uniform random numbers, as well as the methods we can use for random distribution evaluation.

2 Understanding Randomness and Random Numbers

In many real-life situations, it is useful to flip a coin in order to decide what to do. Many computers also use this procedure as part of their decision-making process. In fact, many problems can be solved in a very effective and relatively simple way by using probabilistic algorithms. In an algorithm of this type, decisions are made based on random contributions that remember the dice roll with the help of a randomly chosen value.

The generation of random numbers has ancient roots, but only recently has the process been sped up, allowing it to be used on a large scale in scientific research as well. These generators are mainly used for computer simulations, statistical sampling techniques, or in the field of cryptography.

In this chapter, we're going to cover the following topics:

- Stochastic processes
- Random number simulation
- The pseudorandom number generator

- Testing uniform distribution
- Exploring generic methods for random distributions
- Random number generation using Python

Technical requirements

In this chapter, we will introduce random number generation techniques. In order to understand these topics, a basic knowledge of algebra and mathematical modeling is needed.

To work with the Python code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `LinearCongruentialGenerator.py`
- `LearmouthLewisGenerator.py`
- `LaggedFibonacciAlgorithm.py`
- `UniformityTest.py`
- `Random.Generation.py`

Stochastic processes

A **stochastic process** is a family of random variables that depends on a parameter, *t*. A stochastic process is specified using the following notation:

$$\{X_t , t \in T\}$$

Here, *t* is a parameter, and *T* is the set of possible values of *t*.

Usually, time is indicated by *t*, so a stochastic process is a family of time-dependent random variables. The variability range of *t*, that is, the set, *T*, can be a set of real numbers, possibly coinciding with the entire time axis. But it can also be a discrete set of values.

The random variables, X_t, are defined on the set, *X*, called the space of states. This can be a continuous set, in which case it is defined as a continuous stochastic process, or a discrete set, in which case it is defined as a discrete stochastic process.

Consider the following elements:

$$x_0, x_1, x_2, \dots, x_n \in X$$

This means the values that the random variables, X_t, can take are called system states and represent the possible results of an experiment. The X_t variables are linked together by dependency relationships. We can know a random variable if we know both the values it can assume and the probability distribution. So, to understand a stochastic process, it is necessary not only to know the values that X_t can take but also the probability distributions of the variables and the joint distributions between the values. Simpler stochastic processes, in which the variability range of *t* is a discrete set of time values, can also be considered.

Important note

In practice, there are numerous phenomena that are studied through the theory of stochastic processes. A classic application in physics is the study of the motion of a particle in each medium, the so-called **Brownian motion**. This study is carried out statistically using a stochastic process. There are processes where even by knowing the past and the present, the future cannot be determined; whereas, in other processes, the future is determined by the present without considering the past.

Types of stochastic process

Stochastic processes can be classified according to the following characteristics:

- Space of states
- Time index
- Type of stochastic dependence between random variables

The state space can be **discrete** or **continuous**. In the first case, the stochastic process with discrete space is also called a **chain**, and space is often referred to as the set of non-negative integers. In the second case, the set of values assumed by the random variables is not finite or countable, and the stochastic process is in continuous space.

The time index can also be discrete or continuous. A discrete-time stochastic process is also called a **stochastic sequence** and is denoted as follows:

$$\{X_n \mid n \in T\}$$

Here, the set, *T*, is finite or countable.

In this case, the changes of state are observed only in certain instances: finite or countable. If state changes occur at any instant in a finite or infinite set of real intervals, then there is a continuous-time process, which is denoted as follows:

$$\{X(t) \mid t \in T\}$$

The stochastic dependence between random variables, *X(t)*, for different values of *t* characterizes a stochastic process and sometimes simplifies its description. A stochastic process is stationary in the strict sense that the distribution function is invariant with respect to a shift on the time axis, *T*. A stochastic process is stationary in the broad sense that the first two moments of the distribution are independent of the position on the *T* axis.

Examples of stochastic processes

The mathematical treatment of stochastic processes seems complex, yet we find cases of stochastic processes every day. For example, the number of patients admitted to a hospital as a function of time, observed at noon each day, is a stochastic process in which the space of states is discrete, being a finite subset of natural numbers, and time is discrete. Another example of a stochastic process is the temperature measured in a room as a function of time, observed at every instant, with continuous state space and continuous time. Let's now look at a number of structured examples that are based on stochastic processes.

The Bernoulli process

The concept of a random variable allows us to formulate models that are useful for the study of many random phenomena. An important early example of a probabilistic model is the **Bernoulli distribution**, named in honor of the Swiss mathematician, James Bernoulli (1654-1705), who made important contributions to the field of probability.

Some of these experiments consist of repeatedly performing a given test. For example, we want to know the probability of getting a head when throwing a coin 1,000 times.

In each of these examples, we look for the probability of obtaining *x* successes in *n* trials. If *x* indicates the successes, then *n - x* will be the failures.

A sequence of Bernoulli trials consists of a Bernoulli trial under the following hypotheses:

- There are only two possible mutually exclusive results for each trial, arbitrarily called **success** and **failure**.
- The probability of success, *p*, is the same for each trial.
- All tests are independent.

Independence means that the result of a test is not influenced by the result of any other test. For example, the event, *the third test was successful*, is independent of the event, *the first test was successful.*

The toss of a coin is a Bernoulli trial: the *heads* event can be considered successful, and the *tails* event can be considered unsuccessful. In this case, the probability of success is $p = 1/2$. In rolling two dice, the event, *the sum of the points is seven*, and the complementary event are both unsuccessful. In this case, it is a Bernoulli trial and the probability of success is $p = 1/6$.

Important note

Two events are said to be complementary when the occurrence of the first excludes the occurrence of the second but one of the two will certainly occur.

Let p denote the probability of success in a Bernoulli trial. The random variable, X, which counts the number of successes in n trials is called the binomial random variable of the n and p parameters. X can take integer values between 0 and n.

Random walk

The **random walk** is a discrete parameter stochastic process in which X_t, where X represents a random variable, describes the position taken at time t by a moving point. The term, random walk, refers to the mathematical formalization of statistics that describe the displacement of an object that moves randomly. This kind of simulation is extremely important for a physicist and has applications in statistical mechanics, fluid dynamics, and quantum mechanics.

Random walks represent a mathematical model that is used universally to simulate a path formalized by a succession of random steps. This model can assume a variable number of degrees of freedom, depending on the system we want to describe. From a physical point of view, the path traced over time will not necessarily simulate a real motion, but it will represent the trend of the characteristics of the system over time. Random walks find applications in chemistry, biology, and physics, but also in other fields such as economics, sociology, and information technology.

Random one-dimensional walking is a model that is used to simulate the movement of a particle moving along a straight line. There are only two potential movements on the allowed path: either to the right (with a probability that is equal to **p**) or to the left (with a probability that is equal to **q**) of the current position. Each step has a constant length and is independent of the others, as shown in the following diagram:

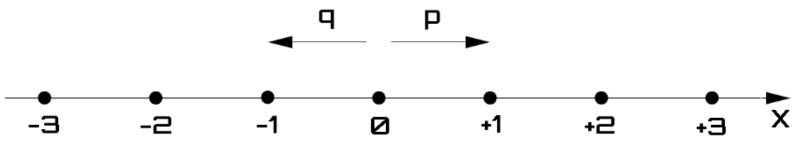


Figure 2.1 – One-dimensional walking

The position of the point in each instant is identified by its abscissa, *X(n)*. This position, after *n* steps, will be characterized by a random term. Our aim is to calculate the probability of passing from the starting point after *n* movements. Obviously, nothing assures us that the point will return to the starting position. The variable, *X(n)*, returns the abscissa of the particle after *n* steps. It is a discrete random variable with a binomial distribution.

At each instant, the particle steps right or left based on the value returned by a random variable, *Z(n)*. This variable can take only two values: +1 and -1. It assumes a + 1 value with a probability of $p > 0$ and a value of -1 with a probability that is equal to q. The sum of the two probabilities is $p + q = 1$. The position of the particle at instant n is given by the following equation:

$$X_n = X_{n-1} + Z_n \, ; \; n = 1, 2, \ldots$$

This shows the average number of returns to the origin of the particle, named p. The probability of a single return is given by the following geometric series:

$$\mu = \sum_{n=0}^{\infty} n \, p^n (1-p) = \sum_{n=0}^{\infty} n \, p^n \frac{1}{\sqrt{n * \pi}} \to \infty$$

We assume that the probability of the particle returning to the origin tends to 1. This means that despite the frequency of the returns decreasing with the increase in the number of steps taken, they will always be in an infinite value of steps taken. So, we can conclude that a particle with equal probability of left and right movement, left free to walk casually to infinity with great probability, returns infinite times to the point from which it started.

The Poisson process

There are phenomena in which certain events, with reference to a certain interval of time or space, rarely happen. The number of events that occur in that interval varies from 0 to *n*, and *n* cannot be determined a priori. For example, the number of cars passing through an uncrowded street in a randomly chosen 5-minute time frame can be considered a rare event. Similarly, the number of accidents at work that happen at a company in a week, or the number of printing errors on a page of a book, is rare.

In the study of rare events, a reference to a specific interval of time or space is fundamental. For the study of rare events, the Poisson probability distribution is used, named in honor of the French mathematician, Simeon Denis Poisson (1781-1840), who first obtained the distribution. The Poisson distribution is used as a model in cases where the events or realizations of a process, distributed randomly in space or time, are counts, that is, discrete variables.

The binomial distribution is based on a set of hypotheses that define the Bernoulli trials, and the same happens for the Poisson distribution. The following conditions describe the so-called Poisson process:

- The realizations of the events are independent, meaning that the occurrence of an event in a time or space interval has no effect on the probability of the event occurring a second time in the same, or another, interval.
- The probability of a single realization of the event in each interval is proportional to the length of the interval.
- In any arbitrarily small part of the interval, the probability of the event occurring more than once is negligible.

An important difference between the Poisson distribution and the binomial distribution is the number of trials and successes. In a binomial distribution, the number, *n*, of trials is finite and the number, *x*, of successes cannot exceed *n*; in a Poisson distribution, the number of tests is essentially infinite and the number of successes can be infinitely large, even if the probability of having *x* successes becomes very small as *x* increases.

Random number simulation

The availability of random numbers is a necessary requirement in many applications. In some cases, the quality of the final application strictly depends on the possibility of generating good quality random numbers. Think, for example, of applications such as video games, cryptography, generating visuals or sound effects, telecommunications, signal processing, optimizations, and simulations. In an algorithm of this type, decisions are made based on the pull of a virtual currency, which is based on a randomly chosen value.

There is no single or general definition of a random number since it often depends on the context. The concept of random number itself is not absolute, as any number or sequence of numbers can appear to be random to an observer, but not to another who knows the law with which they are generated. Put simply, a random number is defined as a number selected in a random process from a finite set of numbers. With this definition, we focus on the concept of randomness in the process of selecting a sequence of numbers.

In many cases, the problem of generating random numbers concerns the random generation of a sequence of 0 and 1, from which numbers in any format can be obtained: integers, fixed points, floating points, or strings of arbitrary length. With the right functions, it is possible to obtain good quality sequences that can also be used in scientific applications, such as the **Monte Carlo** simulation. These techniques should be easy to implement and be usable by any computer. In addition, like all software solutions, they should be very versatile and quickly improved.

Important note

These techniques have a big problem that is inherent to the algorithmic nature of the process: the final string can be predicted from the starting seed. This is why we call this process **pseudorandom**.

Despite this, many problems of an algorithmic nature are solved very effectively and relatively simply using probabilistic algorithms. The simplest example of a probabilistic algorithm is perhaps the randomized quicksort. This is a probabilistic variant of the homonymous sorting algorithm, where, by choosing the pivot element, the algorithm manages to randomly guarantee optimal complexity in the average case, no matter the distribution of the input. Cryptography is a field in which randomness plays a fundamental role and deserves specific mention. In this context, randomness does not lead to computational advantages, but it is essential to guarantee the security of authentication protocols and encryption algorithms.

Probability distribution

It is possible to characterize a random process from different points of view. One of the most important characteristics is the **probability distribution**. The probability distribution is a model that associates a probability with each observable modality of a random variable.

The probability distribution can be either discrete or continuous, depending on whether the variable is random, discrete, or continuous. It is discrete if the phenomenon is observable with an integer number of modes. The throw of the dice is a discrete statistical phenomenon because the number of observable modalities is equal to 6. The random variable can take only six values (1, 2, 3, 4, 5, and 6). Therefore, the probability distribution of the phenomenon is discrete. The probability distribution is continuous when the random variable assumes a continuous set of values; in this case, the statistical phenomenon can be observed with an infinite or very high number of modalities. The probability distribution of body temperature is continuous because it is a continuous statistical phenomenon, that is, the values of the random variable vary continuously.

Let's now look at different kinds of probability distributions.

Uniform distribution

In many cases, processes characterized by a **uniform distribution** are considered and used. This means that each element is as likely as any of the others to be selected if an infinite number of extractions is performed. If you represent the elements and their respective probabilities of being extracted on a graph, you get a rectangular graph as follows:

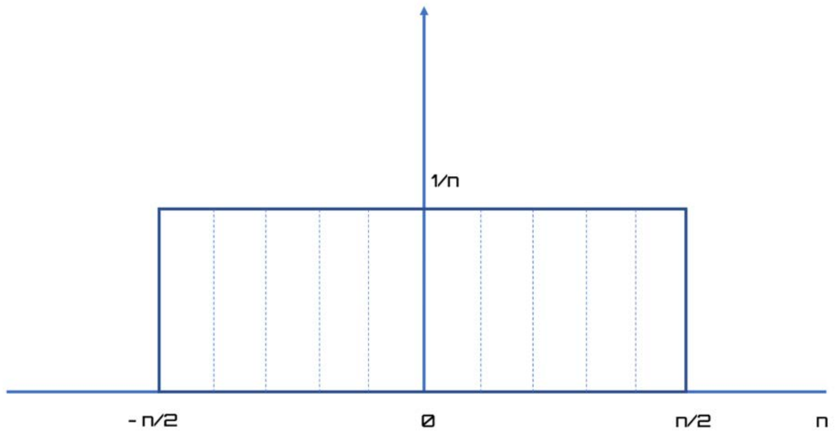


Figure 2.2 – The probabilities of the elements

Since the probability is expressed as a real number between 0 and 1, where 0 represents the impossible event and 1 the certain event, in a uniform distribution, each element will have a $1/n$ probability of being selected, where n is the number of items. In this case, the sum of all the probabilities must give a uniform result, since, in an extraction, at least one of the elements is chosen for sure. A uniform distribution is typical of artificial random processes such as dice rolling, lotteries, and roulette and is also the most used in several applications.

Gaussian distribution

Another very common probability distribution is the **Gaussian** or **normal distribution**, which has a typical bell shape. In this case, the smaller values, or those that are closer to the center of the curve, are more likely to be extracted than the larger ones, which are far away from the center. The following diagram shows a typical Gaussian distribution:

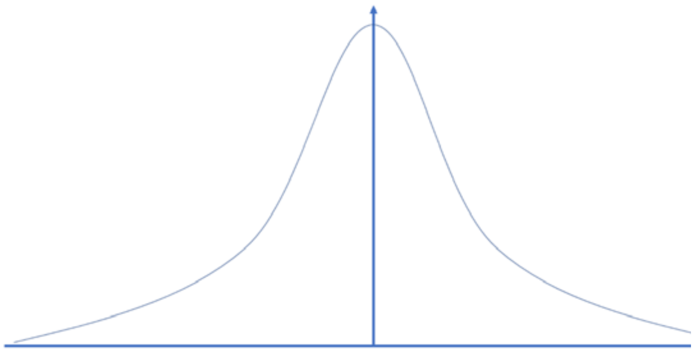


Figure 2.3 – Gaussian distribution

Gaussian distribution is important because it is typical of natural processes. For example, it can represent the distribution of the noise in many electronic components, or it can represent the distribution of errors in measurements. It is, therefore, used to simulate statistical distributions in the fields of telecommunications or signal processing.

Properties of random numbers

By random number, we refer to a random variable distributed in a uniform way between 0 and 1. The statistical properties that a sequence of random numbers must possess are as follows:

- Uniformity
- Independence

Suppose you divide an interval, [0.1], into *n* subintervals of equal amplitude. The consequence of the uniformity property is that if *N* observations of a random number are made, then the number of observations in each subinterval is equal to *N/n*. The consequence of the independence property is that the probability of obtaining a value in a particular range is independent of the values that were previously obtained.

The pseudorandom number generator

The generation of real random sequences using deterministic algorithms is impossible: at most, pseudorandom sequences can be generated. These are, apparently, random sequences that are actually perfectly predictable and can be repeated after a certain number of extractions. A PRNG is an algorithm designed to output a sequence of values that appear to be generated randomly.

The pros and cons of a random number generator

A random number generation routine must be the following:

- Replicable
- Fast
- Not have large gaps between two generated numbers
- Have a sufficiently long running period
- Generate numbers with statistical properties that are as close as possible to ideal ones

The most common cons of random number generators are as follows:

- Numbers not uniformly distributed
- Discretization of the generated numbers
- Incorrect mean or variance
- Presence of cyclical variations

Random number generation algorithms

The first to deal with the random number generation problem was John von Neumann, in 1949. He proposed a method called **middle-square**. This method allows us to understand some important characteristics of a random number generation process. To start, we need to provide an input as the seed or a value that starts the sequence. This is necessary to be able to generate different sequences each time. However, it is important to ensure that the good behavior of the generator does not depend on which seed is used. From here, the first flaw of the middle-square method appears, that is, when using the zero value as a seed, only a sequence of zeros will be obtained.

Another shortcoming of this method is the repetitiveness of the sequences. As in all of the other PRNGs that will be discussed, each value depends on the previous one, and, at most, on the internal state variables of the generator. Since this is a limited number of digits, the sequence can only be repeated from a certain point onward. The length of the sequence, before it begins to repeat itself, is called the period. A long period is important because many practical applications require a large amount of random data, and a repetitive sequence might be less effective. In such cases, it is important that the choice of the seed has no influence on the potential outcomes.

Another important aspect is the efficiency of the algorithm. The size of the output data values and internal state, and, therefore, the generator input (seed), are often intrinsic features of the algorithm and remain constant. For this reason, the efficiency of a PRNG should be assessed not so much in terms of computational complexity, but in terms of the possibility of a fast and efficient implementation of the calculation architectures available. In fact, depending on the architecture you are working on, the choice of different PRNGs, or different design parameters of a certain PRNG, can result in a faster implementation by many orders of magnitude.

Linear congruential generator

One of the most common methods for generating random numbers is the **Linear Congruence Generator (LCG)**. The theory on which it rests is simple to understand and implement. It also has the advantage of being computationally light. The recursive relationship underlying this technique is provided by the following equation:

$$x_{k+1} = (a * x_k + c) \; mod \; m$$

Here, we can observe the following:

- a is the multiplier (non-negative integers)
- c is the increment (non-negative integers)

- m is the mode (non-negative integers)
- x_0 is the initial value (seed or non-negative integers)

The **modulo function**, denoted by *mod*, results in the remainder of the Euclidean division of the first number by the second. For example, 18 mod 4 gives 2 as it is the remainder of the Euclidean division between the two numbers.

The linear congruence technique has the following characteristics:

- It is cyclical with a period that is approximately equal to m
- The generated numbers are discretized

To use this technique effectively, it is necessary to choose very large m values. As an example, set the parameters of the method and generate the first 16 pseudorandom values. Here is the Python code that allowed us to generate that sequence of numbers:

```
import numpy as np
a = 2
c = 4
m = 5
x = 3

for i in range (1,17):
    x= np.mod((a*x+c), m)
    print(x)
```

The following results are returned:

```
0
4
2
3
0
4
2
3
0
4
2
3
```

```
0
4
2
3
```

In this case, the period is equal to `4`. It is easy to verify that, at most, *m* distinct integers, X_n, can be generated in the interval, [0, *m* - 1]. If *c* = 0, the generator is called **multiplicative**. Let's analyze the Python code line by line. The first line was used to import the library:

```
import numpy as np
```

`numpy` is a library of additional scientific functions of the Python language, designed to perform operations on vectors and dimensional matrices. `numpy` allows you to work with vectors and matrices more efficiently and faster than you can do with lists and lists of lists (matrices). In addition, it contains an extensive library of high-level mathematical functions that can operate on these arrays.

After importing the `numpy` library, we set the parameters that will allow us to generate random numbers using LCG:

```
a = 2
c = 4
m = 5
x = 3
```

At this point, we can use the LCG formula to generate random numbers. We only generate the first 16 numbers, but we will see from the results that these are enough to understand the algorithm. To do this, we use a `for` loop:

```
for i in range (1,17):
    x= np.mod((a*x+c), m)
    print(x)
```

To generate random numbers according to the LCG formula, we have used the `np.mod()` function. This function returns the remainder of a division when given a dividend and divisor.

Random numbers with uniform distribution

A sequence of numbers uniformly distributed between [0, 1] can be obtained using the following formula:

$$U_n = \frac{X_n}{m}$$

The obtained sequence is periodic, with a period less than or equal to m. If the period is m, then it has a full period. This occurs when the following conditions are true:

- If m and c have prime numbers
- If m is divisible by a prime number, b, for which it must also be divisible
- If m is divisible by 4, then a - 1 must also be divisible by 4

Important note

By choosing a large value of m, you can reduce both the phenomenon of periodicity and the problem of generating rational numbers.

Furthermore, it is not necessary for simulation purposes that all numbers between [0, 1] are generated, because these are infinite. However, it is necessary that as many numbers as possible within the range have the same probability of being generated.

Generally, a value of m is $m \geq 109$ so that the generated numbers constitute a dense subset of the interval, [0, 1].

An example of a multiplicative generator that is widely used in 32-bit computers is the **Learmonth-Lewis generator**. This is a generator in which the parameters assume the following values:

- `a = 75`
- `c = 0`
- `m =` 2^{31} `- 1`

Let's analyze the code that generates the first 100 random numbers according to this method:

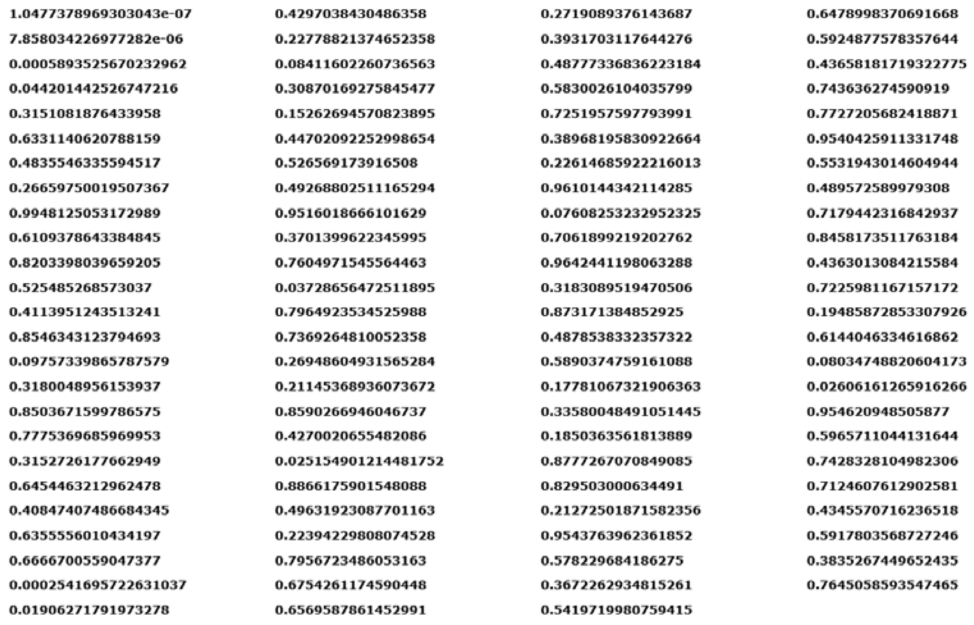
```
import numpy as np
a = 75
c = 0
m = 2**(31) -1
x = 0.1
```

```
for i in range(1,100):
    x= np.mod((a*x+c),m)
    u = x/m
    print(u)
```

The code we have just seen was analyzed line by line in the *Linear congruential generator* section of this chapter. The difference, in addition to the values of the parameters, lies in the generation of a uniform distribution in the range of [0, 1] through the following command:

```
u = x/m
```

The following results are returned:



```
1.0477378969303043e-07    0.4297038430486358      0.2719089376143687      0.6478998370691668
7.858034226977282e-06     0.22778821374652358     0.3931703117644276      0.5924877578357644
0.0005893525670232962     0.08411602260736563     0.48777336836223184     0.43658181719322775
0.044201442526747216      0.30870169275845477     0.5830026104035799      0.743636274590919
0.3151081876433958        0.15262694570823895     0.7251957597793991      0.7727205682418871
0.6331140620788159        0.44702092252998654     0.38968195830922664     0.9540425911331748
0.4835546335594517        0.526569173916508       0.22614685922216013     0.5531943014604944
0.26659750019507367       0.49268802511165294     0.9610144342114285      0.489572589979308
0.9948125053172989        0.9516018666101629      0.07608253232952325     0.7179442316842937
0.6109378643384845        0.3701399622345995      0.7061899219202762      0.8458173511763184
0.8203398039659205        0.7604971545564463      0.9642441198063288      0.4363013084215584
0.525485268573037         0.03728656472511895     0.3183089519470506      0.7225981167157172
0.4113951243513241        0.7964923534525988      0.873171384852925       0.19485872853307926
0.8546343123794693        0.7369264810052358      0.4878538332357322      0.6144046334616862
0.09757339865787579       0.26948604931565284     0.5890374759161088      0.08034748820604173
0.3180048956153937        0.21145368936073672     0.17781067321906363     0.02606161265916266
0.8503671599786575        0.8590266946046737      0.33580048491051445     0.954620948505877
0.7775369685969953        0.4270020655482086      0.1850363561813889      0.5965711044131644
0.3152726177662949        0.025154901214481752    0.8777267070849085      0.7428328104982306
0.6454463212962478        0.8866175901548088      0.829503000634491       0.7124607612902581
0.40847407486684345       0.49631923087701163     0.21272501871582356     0.4345570716236518
0.6355556010434197        0.22394229808074528     0.9543763962361852      0.5917803568727246
0.6666700559047377        0.7956723486053163      0.578229684186275       0.3835267449652435
0.0002541695722631037     0.6754261174590448      0.3672262934815261      0.7645058593547465
0.01906271791973278       0.6569587861452991      0.5419719980759415
```

Figure 2.4 – LCG output

Since we are dealing with random numbers, the output will be different from the previous one.

A comparison between the different generators must be made based on the analysis of the periodicity, the goodness of the uniformity of the numbers generated, and the computational simplicity. This is because the generation of very large numbers can lead to the use of expensive computer resources. Also, if the X_n numbers get too big, they are truncated, which can cause a loss of the desired uniformity statistics.

Lagged Fibonacci generator

The **lagged Fibonacci algorithm** for generating pseudorandom numbers arises from the attempt to generalize the method of linear congruences. One of the reasons that led to the search for new generators was the need, useful for many applications especially in parallel computing, to lengthen the generator period. The period of a linear generator when *m* is approximately 109 is enough for many applications, but not all of them.

One of the techniques developed is to make X_{n+1} dependent on the two previous values, X_n and X_{n-1}, instead of only on X_n, as is the case in the LCG method. In this case, the period may arrive close to the value, m_2, because the sequence will not repeat itself until the following equality is obtained:

$$(X_{n+\lambda}, X_{n+\lambda+1}) = (X_n, X_{n+1})$$

The simplest generator of this type is the Fibonacci sequence represented by the following equation:

$$X_{n+1} = (X_n + X_{n-1})\, mod\, m$$

This generator was first analyzed in the 1950s and provides a period, *m*, but the sequence does not pass the simplest statistical tests. We then tried to improve the sequence using the following equation:

$$X_{n+1} = (X_n + X_{n-k})\, mod\, m$$

This sequence, although better than the Fibonacci sequence, does not return satisfactory results. We had to wait until 1958, when Mitchell and Moore proposed the following sequence:

$$X_n = (X_{n-24} + X_{n-55})\, mod\, m, \qquad n \geq 55$$

Here, *m* is even and X_0, ... X_{54} are arbitrary integers that are not all even. Constants 24 and 55 are not chosen at random but are numbers that define a sequence whose least significant bits (X_n mod 2) have a period of length 2^{55}-1. Therefore, the sequence (X_n) must have a period of length of at least 2^{55}-1. The succession has a period of 2^M-1 (2^{55}-1) where $m = 2^M$.

Numbers 24 and 55 are commonly called lags and the sequence (X_n) is called a **Lagged Fibonacci Generator** (**LFG**). The LFG sequence can be generalized with the following equation:

$$X_n = (X_{n-l} \otimes X_{n-k})\, mod\, 2^M, l > k > 0$$

Here, $\otimes$ refers to any of the following operations: +, −, ×, or $\otimes$ (exclusive or).

Only some pairs (*k*, *l*) give sufficiently long periods. In these cases, the period is 2^M-1 (2_l-1). The pairs (*k*, *l*) must be chosen appropriately. The only condition on the first *l* values is that at least one of them must be odd; otherwise, the sequence will be composed of even numbers.

Let's look at how to implement a simple example of additive LFG in Python using the following parameters: $x0 = x1 = 1$ and $m = 2^{32}$. Here is the code to generate the first 100 random numbers:

```
import numpy as np
x0=1
x1=1
m=2**32

for i in range (1,101):
    x= np.mod((x0+x1), m)
    x0=x1
    x1=x
    print(x)
```

Let's analyze the Python code line by line. The first line was used to import the library:

```
import numpy as np
```

After importing the `numpy` library, we set the parameters that will allow us to generate random numbers using LFG:

```
x0=1
x1=1
m=2**32
```

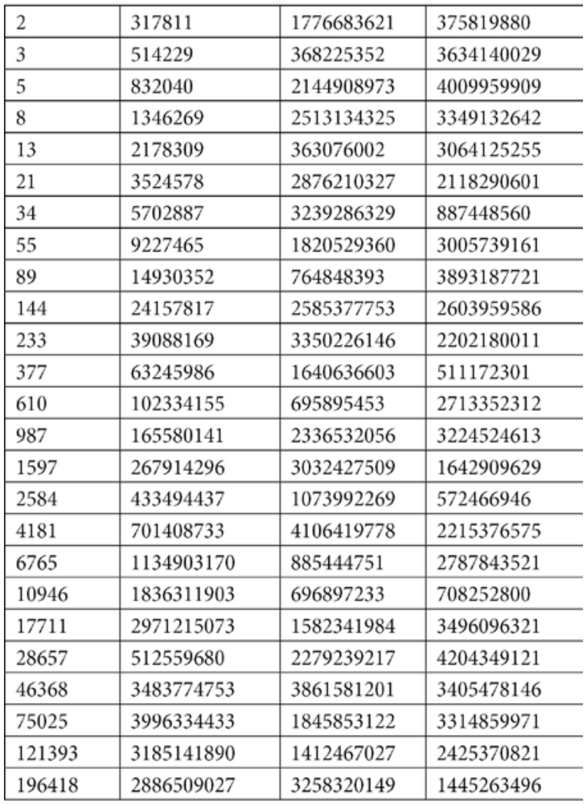
At this point, we can use the LFG formula to generate random numbers. We only generate the first 100 numbers. To do this, we use a `for` loop:

```
for i in range (1,101):
    x= np.mod((x0+x1), m)
    x0=x1
    x1=x
    print(x)
```

To generate random numbers according to the LFG formula, we use the `np.mod()` function. This function returns the remainder of a division when given a dividend and divisor. After generating the random number, the two previous values are updated as follows:

```
x0=x1
x1=x
```

The following random numbers are printed:



2	317811	1776683621	375819880
3	514229	368225352	3634140029
5	832040	2144908973	4009959909
8	1346269	2513134325	3349132642
13	2178309	363076002	3064125255
21	3524578	2876210327	2118290601
34	5702887	3239286329	887448560
55	9227465	1820529360	3005739161
89	14930352	764848393	3893187721
144	24157817	2585377753	2603959586
233	39088169	3350226146	2202180011
377	63245986	1640636603	511172301
610	102334155	695895453	2713352312
987	165580141	2336532056	3224524613
1597	267914296	3032427509	1642909629
2584	433494437	1073992269	572466946
4181	701408733	4106419778	2215376575
6765	1134903170	885444751	2787843521
10946	1836311903	696897233	708252800
17711	2971215073	1582341984	3496096321
28657	512559680	2279239217	4204349121
46368	3483774753	3861581201	3405478146
75025	3996334433	1845853122	3314859971
121393	3185141890	1412467027	2425370821
196418	2886509027	3258320149	1445263496

Figure 2.5 – Table of random numbers using LFG

The initialization of an LFG is particularly complex, and the results of this method are very sensitive to the initial conditions. If extreme care is not taken when choosing the initial values, statistical defects may occur in the output sequence. These defects could harden the initial values along with subsequent values that have a careful periodicity. Another potential problem with LFG is that the mathematical theory behind the method is incomplete, making it necessary to rely on statistical tests rather than theoretical performance.

Testing uniform distribution

Test adaptation (that is, the goodness of fit) in general, has the purpose of verifying whether a variable under examination does or does not have a certain hypothesized distribution on the basis, as usual, of experimental data. It is used to compare a set of frequencies observed in a sample, with similar theoretical quantities assumed for the population. By means of the test, it is possible to quantitatively measure the degree of deviation between the two sets of values.

The results obtained in the samples do not always exactly agree with the theoretical results that are expected according to the rules of probability. Indeed, it is very rare for this to occur. For example, although theoretical considerations lead us to expect 100 heads and 100 tails from 200 flips of a coin, it is rare that these results are obtained exactly. However, despite this, we must not unnecessarily deduce that the coin is rigged.

The chi-squared test

The **chi-squared test** is a test of hypotheses that gives us back the significance of the relationship between two variables. It is a statistical inference technique that is based on the chi-squared statistic and its probability distribution. It can be used with nominal and/or ordinal variables, generally arranged in the form of contingency tables.

The main purpose of this statistic is to verify the differences between observed and theoretical values, called *expected* values, and to make an inference on the degree of deviation between the two. The technique is used with three different objectives that are all based on the same fundamental principle:

- The randomness of the distribution of a categorical variable
- The independence of two qualitative variables (nominal or ordinal)
- The differences with a theoretical model

For now, we will just consider the first aspect. The method consists of a comparison procedure between the observed empirical frequencies and the theoretical frequencies. Let's consider the following definitions:

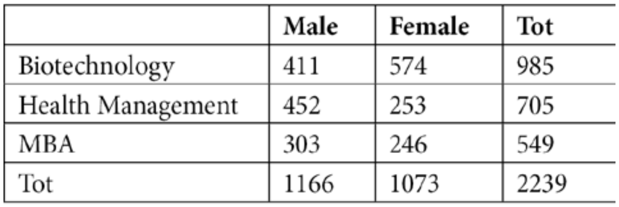
- $\mathbf{H_0}$: Null hypothesis or the absence of a statistical relationship between two variables
- $\mathbf{H_1}$: Research hypothesis that supports the existence of the relationship, for instance, H_1 is true if H_0 is false
- $\mathbf{F_o}$: Observed frequencies, that is, the number of data of a cell detected
- $\mathbf{F_e}$: Expected frequencies, that is, the frequency that should be obtained based on the marginal totals if there was no association between the two variables considered

The chi-squared test is based on the difference between observed and expected frequencies. If the observed frequency is very different from the expected frequency, then there is an association between the two variables.

As the difference between the observed frequency and the expected frequency increases, so does the value of chi-squared. The chi-squared value is calculated using the following equation:

$$\chi^2 = \sum \frac{(F_o - F_e)^2}{F_e}$$

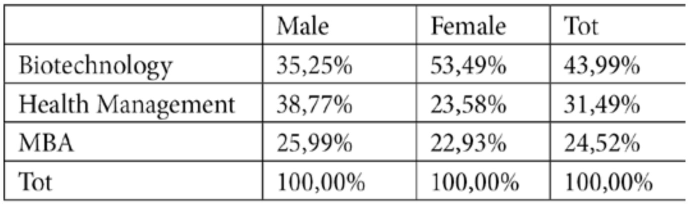
Let's look at an example to understand how to calculate this value. We build a contingency table that shows student choices for specific courses divided by genres. These are the observed values:



	Male	**Female**	**Tot**
Biotechnology	411	574	985
Health Management	452	253	705
MBA	303	246	549
Tot	1166	1073	2239

Figure 2.6 – Table of student choices divided by genres

In addition, we calculate the representation of each value as a percentage of column totals (observed frequencies):



	Male	Female	Tot
Biotechnology	35,25%	53,49%	43,99%
Health Management	38,77%	23,58%	31,49%
MBA	25,99%	22,93%	24,52%
Tot	100,00%	100,00%	100,00%

Figure 2.7 – Table of choices in percentages of column totals

Now we calculate the expected value, as follows:

$$\text{Expected value} = \frac{(Tot\ row) * \ (Tot\ column)}{Tot}$$

Let's calculate it for the first cell (Biotechnology – Male):

$$\text{Expected value} = \frac{(985) * \ (1166)}{2239} = 512.9567$$

The contingency matrix of expected values is as follows:

	Male	Female
Biotechnology	512.9567	472.0433
Health Management	367.1416	337.8584
MBA	285.9017	263.0983

Figure 2.8 – Table of contingency matrix

Let's calculate the contingency differences (*Fo – Fe*), as follows:

	Male	Female
Biotechnology	-101.957	101.9567
Health Management	84.85842	-84.8584
MBA	17.09826	-17.0983

Figure 2.9 – Table of contigency differences

Finally, we can calculate the chi-squared value, as follows:

$$\chi^2 = \sum \frac{(F_o - F_e)^2}{F_e} = \frac{(-101.957)^2}{512.9567} + \frac{(-101.957)^2}{472.0433} + \cdots = 84.35$$

If the two characters were independent, we would expect a chi-squared value of zero. On the other hand, random fluctuations are always possible. So, even in the case of perfect independence, we will never have zero. Therefore, even chi-squared values that are far from zero could make the result compatible with the null hypothesis, H_0, of independence between the variables.

Important note

One question that arises is whether the value obtained is only the result of a fluctuation, or does it arise from the dependence between the data?

Statistical theory tells us that if the variables are independent, the distribution of the chi-squared frequencies follows an asymmetric curve. In our case, we have a frequency distribution table of two features: course and gender. That is, the course feature with three modes and the genre feature with two modes. In the case of independence, how much is the square value that leaves a 5% probability on the right?

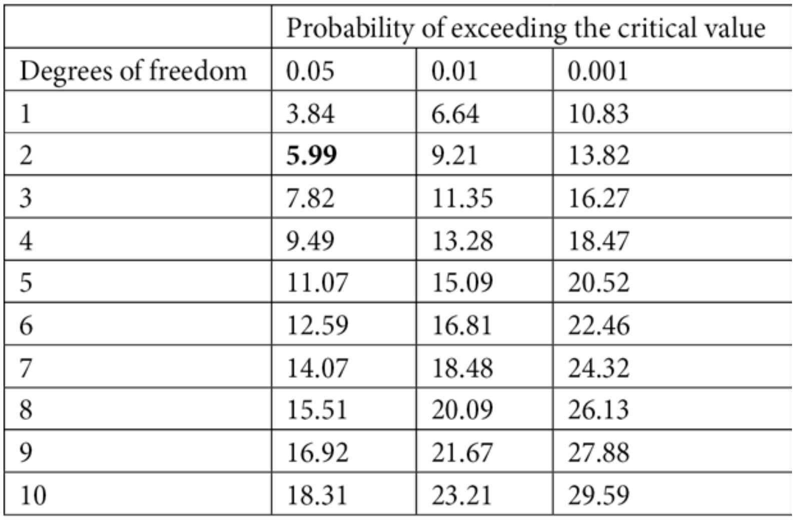
To answer this question, we must first calculate the so-called degrees of freedom, n, which is defined as follows:

$$n = (number\ of\ rows - 1) * (number\ of\ columns - 1)$$

In our case, from the contingency table, we obtain the following:

$$n = (3 - 1) * (3 - 1) = 2 * 1 = 2$$

Now we need the following chi-squared distribution table:



	Probability of exceeding the critical value		
Degrees of freedom	0.05	0.01	0.001
1	3.84	6.64	10.83
2	**5.99**	9.21	13.82
3	7.82	11.35	16.27
4	9.49	13.28	18.47
5	11.07	15.09	20.52
6	12.59	16.81	22.46
7	14.07	18.48	24.32
8	15.51	20.09	26.13
9	16.92	21.67	27.88
10	18.31	23.21	29.59

Figure 2.10 – Chi-squared distribution table

In the previous table, we look for the value, $n = 2$, in the first column, and then we scroll through the rows until we reach the column that is equal to 0.05. Here, we find the following:

$$\chi^2_{2,0.05} = 5.99$$

This means that if the data was independent, we would only have a 5% chance of getting $\chi^2 > 5.99$ from the calculations. Having obtained $\chi^2 = 84.35$, we can discard the null hypothesis, H_0, of independence from the data with a confidence of 5%. This means the possibility that H_0 is true is only 5%. Therefore, the research hypothesis, H_1, will be true, with 95% confidence.

Uniformity test

After having generated the pseudorandom numerical sequence, it is necessary to verify the goodness of the obtained sequence. It is a question of checking whether the sequence obtained, which constitutes a random sample of the experiment, follows a uniform distribution. To carry out this check, we can use the χ2 test (chi-squared test). Let's demonstrate how to do this.

The first operation is to divide the interval, [0, 1], into *s* subintervals of the same length. Then, we count how many numbers of the generated sequence are included in the *i*-th interval, as follows:

$$R_i = \{x_i \,|x_j \in s_i, \qquad j = 1, \dots N\}$$

The R_i values should be as close as possible to the *N/s* value. If the sequence were perfectly uniform, then each subinterval would have the same number of samples in the sequence.

We indicate, with *V*, the variable to perform the test. This variable is calculated using the following formula:

$$V = \sum_{i=1}^{s} \frac{\left(R_i - \frac{N}{s}\right)^2}{\frac{N}{s}}$$

After introducing the tools that allow us to perform a uniformity test, let's analyze a practical example that will help us to understand how to carry out this procedure. We generate a pseudorandom numerical sequence of 100 values, by means of the congruent linear generator, by fixing the parameters as follows:

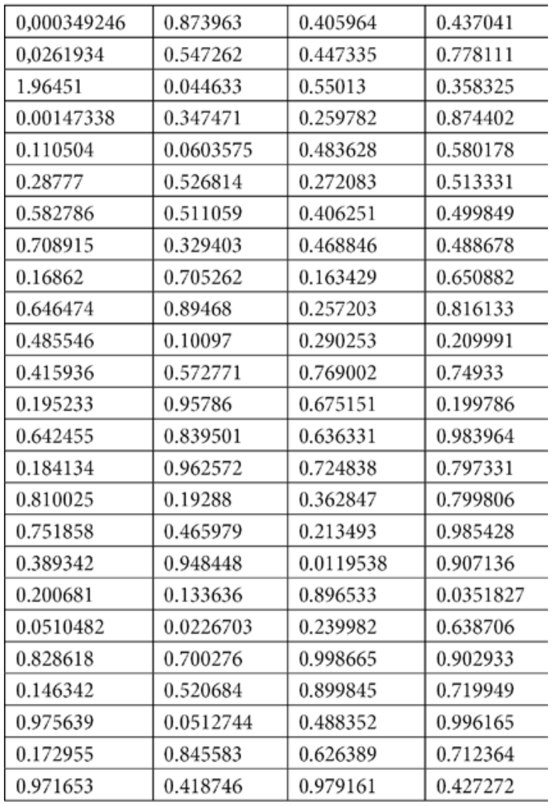
- `a = 75`
- `c = 0`
- `m =` 2^{31} `- 1`

This is the random number generator already seen in the *Lagged Fibonacci generator* section. We have already introduced the code that allows us to generate the sequence, so let's modify it for our new requirements by storing the sequence in an array:

```
import numpy as np
a = 75
c = 0
m = 2**(31) -1
x = 0.1
u=np.array([])

for i in range(0,100):
    x= np.mod((a*x+c),m)
    u= np.append(u,x/m)
    print(u[i])
```

The following results are returned:



0,000349246	0.873963	0.405964	0.437041
0,0261934	0.547262	0.447335	0.778111
1.96451	0.044633	0.55013	0.358325
0.00147338	0.347471	0.259782	0.874402
0.110504	0.0603575	0.483628	0.580178
0.28777	0.526814	0.272083	0.513331
0.582786	0.511059	0.406251	0.499849
0.708915	0.329403	0.468846	0.488678
0.16862	0.705262	0.163429	0.650882
0.646474	0.89468	0.257203	0.816133
0.485546	0.10097	0.290253	0.209991
0.415936	0.572771	0.769002	0.74933
0.195233	0.95786	0.675151	0.199786
0.642455	0.839501	0.636331	0.983964
0.184134	0.962572	0.724838	0.797331
0.810025	0.19288	0.362847	0.799806
0.751858	0.465979	0.213493	0.985428
0.389342	0.948448	0.0119538	0.907136
0.200681	0.133636	0.896533	0.0351827
0.0510482	0.0226703	0.239982	0.638706
0.828618	0.700276	0.998665	0.902933
0.146342	0.520684	0.899845	0.719949
0.975639	0.0512744	0.488352	0.996165
0.172955	0.845583	0.626389	0.712364
0.971653	0.418746	0.979161	0.427272

Figure 2.11 – Output table of LFG random numbers

To better understand how the numbers are distributed in the range considered, we will divide the interval, [0, 1], into 20 parts `(s = 20)`, and then count how many values of the sequence fall into each interval of amplitude `0.05`.

Finally, we calculate the `V` variable:

```
N=100
s=20
Ns =N/s
S = np.arange(0, 1, 0.05)
counts = np.empty(S.shape, dtype=int)
V=0
for i in range(0,20):
    counts[i] = len(np.where((u >= S[i]) & (u < S[i]+0.05))[0])
    V=V+(counts[i]-Ns)**2 / Ns

print("R = ",counts)
print("V = ", V)
```

Let's analyze the code line by line:

```
N=100
s=20
Ns =N/s
```

The first three lines set the variable, `N` (the number of random numbers), and `s` (the number of pats), and then we calculated the ratio.

After that, we divide the interval, [0, 1], into 20 subintervals:

```
S = np.arange(0, 1, 0.05)
```

Now we initialize the `counts` array, which contains how many values of the sequence fall in each interval, and the *V* variable, as follows:

```
counts = np.empty(S.shape, dtype=int)
V=0
```

To count how many values of the sequence fall in each interval, we will use a `for` loop:

```
for i in range(0,20):
    counts[i] = len(np.where((u >= S[i]) & (u < S[i]+0.05))[0])
    V=V+(counts[i]-Ns)**2 / Ns
```

First, we use the `np.where()` function to count how many values satisfy the following conditions (`(u >= S[i]) & (u < S[i]+0.05)`); these are the extremes of each subinterval. Then, we calculate the *V* variable using the following equation:

$$V = \sum_{i=1}^{s} \frac{\left(R_i - \frac{N}{s}\right)^2}{\frac{N}{s}}$$

Finally, we will print the results:

```
print("R = ",counts)
print("V = ", V)
```

The following results are returned:

```
R =  [8 3 4 7 4 5 2 3 7 7 5 4 5 2 7 5 5 5 3 9]
V =  14.8
```

Before analyzing the meaning of the calculated `V` value, let's consider the sequence of counts obtained. To appreciate the distribution of the frequencies obtained, we draw a bar graph:

```
import matplotlib.pyplot as plt
Ypos = np.arange(len(counts))
plt.bar(Ypos,counts)
```

The following plot is printed:

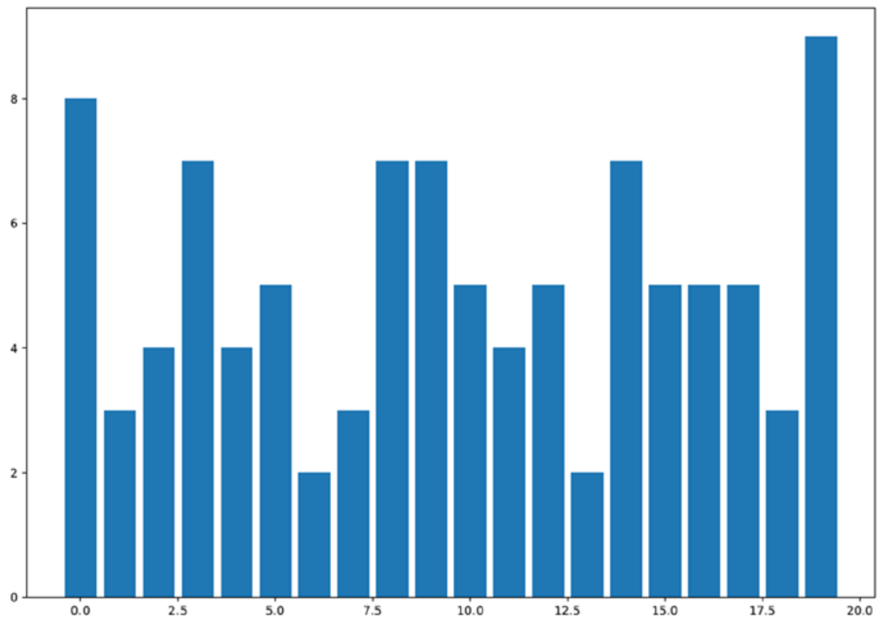


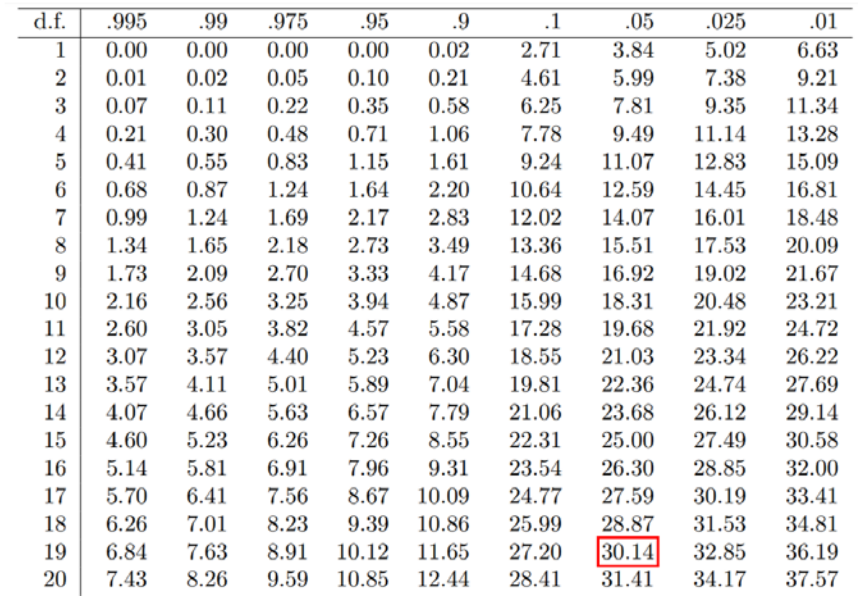
Figure 2.12 – Distribution of frequencies

As you can see, all of the ranges are covered with values ranging from a minimum of 2 to a maximum of 9.

However, now, let's analyze the value of `V` obtained. As we anticipated, `V = 14.8`, so what do we do with this value? First, let's calculate the so-called degrees of freedom, n:

$$n = (2 - 1) * (20 - 1) = 1 * 19 = 19$$

Now, we must compare the `V` value obtained with the probability of exceeding the critical value. To get this value, we must visualize the chi-squared distribution table:



d.f.	.995	.99	.975	.95	.9	.1	.05	.025	.01
1	0.00	0.00	0.00	0.00	0.02	2.71	3.84	5.02	6.63
2	0.01	0.02	0.05	0.10	0.21	4.61	5.99	7.38	9.21
3	0.07	0.11	0.22	0.35	0.58	6.25	7.81	9.35	11.34
4	0.21	0.30	0.48	0.71	1.06	7.78	9.49	11.14	13.28
5	0.41	0.55	0.83	1.15	1.61	9.24	11.07	12.83	15.09
6	0.68	0.87	1.24	1.64	2.20	10.64	12.59	14.45	16.81
7	0.99	1.24	1.69	2.17	2.83	12.02	14.07	16.01	18.48
8	1.34	1.65	2.18	2.73	3.49	13.36	15.51	17.53	20.09
9	1.73	2.09	2.70	3.33	4.17	14.68	16.92	19.02	21.67
10	2.16	2.56	3.25	3.94	4.87	15.99	18.31	20.48	23.21
11	2.60	3.05	3.82	4.57	5.58	17.28	19.68	21.92	24.72
12	3.07	3.57	4.40	5.23	6.30	18.55	21.03	23.34	26.22
13	3.57	4.11	5.01	5.89	7.04	19.81	22.36	24.74	27.69
14	4.07	4.66	5.63	6.57	7.79	21.06	23.68	26.12	29.14
15	4.60	5.23	6.26	7.26	8.55	22.31	25.00	27.49	30.58
16	5.14	5.81	6.91	7.96	9.31	23.54	26.30	28.85	32.00
17	5.70	6.41	7.56	8.67	10.09	24.77	27.59	30.19	33.41
18	6.26	7.01	8.23	9.39	10.86	25.99	28.87	31.53	34.81
19	6.84	7.63	8.91	10.12	11.65	27.20	$\boxed{30.14}$	32.85	36.19
20	7.43	8.26	9.59	10.85	12.44	28.41	31.41	34.17	37.57

Figure 2.13 – The chi-squared distribution table

In the previous table, we look for the value, $n = 19$, in the first column, and then we scroll through the rows until we reach the column that is equal to 0.05. Here, we find the following:

$$\chi^2_{19,0.05} = 30.14$$

If the statistic of the `V` test is less ($14.8 < 30.14$), then we can accept the hypothesis of uniformity of the generated sequence.

Exploring generic methods for random distributions

Most programming languages provide users with functions for the generation of pseudorandom numbers with uniform distributions in the range of [0, 1]. These generators are, very often, considered to be continuous. However, in reality, they are discrete even if they have a very small discretization step. Any sequence of pseudorandom numbers can always be generated from a uniform distribution of random numbers. In the following sections, we will examine some methods that allow us to derive a generic distribution starting from a uniform distribution of random numbers.

The inverse transform sampling method

By having a PRNG with continuous and uniform distributions in the range of [0, 1], it is possible to generate continuous sequences with any probability distribution using the inverse transform sampling technique. Consider a continuous random variable, *x*, having a probability density function of *f(x)*. The corresponding distribution function, *F(x)*, is determined for this function, as follows:

$$F(x) = \int_0^x f(x) * dx$$

The distribution function, *F(x)*, of a random variable indicates the probability that the variable assumes a value that is less than or equal to *x*. The analytical expression (if any) of the inverse function is then determined, such as *x* = *F* - 1. The determination of the sample of the variable, *x*, is obtained by generating a value between 0 and 1 and replacing it in the expression of the inverse distribution function.

This method can be used to obtain samples from many types of distribution functions, such as exponential, uniform, or triangular. It turns out to be the most intuitive, but not the most computationally effective, method.

Let's proceed by starting with a decreasing exponential distribution:

$$f(x) = \lambda * e^{-\lambda x}, \qquad \lambda > 0$$

The corresponding distribution function, *F(x)*, is determined for this function, as follows:

$$F(x) = \int_0^\infty \lambda * e^{-\lambda x} * dx = 1$$

The trend of the decreasing exponential distribution function is shown in the following plot:

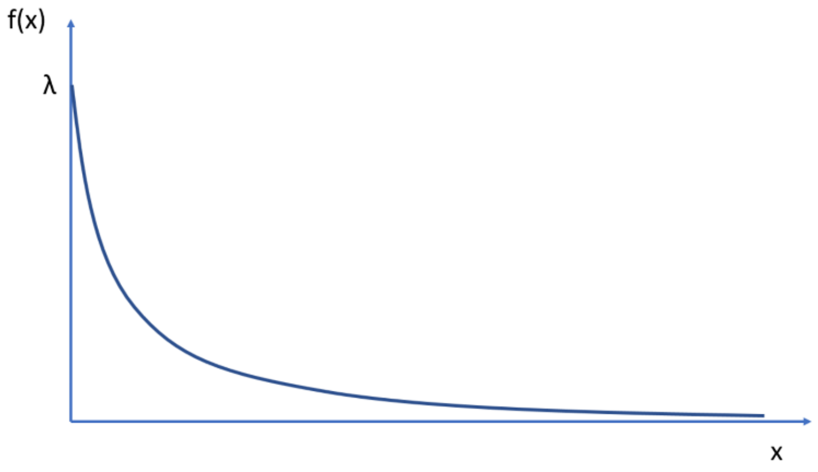


Figure 2.14 – Representation of the decreasing exponential function

Get the distribution function by solving the integral:

$$F(x) = \int_0^{\infty} \lambda * e^{-\lambda x} * dx \; = 1 - e^{-\lambda x} = r$$

By operating the inverse transformation of the distribution function, we get the following:

$$x = -\frac{1}{\lambda} * ln(1-r)$$

Here, r is within the range of [0 ÷ 1]. r is extracted from a uniform distribution by means of a uniform generator. λ represents the average inter-arrival frequency if x represents the time, and $1/\lambda$ is the average inter-arrival time.

The method of the inverse transformation in the discrete case has a very intuitive justification. The interval, [0, 1], is divided into contiguous subintervals of amplitude p (x1), p (x2),. . . and X is assigned according to whether these intervals contain the U that is being generated.

The acceptance-rejection method

The inverse transformation method is based on the calculation of the inverse transformation, F – 1, which cannot always be calculated (or, at least, not efficiently). In the case of law distributions defined on finite intervals [a, b], the rejection-acceptance method is used.

Suppose we know the probability density of the random variable, X, that we intend to generate: $f_X(x)$. This is defined on a finite interval, $[a, b]$, and the image is defined in the range of $[0, c]$. In practice, the $f_X(x)$ function is entirely contained within the rectangle, $[a, b]$ x $[0, c]$, as shown in the following plot:

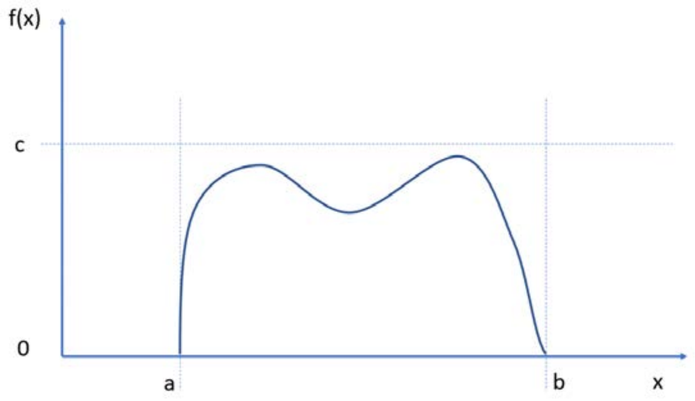


Figure 2.15 – Representation of the f(x) function

We generate two uniform pseudorandom sequences between [0, 1]: U_1 and U_2. Next, we derive two other uniform numerical sequences according to the following rule:

$$\begin{cases} X = a + (b - a) * U_1 \\ Y = c * U_2 \end{cases}$$

Each pair of values (U_1, U_2) will correspond to a pair (x, y) belonging to the rectangle, $[a, b]$ x $[0, c]$. If the pair (x, y) falls within the area of the function, $f_X(x)$, it is accepted and will subsequently be used to create the desired pseudorandom sequence; otherwise, it will be discarded. In the latter case, the procedure is repeated until a new pair located in the area of $f_X(x)$ is found. The sequence of X values that is obtained is a pseudorandom sequence that follows the distribution law, $f_X(x)$, because we have chosen only values that fall in that area.

Random number generation using Python

So far, we have seen what methods can be used for generating random numbers. We have also proposed some solutions in Python code for the generation of random numbers through some universally used methods. These applications have been useful for understanding the basis on which random number generators have been made. In Python, there is a specific module for the generation of random numbers: this is the **random module**. Let's examine what it is.

Introducing the random module

The random module implements PRNGs for various distributions. The random module is based on the Mersenne Twister algorithm, which was originally developed to produce inputs for Monte Carlo simulations. The Mersenne Twister algorithm is a PRNG that produces almost uniform numbers suitable for a wide range of applications.

It is important to note that random numbers are generated using repeatable and predictable deterministic algorithms. They begin with a certain seed value and, every time we ask for a new number, we get one based on the current seed. The seed is an attribute of the generator. If we invoke the generator twice with the same seed, the sequence of numbers that will be generated starting from that seed will always be the same. However, these numbers will be evenly distributed.

Let's analyze, in detail, the functions contained in the module through a series of practical examples.

The random.random() function

The `random.random()` function returns the next nearest floating-point value from the generated sequence. All return values are enclosed between `0` and `1.0`. Let's explore a practical example that uses this function:

```
import random
for i in range(20):
    print('%05.4f' % random.random(), end=' ')
print()
```

We first imported the random module and then we used a `for` loop to generate 20 pseudorandom numbers. Each number is printed in a format that includes 5 digits, including 4 decimal places.

The following results are returned:

```
0.7916 0.2058 0.0654 0.6160 0.1003 0.3985 0.3573 0.9567 0.0193
0.4709 0.8573 0.2533 0.8461 0.1394 0.4332 0.7084 0.7994 0.3361
0.1639 0.4528
```

As you can see, the numbers are uniformly distributed in the range of [0, 1].

By running the code repeatedly, you get sequences of different numbers. Let's try the following:

```
0.6918 0.8197 0.4329 0.2674 0.4118 0.1937 0.2267 0.8259 0.9081
0.4583 0.7300 0.7148 0.9814 0.2237 0.7419 0.7766 0.2626 0.1886
0.1328 0.0037
```

We have confirmed that every time we invoke the `random()` function, the generated sequence is different from the previous one.

The random.seed() function

As we have verified, the `random()` function produces different values each time it is invoked and has a very large period before any number is repeated. This is useful for producing unique values or variations, but there are times when it is useful to have the same set of data available to be processed in different ways. To do this, we can use the `random.seed()` function. This function initializes the basic random number generator. Let's look at an example:

```
import random
random.seed(1)
for i in range(20):
    print('%05.4f' % random.random(), end=' ')
print()
```

We used the same code from the previous example. However, this time, we set the seed (`random.seed(1)`). The number in parentheses is an optional argument and can be any object whose hash can be calculated. If this argument is omitted, the current system time is used. The current system time is also used to initialize the generator when the module is imported for the first time.

The following results are returned:

```
0.1344 0.8474 0.7638 0.2551 0.4954 0.4495 0.6516 0.7887 0.0939
0.0283 0.8358 0.4328 0.7623 0.0021 0.4454 0.7215 0.2288 0.9453
0.9014 0.0306
```

Let's see what happens if we launch this piece of code again:

```
0.1344 0.8474 0.7638 0.2551 0.4954 0.4495 0.6516 0.7887 0.0939
0.0283 0.8358 0.4328 0.7623 0.0021 0.4454 0.7215 0.2288 0.9453
0.9014 0.0306
```

The result is similar. The seed setting is particularly useful when you want to make the simulation repeatable.

The random.uniform() function

The `random.uniform()` function generates numbers within a defined numeric range. Let's look at an example:

```
import random
for i in range(20):
    print('%6.4f' % random.uniform(1, 100), end=' ')
print()
```

We asked it to generate 20 random numbers in the range of [1, 100). The following results are returned:

```
26.2741 84.3327 67.6382 9.2402 2.6524 2.4414 75.8031 25.7064
11.8394 62.8554 35.0979 7.8820 16.8029 53.2107 17.6463 28.0185
71.4474 46.0155 32.8782 47.9033
```

This function can be used when requesting random numbers in well-defined intervals.

The random.randint() function

This function generates random integers. The arguments for `randint()` are the values of the range, including the extremes. The numbers may be negative or positive, but the first value should be less than the second. Let's look at an example:

```
import random
for i in range(20):
    print(random.randint(-100, 100), end=' ')
print()
```

The following results are returned:

```
9 -85 88 -24 -68 -46 -88 -22 -82 -81 -21 -24 90 -60 6 44 -36
-67 -98 43
```

The entire range is represented in the sequence of randomly generated numbers.

A more generic form of selecting values from a range is obtained by using the `random.range()` function. In this case, the step argument is provided, in addition to the start and end values. Let's look at an example:

```
import random
for i in range(20):
    print(random.randrange(0, 100,5), end=' ')
print()
```

The following results are returned:

```
5 90 30 90 70 25 95 80 5 60 30 55 15 30 90 65 90 30 75 15
```

The returned sequence is a random distribution of the values expected from the passed arguments.

The random.choice() function

A common use for random number generators is to select a random element from a sequence of enumerated values, even if these values are not numbers. The `choice()` function returns a random element of the non-empty sequence passed as an argument:

```
import random
CitiesList = ['Rome','New York','London','Berlin','Moskov',
'Los Angeles','Paris','Madrid','Tokio','Toronto']
for i in range(10):
    CitiesItem = random.choice(CitiesList)
    print ("Randomly selected item from Cities list is - ",
CitiesItem)
```

The following results are returned:

```
Randomly selected item from Cities list is -  Paris
Randomly selected item from Cities list is -  Moskov
Randomly selected item from Cities list is -  Tokio
Randomly selected item from Cities list is -  Madrid
Randomly selected item from Cities list is -  Rome
Randomly selected item from Cities list is -  Los Angeles
Randomly selected item from Cities list is -  Toronto
Randomly selected item from Cities list is -  Paris
Randomly selected item from Cities list is -  Moskov
Randomly selected item from Cities list is -  Rome
```

At each iteration of the cycle, a new element is extracted from the list containing the names of the cities. This function is suitable to use in extracting values from a predetermined list.

The random.sample() function

Many simulations require random samples from a population of input values. The `random.sample()` function generates samples without repeating the values and without changing the input sequence. Let's look at an example:

```
import random
DataList = range(10,100,10)
print("Initial Data List = ",DataList)
DataSample = random.sample(DataList,k=5)
print("Sample Data List = ",DataSample)
```

The following results are returned:

```
Initial Data List =  range(10, 100, 10)
Sample Data List =  [30, 60, 40, 20, 90]
```

Only five elements of the initial list were selected, and this selection was completely random.

Generating real-valued distributions

The following functions generate specific distributions of real numbers:

- `betavariate` (alpha, beta): This is the beta distribution. The conditions for the parameters are `alpha`> -1 and `beta`> -1. Return values are in the range of 0 to 1.
- `expovariate` (`Lambd`): This is the exponential distribution. `lambd` is 1.0 divided by the desired average. (The parameter was supposed to be called "lambda," but this is a reserved word in Python.) The return value is between 0 and positive infinity.
- `gammavariate` (`alpha`, `beta`): This is the gamma distribution. The conditions on the parameters are `alpha`> 0 and `beta`> 0.

- `gauss (mu, sigma)`: This is the Gaussian distribution. mu is the mean and sigma is the standard deviation. This is slightly faster than the `normalvariate()` function defined next.
- `lognormvariate (mu, sigma)`: This is the normal logarithmic distribution. If you take the natural logarithm of this distribution, you will get the normal distribution with mean `mu` and `sigma` standard deviation. mu can have any value, while `sigma` must be greater than zero.
- `normalvariate (mu, sigma)`: This is the normal distribution. `mu` is the mean and `sigma` is the standard deviation.
- `vonmisesvariate (mu,kappa)`: mu is the average angle, expressed in radians with a value between 0 and 2 * pi, while `kappa` is the concentration parameter, which must be greater than or equal to zero. If `kappa` is equal to zero, this distribution narrows to a constant random angle in a range between 0 and 2 * pi.
- `paretovariate (alpha)`: This is the Pareto distribution. alpha is the form parameter.
- `weibullvariate (alpha,beta)`: This is the Weibull distribution. Here, `alpha` is the scale parameter and `beta` is the form parameter.

Summary

In this chapter, we learned how to define stochastic processes and understand the importance of using them to address numerous real-world problems. For instance, the operation of slot machines is based on the generation of random numbers, as are many complex data encryption procedures. Next, we introduced the concepts behind random number generation techniques. We explored the main methods of generating random numbers using practical examples in Python code. The generation of uniform and generic distributions was discussed. We also learned how to perform a uniformity test using the chi-squared method. Finally, we looked at the main functions available in Python for generating random numbers: `random`, `seed`, `uniform`, `randint`, `choice`, and `sample`.

In the next chapter, we will learn the basic concepts of probability theory. Additionally, we will learn how to calculate the probability of an event happening after it has already occurred, and then we will learn how to work with discrete and continuous distributions.

3
Probability and Data Generation Processes

The field of **probability calculation** was born in the context of gambling. It was then developed further, assuming a relevant role in the analysis of collective phenomena and becoming an essential feature of statistics and statistical decision theory. Probability calculation is an abstract and highly formalized mathematical discipline, while maintaining relevance to its original and pertinent empirical context. The concept of probability is strongly linked to that of uncertainty. The probability of an event can, in fact, be defined as the quantification of the level of randomness of that event. What is not known or cannot be predicted with an absolute level of certainty is known as being **random**. In this chapter, we will learn how to distinguish between the different definitions of probabilities and how these can be integrated to obtain useful information in the simulation of real phenomena.

In this chapter, we're going to cover the following main topics:

- Explaining probability concepts
- Understanding Bayes' theorem
- Probability distributions

Technical requirements

In this chapter, an introduction to theory of probability will be discussed. To deal with these topics, it is necessary that you have a basic knowledge of algebra and mathematical modeling.

To install a library not contained in your Python environment, use the `pip install` command. To work with the Python code in this chapter, you need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `UniformDistribution.py`
- `BinomialDistribution.py`
- `NormalDistribution.py`

Explaining probability concepts

If we take a moment to reflect, we'll notice that our everyday lives are full of **probabilistic** considerations, although not necessarily formalized as such. Examples of probabilistic assessments include choosing to participate in a competition given the limited chance of winning, the team's predictions of winning the championship, statistics that inform us about the probability of death from smoking or failure to use seat belts in the event of a road accident, and the chances of winning in games and lotteries.

In all situations of uncertainty, there is basically a tendency to give a measure of uncertainty that, although indicated in various terms, expresses the intuitive meaning of probability. The fact that probability has an intuitive meaning also means that establishing its rules can, within certain limits, be guided by intuition. However, relying completely on intuition can lead to incorrect conclusions. To avoid reaching incorrect conclusions, it is necessary to formalize the calculation of probabilities by establishing their rules and concepts in a logical and rigorous way.

Types of events

We define an **event** as any result to which, following an experiment or an observation, a well-defined degree of truth can be uniquely assigned. In everyday life, some events happen with certainty, while others never happen. For example, if a box contains only yellow marbles, by extracting one at random, we are sure that it will be yellow, while it is impossible to extract a red ball. We call the events of the first type – that is, extracting a yellow marble – **certain events**, while those of the second type – that is, extracting a red marble – **impossible events**.

To these two types of events – certain and impossible – are events that can happen, but without certainty. If the box contains both yellow and red balls, then extracting a yellow ball is a possible but not certain event, as is extracting a red ball. In other words, we cannot predict the color of the extracted ball because the extraction is random.

Something that may or may not happen at random is called a **random event**. In *Chapter 2, Understanding Randomness and Random Numbers*, we introduced random events. An example of such a random event is being selected in chemistry to check homework over a week's worth of lessons.

The same event can be certain, random, or impossible, depending on the context in which it is considered. Let's analyze an example: winning the Mega Millions jackpot game. This event can be considered certain if we buy all the tickets of the game; it is impossible if we do not buy even one; and it is random if we buy one or more than one, but not all.

Calculating probability

The succession of random events has led people to formulate bets on their occurrence. The concept of probability was born precisely because of gambling. 3,000 years ago, the Egyptians played an ancestor of the dice game. The game of dice was widespread in ancient Rome too, so much so that some studies have found that this game dates back to the age of Cicero. But the birth of the systematic study of the calculation of probabilities dates back to 1654, by the mathematician and philosopher Blaise Pascal.

Probability definition with an example

Before we analyze some simple examples of calculating the probability of the occurrence of an event, it is good to define the concept of probability. To start, we must distinguish between a classical approach to the definition of probability and the **frequentist** point of view.

A priori probability

The **a priori probability** *P(E)* of a random event *E* is defined as the ratio between the number *s* of the favorable cases and the number *n* of the possible cases, which are all considered equally probable:

$$P(E) = \frac{\text{number of the favorable cases}}{\text{number of the possible cases}} = \frac{s}{n}$$

In a box, there are 14 yellow marbles and six red marbles. The marbles are similar in every way except for their color; they're made of the same material, same size, perfectly spherical, and so on. We'll put a hand into the box without looking inside, pulling out a random marble. What is the probability that the pulled-out marble is red?

In total, there are 14 + 6 = 20 marbles. By pulling out a marble, we have 20 possible cases. We have no reason to think that some marbles are more privileged than others; that is, they are more likely to be pulled out. Therefore, the 20 possible cases are equally probable.

Of these 20 possible cases, there are only six cases in which the marble being pulled out is red. These are the cases that are favorable to the expected event.

Therefore, the red marble being pulled out has six out of 20 possible occurrences. Defining its probability as the ratio between the favorable and possible cases, we will get the following:

$$P(E) = \mathrm{P(red\ marble\ pulled-out)} = \frac{6}{20} = 0.3 = 30\%$$

Based on the definition of probability, we can say the following:

- The probability of an impossible event is 0
- The probability of a certain event is 1
- The probability of a random event is between 0 and 1

Previously, we introduced the concept of equally probable events. Given a group of events, if there is no reason to think that some event occurs more frequently than others, then all group events should be considered equally likely.

Complementary events

Complementary events are two events – usually referred to as E and $\bar{E}$ – that are mutually exclusive.

For example, when rolling some dice, we consider the event as E = number 5 comes out.

The complementary event will be $\bar{E}$ = number 5 does not come out.

E and $\bar{E}$ are mutually exclusive because the two events cannot happen simultaneously; they are exhaustive because the sum of their probabilities is 1.

For event E, there are 1 (5) favorable cases, while for event $\bar{E}$, there are 5 favorable cases; that is, all the remaining cases (1, 2, 3, 4, 6). So, the a priori probability is as follows:

$$P(E) = \frac{1}{6}\ ; P(\bar{\mathrm{E}}) = \frac{5}{6}$$

Due to this, we can observe the following:

$$P(E) + P(\bar{E}) = \frac{1}{6} + \frac{5}{6} = 1$$

Relative frequency and probability

However, the classical definition of probability is not applicable to all situations. To affirm that all cases are equally probable is to make an a priori assumption about their probability of occurring, thus using the same concept in the definition that you want to define.

The relative frequency *f(E)* of an event subjected to *n* experiments, all carried out under the same conditions, is the ratio between the number *v* of the times the event occurred and the number *n* of tests carried out:

$$f(E) = \frac{v}{n}$$

If we consider the toss of a coin and the event *E* = heads up, classical probability gives us the following value:

$$P(E) = \frac{1}{2}$$

If we perform many throws, we will see that the number of times the coin landed heads up is almost equal to the number of times a cross occurs. That is, the relative frequency of the event *E* approaches the theoretical value:

$$f(E) \cong P(E) = \frac{1}{2}$$

Given a random event *E*, subjected to *n* tests performed all under the same conditions, the value of the relative frequency tends to the value of the probability as the number of tests carried out increases.

Important Note

The probability of a repeatable event coincides with the relative frequency of its occurrence when the number of tests being carried out is sufficiently high.

Note that in the classical definition, the probability is evaluated a priori, while the frequency is a value that's evaluated posteriori.

The frequency-based approach is applied, for example, in the field of insurance to assess the average life span of an individual, the probability of theft, and the probability of accidents. It can also be applied in the field of medicine in order to evaluate the probability of contracting a certain disease, or the probability that a drug is effective. In all these events, the calculation is based on what has happened in the past; that is, by evaluating the probability by calculating the relative frequencies.

Let's now look at another approach we can use to calculate probabilities that estimates the levels of confidence in the occurrence of a given event.

Understanding Bayes' theorem

From the Bayesian point of view, probability measures the degree of likelihood that an event will occur. It is an inverse probability in the sense that from the observed frequencies, we obtain the probability. Bayesian statistics foresee the calculation of the probability of a certain event before carrying out the experiment; this calculation is made based on previous considerations. Using Bayes' theorem, by using the observed frequencies, we can calculate the a priori probability, and from this, we can determine the posterior probability. By adopting this method, the prediction of the degree of credibility of a given hypothesis is used before observing the data, which is then used to calculate the probability after observing the data.

Important Note

In the frequentist approach, we determine how often the observation falls in a certain interval, while in the Bayesian approach, the probability of truth is directly attributable to the interval.

In cases where a frequentist result exists within the limit of a very large sample, the Bayesian and frequentist results coincide. There are also cases where the frequentist approach is not applicable.

Compound probability

Now, consider two events, E_1 and E_2, where we want to calculate the probability $P(E_1 \cap E_2)$ that both occur. Two cases can occur:

- E_1 and E_2 are stochastically independent
- E_1 and E_2 are stochastically dependent

The two events, E_1 and E_2, are stochastically independent if they do not influence each other, that is, if the occurrence of one of the two does not change the probability of the second occurring. Conversely, the two events, E_1 and E_2, are stochastically dependent if the occurrence of one of the two changes the probability of the second occurring.

Let's look at an example: You draw a card from a deck of 40 that contains the numbers 1-7, plus the three face cards for each suit. What is the probability that it is a face card and from the hearts suit?

To start, we must ask ourselves whether the two events are dependent or independent.

There are 12 faces, three for each symbol, so the probability of the first event is equal to 12/40, that is, 3/10. The probability that the card is from the hearts suit is not influenced by the occurrence of the event that the card is a face; therefore, it is worth 10/40, that is, 1/4. Therefore, the compound probability will be 3/40.

Therefore, this is a case of independent events. The **compound probability** is given by the product of the probabilities of the individual events, as follows:

$$P(E_1 \cap E_2) = P(E_1) * P(E_2) = \frac{3}{10} * \frac{1}{4} = \frac{3}{40}$$

Let's look at a second example: We draw a card from a deck of 40 and, without putting it back in the deck, we draw a second one. What is the probability that they are two queens?

The probability of the first event is 4/40, that is, 1/10. But when drawing the second card, there's only 39 remaining, and there's only three queens. So, the probability that the second card is still a queen will have become 3/39, that is, 1/13. Therefore, the compound probability will be given by the product of the probability that the first card is a queen for the probability that the second is still a queen, that is, 1/130.

Thus, this is a case of dependent events; that is, the probability of the second event is conditioned by the occurrence of the first event. Similarly, the two events are considered dependent if the two cards are drawn simultaneously, when there is no reintegration.

When the probability of an E_2 event depends on the occurrence of the E_1 event, we speak of the **conditional probability**, which is denoted by $P(E_2 \mid E_1)$, and we see that the probability of E_2 is conditional on E_1.

When the two events are stochastically dependent, the compound probability is given by the following equation:

$$P(E_1 \cap E_2) = P(E_1) * P(E_2 \mid E_1) = \frac{1}{10} * \frac{1}{13} = \frac{1}{130}$$

From the previous equation, we can derive the equation that gives us the conditional probability:

$$P(E_2 \mid E_1) = \frac{P(E_1 \cap E_2)}{P(E_1)}$$

After defining the concept of conditional probability, we can move on and analyze the heart of Bayesian statistics.

Bayes' theorem

Let's say that E_1 and E_2 are two dependent events. In the *Compound probability* section, we learned that the compound probability between the two events is calculated using the following equation:

$$P(E_1 \cap E_2) = P(E_1) * P(E_2 \mid E_1)$$

By exchanging the order of succession of the two events, we can write the following equation:

$$P(E_1 \cap E_2) = P(E_2) * P(E_1 \mid E_2)$$

The left-hand part of the two previous equations contain the same quantity, which must also be true for the right part. Based on this consideration, we can write the following equation:

$$P(E_2|E_1) = \frac{P(E_2) * P(E_1|E_2)}{P(E_1)}$$

The same is true by exchanging the order of events:

$$P(E_1|E_2) = \frac{P(E_1) * P(E_2|E_1)}{P(E_2)}$$

The preceding equations represent the mathematical formulation of Bayes' theorem. The use of one or the other depends on the purpose of our work. Bayes' theorem is derived from two fundamental probability theorems: the compound probability theorem and the total probability theorem. It is used to calculate the probability of a cause that triggered the verified event.

In Bayes' theorem, we know the result of the experiment and we want to calculate the probability that it is due to a certain cause. Let's analyze the elements that appear in the equation that formalizes Bayes' theorem in detail:

$$P(E_2|E_1) = \frac{P(E_2) * P(E_1|E_2)}{P(E_1)}$$

Here, we have the following:

- $P(E_2|E_1)$ is called **posterior probability** (what we want to calculate)
- $P(E_2)$ is called **prior probability**
- $P(E_1|E_2)$ is called **likelihood** (represents the probability of observing the E_1 event when the correct hypothesis is E_2)
- $P(E_1)$ is called **marginal likelihood**

Bayes' theorem applies to many real-life situations, such as in the medical field for finding false positives in one analysis, or to verify the effectiveness of a drug.

Now, let's learn how to represent the probabilities of possible results in an experiment.

Exploring probability distributions

A **probability distribution** is a mathematical model that links the values of a variable to the probabilities that these values can be observed. Probability distributions are used to model the behavior of a phenomenon of interest in relation to the reference population, or to all the cases of which the researcher observes a given sample.

Based on the measurement scale of the variable of interest *X*, we can distinguish two types of probability distributions:

- **Continuous distributions**: The variable is expressed on a continuous scale
- **Discrete distributions**: The variable is measured with integer numerical values

In this context, the variable of interest is seen as a random variable whose probability law expresses the degree of uncertainty with which its values can be observed. Probability distributions are expressed by a mathematical law called **probability density function** (*f(x)*) or **probability function** (*p(x)*) for continuous or discrete distributions, respectively. The following diagram shows a continuous distribution (to the left) and a discrete distribution (to the right):

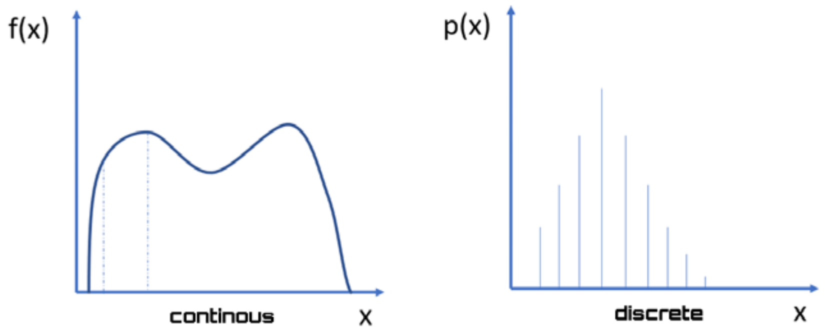


Figure 3.1 – A continuous distribution and a discrete distribution

To analyze how a series of data is distributed, which we assume can take any real value, it is necessary to start with the definition of the probability density function. Let's see how that works.

Probability density function

The **Probability Density Function (PDF)** *P(x)* represents the probability *p(x)* that a given *x* value of the continuous variable is contained in the interval (*x*, *x* + *Δx*), divided by the width of the interval *Δx*, when this tends to be zero:

$$P(x) = \lim_{\Delta x \to 0} \frac{p(x + \Delta x) - p(x)}{\Delta x} = \frac{dp(x)}{dx}$$

The probability of finding a given *x* value in the interval [*a*, *b*] is given by the following equation:

$$p(x) = \int_a^b P(x)dx$$

Since *x* takes a real value, the following property holds:

$$\int_{-\infty}^{+\infty} P(x)dx = 1$$

In practice, we do not have an infinite set of real values, but rather a discrete set *N* of real numbers x_i. Then, we proceed by dividing the interval [x_{min}, x_{max}] into a certain number N_c of subintervals (bins) of amplitude *Δx*, considering the definition of probability as the ratio between the number of favorable cases and the number of possible cases.

The calculation for the PDF refers to dividing the interval [x_{min}, x_{max}] into N_c subintervals and counting how many x_i values fall into each of these subintervals, before dividing each value by *Δx * N*, as shown in the following equation:

$$P(x) = \frac{n_i}{\Delta x * N}$$

Here, we can see the following:

- *P(x)* is the PDF
- n_i is the number of *x* values that fall in the *i-th* sub-interval
- Δx is the amplitude of each sub-interval
- *N* is the number of observations *x*

Now, let's learn how to determine the probability distribution of a variable in the Python environment.

Mean and variance

The expected value, which is also called the average of the distribution of a random variable, is a position index. The expected value of a random variable represents the expected value that can be obtained with a sufficiently large number of tests so that it is possible to predict, by probability, the relative frequencies of the various events.

The expected value of a discrete random variable, if the distribution is finite, is a real number given by the sum of the products of each value of the random variable for the respective probability:

$$E(x) = x_1p_1 + x_2p_2 + \cdots + x_np_n = \sum_{i=1}^{n} x_ip_i$$

The expected value is, therefore, a weighted sum of the values that the random variable assumes when weighted with the associated probabilities. Due to this, it can be either negative or positive.

After the expected value, the most used parameter to characterize the probability distributions of the random variables is the variance, which indicates how scattered the values of the random variable are relative to its average value.

Given a random variable *X*, whatever *E(X)* is its expected value. Consider the random variable *X–E(X)*, whose values are the distances between the values of *X* and the expected value *E(X)*. Substituting a variable *X* for the variable *X–E(X)* is equivalent to translating the reference system that brings the expected value to the origin of the axes.

The variance of a discrete random variable *X*, if the distribution is finite, is calculated with the following equation:

$$\sigma^2 = \sum_{i=1}^{n} (x_i - E(x))^2 * p_i$$

The variance is equal to zero when all the values of the variable are equal and therefore there is no variability in the distribution; in any case, it is positive and measures the degree of variability of a distribution. The greater the variance, the more scattered the values are. The smaller the variance, the more the values of *X* are concentrated around the average value.

Uniform distribution

The simplest of the continuous variable probability distribution functions is the one in which the same degree of confidence is assigned to all the possible values of a variable defined in a certain range. Since the probability density function is constant, the distribution function is linear. The uniform distribution is used to treat measurement errors whenever they occur with certainty that a certain variable is contained in a certain range, but there is no reason to believe some values are more plausible than others. Using suitable techniques, starting from a uniformly distributed variable, it is possible to build other variables that have been distributed at will.

Now, let's start practicing using it. We will start by generating a uniform distribution of random numbers contained in a specific range. To do this, we will use the `numpy random.uniform()` function. This function generates random values uniformly distributed over the half-open interval *[a, b)*; that is, it includes the first, but excludes the second. Any value within the given interval is equally likely to be drawn by uniform distribution:

1. To start, we import the necessary libraries:

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

`numpy` is a Python library that contains numerous functions that help us manage multidimensional matrices. Furthermore, it contains a large collection of high-level mathematical functions that we can perform on these matrices.

`matplotlib` is a Python library for printing high-quality graphics. With `matplotlib`, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and so on with a few commands. This is a collection of command-line functions like those provided by the MATLAB software.

2. After this, we define the extremes of the range and the number of values we want to generate:

```
a=1
b=100
N=100
```

Now, we can generate the uniform distribution using the `random.uniform()` function, as follows:

```
X1=np.random.uniform(a,b,N)
```

With that, we can view the numbers that we generated. To begin, draw a diagram in which we report the values of the 100 random numbers that we have generated:

```
plt.plot(X1)
plt.show()
```

The following graph will be output:

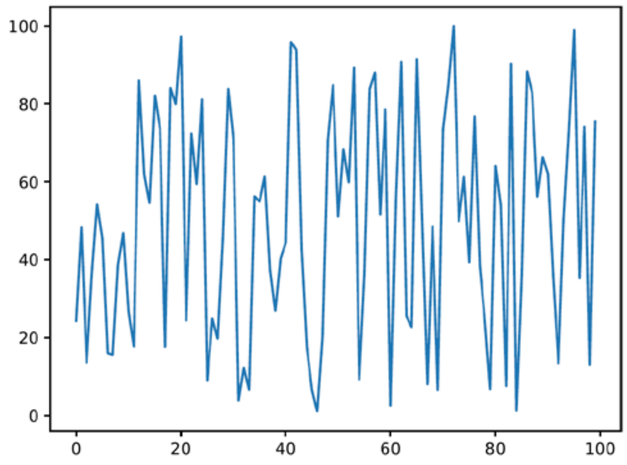


Figure 3.2 – Diagram plotting the 100 numbers

3. At this point, to analyze how the generated values are distributed in the interval considered, we draw a graph of the probability density function:

```
plt.figure()
plt.hist(X1, density=True, histtype='stepfilled',
alpha=0.2)
plt.show()
```

The `matplotlib.hist()` function draws a histogram, that is, a diagram of a continuous character shown in classes. It is used in many contexts, usually to show statistical data, when there is an interval of the definition of the independent variable divided into subintervals. These subintervals can be intrinsic or artificial, can be of equal or unequal amplitude, and are or can be considered constant. Each of these can either be an independent or dependent variable. Each rectangle has a non-random length equal to the width of the class it represents. The height of each rectangle is equal to the ratio between the absolute frequency associated with the class and the amplitude of the class, and it can be defined as **frequency density**. The following four parameters are passed:

- `X1`: Input values.
- `density=True`: This is a bool which, if `True`, makes the function return the counts normalized to form a probability density.
- `histtype='stepfilled'`: This parameter defines the type of histogram to draw. The `stepfilled` value generates a line plot that is filled by default.
- `alpha=0.2`: This is a float value that defines the characteristics of the content (0.0 is transparent and 1.0 is opaque).

The following graph will be output:

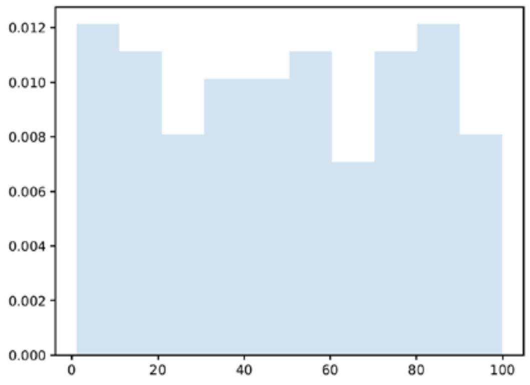


Figure 3.3 – Graph plotting the generated values

Here, we can see that the generated values are distributed almost evenly throughout the range. What happens if we increase the number of generated values?

4. Then, we repeat the commands we just analyzed to modify only the number of samples to be managed. We change this from `100` to `10000`:

```
a=1
b=100
N=10000
X2=np.random.uniform(a,b,N)

plt.figure()
plt.plot(X2)
plt.show()

plt.figure()
plt.hist(X2, density=True, histtype='stepfilled',
alpha=0.2)
plt.show()
```

It is not necessary to reanalyze the piece of code line by line since we are using the same commands. Let's see the results, starting from the generated values:

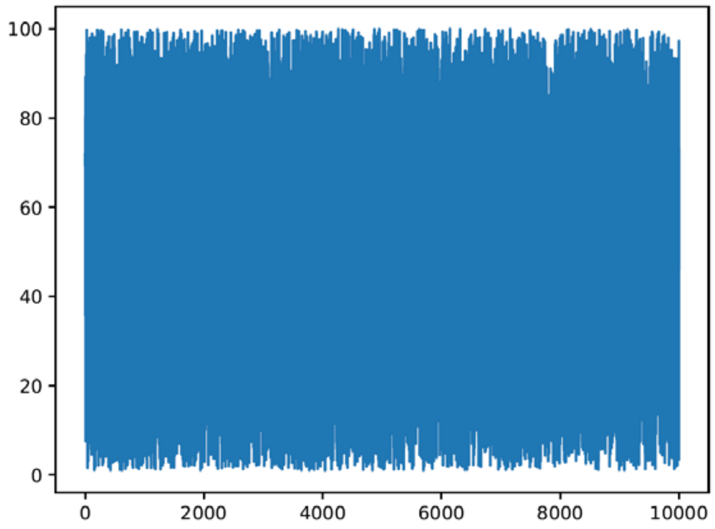


Figure 3.4 – Graph plotting the number of samples

Now, the number of samples that have been generated has increased significantly. Let's see how they are distributed in the range considered:

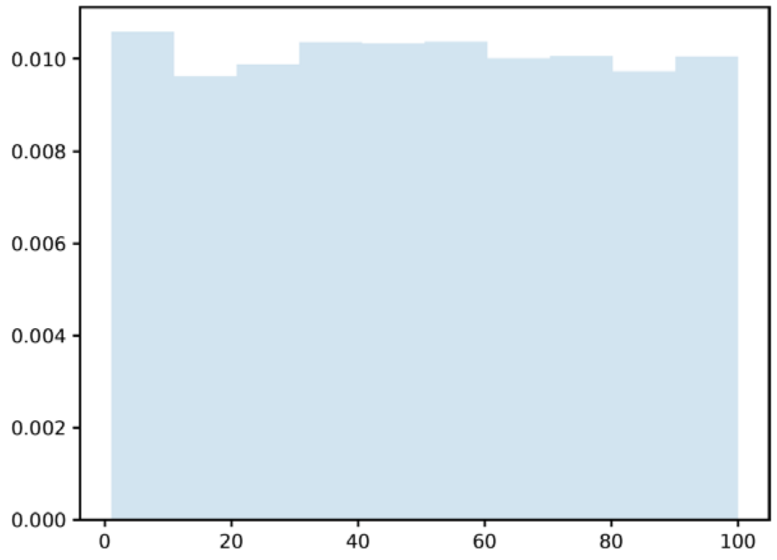


Figure 3.5 – Graph showing the sample distribution

Analyzing the previous histogram and comparing it with what we obtained in the case of `N=100`, we can see that this time, the distribution appears to be flatter. The distribution becomes more and more flat as `N` increases, to increase the statistics in each individual bin.

Binomial distribution

In many situations, we are interested in checking whether a certain characteristic occurs or not. This corresponds to an experiment with only two possible outcomes-also called a dichotomous-that can be modeled with a random variable *X* that assumes value 1 (success) with probability *p* and value 0 (failure) with probability *1-p*, with *0 <p < 1*, as follows:

$$X = \begin{cases} 1, & p \\ 0, & 1-p \end{cases}$$

The expected value and variance of *X* are calculated as follows:

$$E(X) = 0 * (1-p) + 1 * p = p$$

$$\sigma^2 = E(X^2) - \left(E(X)\right)^2 = (0^2 * (1-p) + 1^2 * p) - p^2 = p * (1-p)$$

The binomial distribution is the probability of obtaining x successes in n independent trials. The probability density for the binomial distribution is obtained using the following equation:

$$P_x = \binom{n}{x} p^x q^{n-x}, 0 \leq x \leq n$$

Here, we have the following:

- P_x is the probability density
- n is the number of independent experiments
- x is the number of successes
- p is the probability of success
- q is the probability of fail

Now, let's look at a practical example. We throw a dice `n = 10` times. In this case we want to study the binomial variable `x` = number of times a number <= 3 came out. We define the parameters of the problem as follows:

$$n = 10$$
$$0 \leq x \leq n$$
$$p = \frac{3}{6} = 0.5$$
$$q = 1 - p = 0.5$$

We then evaluate the probability density function with Python code, as follows:

1. Let's start as always by importing the necessary libraries:

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

   Now, we set the parameters of the problem:

```
N = 1000
n = 10
p = 0.5
```

   Here, `N` is the number of trials, `n` is the number of independent experiments in each trial, and `p` is the probability of success for each experiment.

2. Now, we can generate the probability distribution:

```
P1 = np.random.binomial(n,p,N)
```

The `numpy random.binomial()` function generates values from a binomial distribution. These values are extracted from a binomial distribution with the specified parameters. The result is a parameterized binomial distribution, in which each value is equal to the number of successes obtained in the `n` independent experiments. Let's take a look at the return values:

```
plt.plot(P1)
plt.show()
```

The following graph is output:

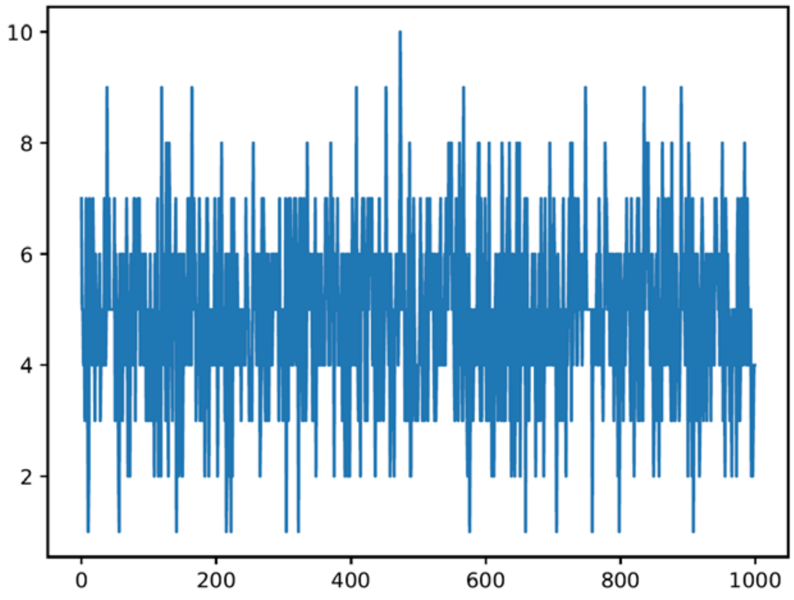


Figure 3.6 – A graph plotting the return values for the binomial distribution

Let's see how these samples are distributed in the range considered:

```
plt.figure()
plt.hist(P1, density=True, alpha=0.8, histtype='bar',
color = 'green', ec='black')
plt.show()
```

This time, we used a higher alpha value to make the colors brighter, we used the traditional bar-type histogram, and we set the color of the bar.

3. Finally, we used the `ec` parameter to set the edge color of each bar. The following results are obtained:

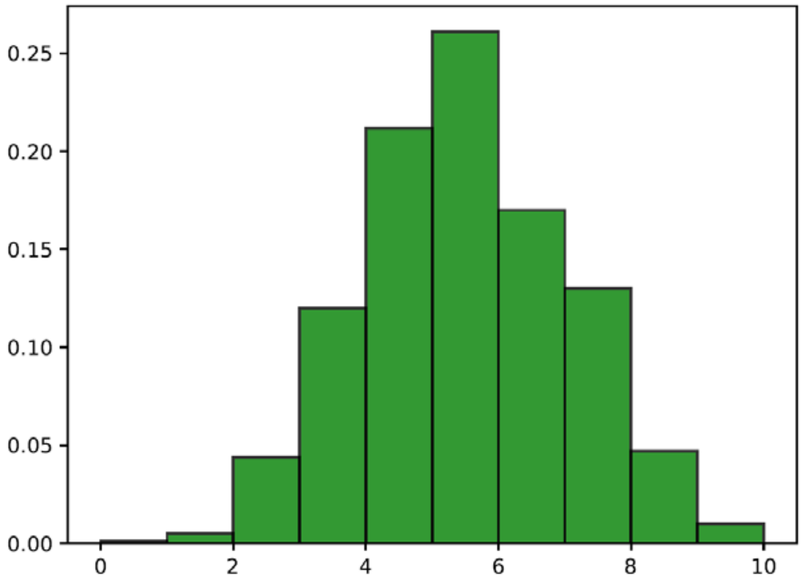


Figure 3.7 – Histogram plotting the return values

All the areas of the binomial distributions, that is, the sum of the rectangles, being the sum of probability, are worth 1.

Normal distribution

As the number of independent experiments that are carried out increases, the binomial distributions approach a curve called the **bell curve** or **Gauss curve**. The **normal distribution**, also called the **Gaussian distribution**, is the most used continuous distribution in statistics. Normal distribution is important in statistics for the following fundamental reasons:

- Several continuous phenomena seem to follow, at least approximately, a normal distribution.
- The normal distribution can be used to approximate numerous discrete probability distributions.
- The normal distribution is the basis of classical statistical inference by virtue of the central limit theorem.

Normal distribution has some important characteristics:

- The normal distribution is symmetrical and bell-shaped.
- Its central position measures – the expected value and the median – coincide.
- Its interquartile range is 1.33 times the mean square deviation.
- The random variable in the normal distribution takes values between -∞ and + ∞.

In the case of a normal distribution, the normal probability density function is given by the following equation:

$$f(X) = \frac{1}{\sqrt{2\pi\sigma}} e^{-(1/2)[(X-\mu)/\sigma]^2}$$

Here, we have the following:

- μ is the expected value.
- σ is the standard deviation.

Note that, since e and π are mathematical constants, the probabilities of a normal distribution depend only on the values assumed by the parameters μ and σ.

Now, let's learn how to generate a normal distribution in Python. Let's start as always by importing the necessary libraries:

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

Here, we have imported a new seaborn library. It is a Python library that enhances the data visualization tools of the `matplotlib` module. In the seaborn module, there are several features we can use to graphically represent our data. There are methods that facilitate the construction of statistical graphs with `matplotlib`.

Now, we set the parameters of the problem. As we've already mentioned, only two parameters are needed to generate a normal distribution: the expected value and the standard deviation. The µ value is also indicated as the center of the distribution and characterizes the position of the curve with respect to the ordinate axis. The σ parameter characterizes the shape of the curve since it represents the dispersion of the values around the maximum of the curve.

To appreciate the functionality of these two parameters, we will generate a normal distribution by changing the values of these parameters, as follows:

```
mu = 10
sigma =2
P1 = np.random.normal(mu, sigma, 1000)

mu = 5
sigma =2
P2 = np.random.normal(mu, sigma, 1000)

mu = 15
sigma =2
P3 = np.random.normal(mu, sigma, 1000)

mu = 10
sigma =2
P4 = np.random.normal(mu, sigma, 1000)

mu = 10
sigma =1
P5 = np.random.normal(mu, sigma, 1000)

mu = 10
sigma =0.5
P6 = np.random.normal(mu, sigma, 1000)
```

For each distribution, we have set the two parameters (μ and σ) and then used the `numpy random.normal()` function to generate a normal distribution. Three parameters are passed: μ, σ, and the number of samples to generate. At this point, it is necessary to view the generated distributions. To do this, we will use the `distplot()` function of the `seaborn` library, as follows:

```
Plot1 = sns.distplot(P1)
Plot2 = sns.distplot(P2)
Plot3 = sns.distplot(P3)
plt.figure()
Plot4 = sns.distplot(P4)
```

```
Plot5 = sns.distplot(P5)
Plot6 = sns.distplot(P6)
plt.show()
```

The `distplot()` function allows us to flexibly plot a univariate distribution of observations. To do this, use the hist function of `matplotlib` and the `kdeplot()` and `rugplot()` functions of seaborn. Let's first analyze the results that were obtained in the first graph:

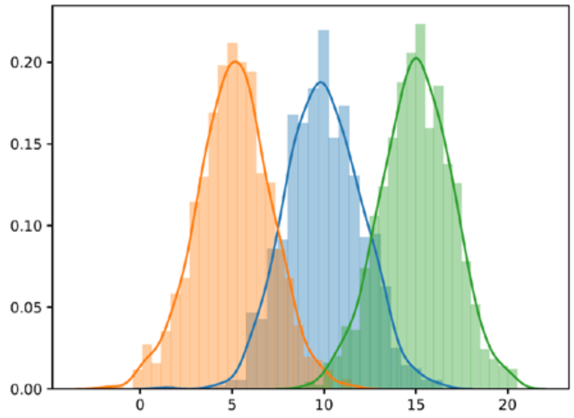


Figure 3.8 – Seaborn plot of the samples

Three curves have been generated that represent the three distributions we have named: P_1, P_2, P_3. The only difference that we can notice lies in the value of μ, which assumes the values 5, 10, 15. Due to the variation of μ, the curve moves along the x-axis, but its shape remains unchanged. Let's now see the graph that represents the remaining distributions:

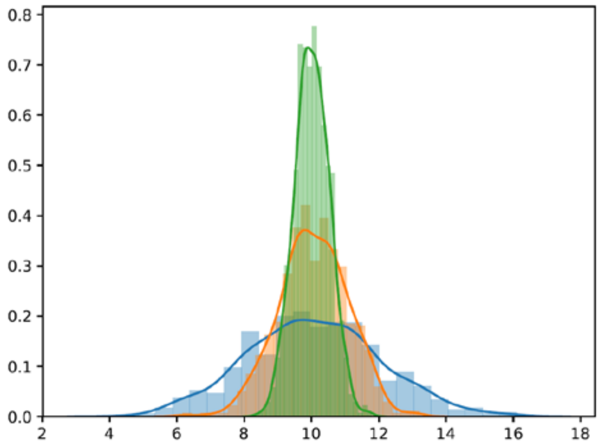


Figure 3.9 – Merged plots

In this case, by keeping the value of μ constant, we have varied the value of σ, which assumes the following values: 2, 1, 0.5. As σ increases, the curve flattens and widens, while as σ decreases, the curve narrows and rises.

A specific normal distribution is one that's obtained with $\mu = 0$ and $\sigma = 1$. This distribution is called the **standardized normal distribution**.

Now that we've seen all the relevant kinds of probability distribution, let's recap what we covered in this chapter.

Summary

Knowing the basics of probability theory in depth helps us to understand how random phenomena work. We discovered the differences between a priori, compound, and conditioned probabilities. We have also seen how Bayes' theorem allows us to calculate the conditional probability of a cause of an event, starting from the knowledge of the a priori probabilities and the conditional probability. Next, we analyzed some probability distributions, and how such distributions can be generated in Python.

In the next chapter, we will learn the basic concepts of Monte Carlo simulation and explore some of its applications. Then, we will discover how to generate a sequence of numbers that have been randomly distributed according to a Gaussian. Finally, we will take a look at the practical application of the Monte Carlo method in order to calculate a definite integral.

Section 2: Simulation Modeling Algorithms and Techniques

In this section, we will analyze some of the most used algorithms in numerical simulation. We will see the basics of how these techniques work and how to apply them to solve real problems.

This section contains the following chapters:

4 Exploring Monte Carlo Simulations

Monte Carlo simulation is used to reproduce and numerically solve a problem in which random variables are also involved, and whose solution by analytical methods is too complex or impossible. In addition, the use of simulation allows you to test the effects of changes in the input variables or in the output function more easily and with a high degree of detail. Starting from modeling the processes and generating random variables, simulations composed of multiple runs capable of obtaining an approximation of the probability of certain results are performed.

This method has assumed great importance in many scientific and engineering areas, above all for its ability to deal with complex problems that previously could only be solved through deterministic simplifications. It is mainly used in three distinct classes of problems: optimization, numerical integration, and the generation of probability functions. In this chapter, we will explore various techniques based on Monte Carlo methods for process simulation. We will first learn the basic concepts and then we will learn how to apply them to practical cases.

In this chapter, we're going to cover the following main topics:

- Introducing Monte Carlo simulation
- Understanding the central limit theorem
- Applying Monte Carlo simulation
- Performing numerical integration using Monte Carlo

Technical requirements

In this chapter, we will provide an introduction to Monte Carlo simulation. In order to deal with the topics in this chapter, it is necessary to have a basic knowledge of algebra and mathematical modeling.

To work with the Python code in this chapter, you'll need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `SimulatingPi.py`
- `CentralLimitTheorem.py`

Introducing Monte Carlo simulation

In simulation procedures, the evolution of a process is followed, but at the same time, forecasts of possible future scenarios are made. A simulation process consists of building a model that closely imitates a system. From the model, numerous samples of possible cases are generated and subsequently studied over time. After this, the results are analyzed over time, all while highlighting the alternative decisions that can be made.

The term Monte Carlo simulation was born at the beginning of the Second World War by J. von Neumann and S. Ulam as part of the Manhattan project at the Los Alamos nuclear research center. They replaced the parameters of the equations that describe the dynamics of nuclear explosions with a set of random numbers. The choice of the name Monte Carlo was due to the uncertainty of the winnings that characterize the famous casino of the Principality of Monaco.

Monte Carlo components

To obtain a simulation with satisfactory results, applications that use the Monte Carlo method are based on the following components:

- **Probability Density Functions (PDFs)** of the physical system
- Methods for estimating and reducing statistical error
- A uniform random number generator, which allows us to obtain a uniform function distributed in the range between 0-1
- An inversion function, which allows one random variable uniform to be passed to a population variable
- Sampling rules, which allow us to divide the space into specific volumes of interest
- Parallelization and optimization algorithms for efficient implementation with respect to the available computing architecture

The Monte Carlo simulation calculates a series of possible realizations of the phenomenon in question, along with the weight of the probability of a specific occurrence, while trying to explore the whole space of the parameters of the phenomenon.

Once this random sample has been calculated, the simulation gathers measurements of the quantities of interest on this sample. It is well executed if the average value of these measurements on the system realizations converge to the true value.

Important Note

The functionality of the Monte Carlo simulation can be summarized as follows: a phenomenon is observed *n* times, and the methods adopted in each event are recorded, with the aim of identifying the statistical distribution of the character.

First Monte Carlo application

The primary objective of the Monte Carlo method is to estimate a parameter representative of a population. To do this, the calculator generates a series of *n* random numbers that make up the sample of the population in question.

For example, suppose we want to evaluate a parameter, *A*, currently unknown, which can be interpreted as the average value of a random variable. The Monte Carlo method consists of, in this case, estimating this parameter by calculating the average of a sample consisting of *N* values of *X*. This is obtained using a procedure that involves the use of random numbers, as shown in the following diagram:

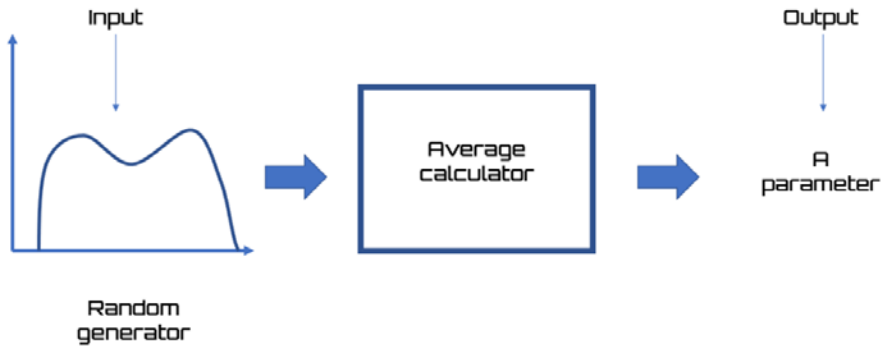


Figure 4.1 – Process of a random generator

In the Monte Carlo simulation, a series of possible realizations of a phenomenon are calculated in order to explore all the available parameters.

Important Note

In this calculation, the weight of the probability of each event assumes importance. When the representative sample is calculated, the simulation measures the quantities of interest on this sample.

Monte Carlo simulation works if the average value of these measurements on the system results converges to the real value.

Monte Carlo applications

The Monte Carlo simulation proves to be a valid tool for addressing the following problems:

- Intrinsically probabilistic problems involving phenomena related to the stochastic fluctuation of random variables
- Problems of an essentially deterministic nature, completely devoid of random components, but whose solution strategy can be treated as an expectation value of a function of stochastic variables

The necessary conditions for the application of the method are the independence and analogy of the experiments. For independence, it is understood that the results of each repetition of the experiment must not be able to influence each other. By analogy, however, reference is made to the fact that, for the observation of the character, the same experiment is repeated *n* times.

Applying the Monte Carlo method for Pi estimation

The Monte Carlo method is a problem-solving strategy that uses statistics. If we indicate with *P* the probability of a certain event, then we can randomly simulate this event and obtain *P* by finding the ratio between the number of times our event occurred and the number of total simulations, as follows:

$$P = \frac{\text{number of event occurrences}}{\text{number of total simulations}}$$

We can apply this strategy to get an approximation of Pi. Pi (π) is a mathematical constant indicating the relationship between the length of a circumference and its diameter. In fact, if we denote by *C* the length of a circumference and by *d* its diameter, we know that $C = d * \pi$. The length of a circumference with a diameter equal to 1 is worth π.

Important Note

Usually, we approximate the value of Pi with 3.14 to simplify the accounts. However, π is an irrational number; that is, it has an infinite number of digits after the decimal point that never repeat on a regular basis.

Given a circle of radius 1, it can be inscribed in a square of length 2. For convenience, we will only consider a fraction of the circle, as shown in the following figure:

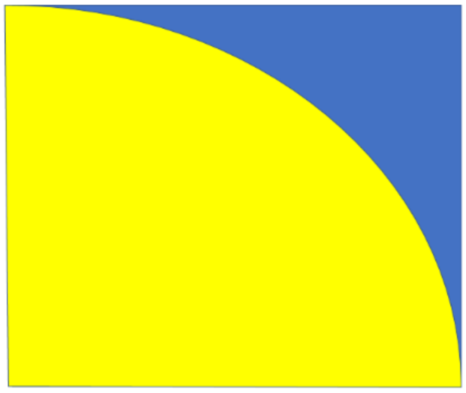


Figure 4.2 – A fraction of the circle

By analyzing the previous figure, we can see that the area of the square in blue is 1 and that the area of the circular sector in yellow (1/4 of the circle) is instead pi / 4. We randomly place a very large number of points inside the square. Thanks to the very large number and random distribution, we can approximate the size of the areas with the number of points contained in them.

If we generate *N* random numbers inside the square, the number of points that fall in the circular sector, which we will denote by *M*, divided by the total number of generated numbers, *N*, we will have to approximate the area of the circular sector and therefore it will be equal to *Pi/4*. From this, we can derive the following equation:

$$\pi = \frac{4 * M}{N}$$

The greater the number of points generated, the more precise the approximation of Pi will be.

Now, let's analyze the code line by line to understand how we have implemented the simulation procedure for estimating Pi:

1. To start, we import the necessary libraries:

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

   The `math` library provides access to the mathematical functions defined by the C standard library. The `random` library implements pseudo-random number generators for various distributions. The `CIT` module is based on the Mersenne Twister algorithm. The `numpy` library offers additional scientific functions of the Python language, designed to perform operations on vectors and dimensional matrices. Finally, the `matplotlib` library is a Python library for printing high-quality graphics.

2. Let's move on and initialize the parameters:

```
N = 10000
M = 0
```

As we mentioned previously, `N` represents the number of points that we generate, that is, those that we are going to position. Instead, `M` will be the points that fall within the circular sector. To start, these points will be zero and as we generate them, we will try to perform a check. In a positive scenario, we will gradually increase this number.

3. Let's proceed and initialize the vectors that will contain the coordinates of the points that we will generate:

```
XCircle=[]
YCircle=[]
XSquare=[]
YSquare=[]
```

Here, we have defined two types of points: `Circle` and `Square`. `Circle` is a point that falls within the circular sector, while `Square` is a point that falls within the space of the square outside the circular sector. Now, we can generate the points:

```
for p in range(N):
    x=random.random()
    y=random.random()
```

Here, we used a `for` loop that iterates the process a number of times equal to the number `(N)` of samples we want to generate. We then used the `random()` function of the `CiT` library to generate the points. The `random()` function returns the next nearest floating-point value from the generated sequence. All the return values are enclosed between `0` and `1.0`.

4. Now, we can check where the point we just generated falls:

```
if(x**2+y**2 <= 1):
        M+=1
        XCircle.append(x)
        YCircle.append(y)
    else:
        XSquare.append(x)
        YSquare.append(y)
```

The `if` loop allows us to check the position of the points. Recall that the points of a circumference are defined by the following equation:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

If $x_0=y_0=0$ and $r=1$, the previous equation turns into the following:

$$x^2 + y^2 = 1$$

This makes us understand that the necessary condition for a point to fall within the circular sector is that the following equation is verified:

$$x^2 + y^2 \leq 1$$

If this condition is satisfied, the value of *M* is increased by 1 unit and the values of the *x* and *y* values that are generated are stored in the `Circle` point vector (`XCircle`, `YCircle`). Otherwise, the value of *M* is not updated, and the values of *x* and *y* that are generated are stored in the vector of the `Square` point vector (`XSquare`, `YSquare`).

5. Now that we've iterated this procedure for the 10,000 points that we have decided to generate, we can make the estimate of `Pi`:

```
Pi = 4*M/N
print('N=%d M=%d Pi=%.2f' %(N,M,Pi))
```

In this way, we can calculate `Pi` and print the results, as follows:

```
N=10000 M=7857 Pi=3.14
```

The estimate that we've obtained is acceptable. Usually, we stop at the second decimal place, so this is okay. Now, let's draw a graph, where we will draw the generated points. To start, we will generate the points of the circumference arc:

```
XLin=np.linspace(0,1)
YLin=[]
for x in XLin:
    YLin.append(math.sqrt(1-x**2))
```

The `linspace()` function of the `numpy` library allows us to define an array composed of a series of *N* numerical elements equally distributed between two extremes `(0,1)`. This will be the *x* of the arc of circumference (`XLin`). On the other hand, the *y* numerical elements (`YLin`) will be obtained from the equation of the circumference while solving them with respect to *y*, as follows:

$$y = \sqrt{1 - x^2}$$

To calculate the square root, we used the `math.sqrt()` function.

6. Now that we have all the points, we can draw the graph:

```
plt.axis    ('equal')
plt.grid    (which='major')
plt.plot    (XLin , YLin, color='red' , linewidth='4')
plt.scatter(XCircle, YCircle, color='yellow', marker
='.')
plt.scatter(XSquare, YSquare, color='blue'  , marker
='.')
plt.title   ('Monte Carlo method for Pi estimation')
plt.show()
```

The `scatter()` function allows us to represent a series of points not closely related to each other on two axes. The following diagram is printed:

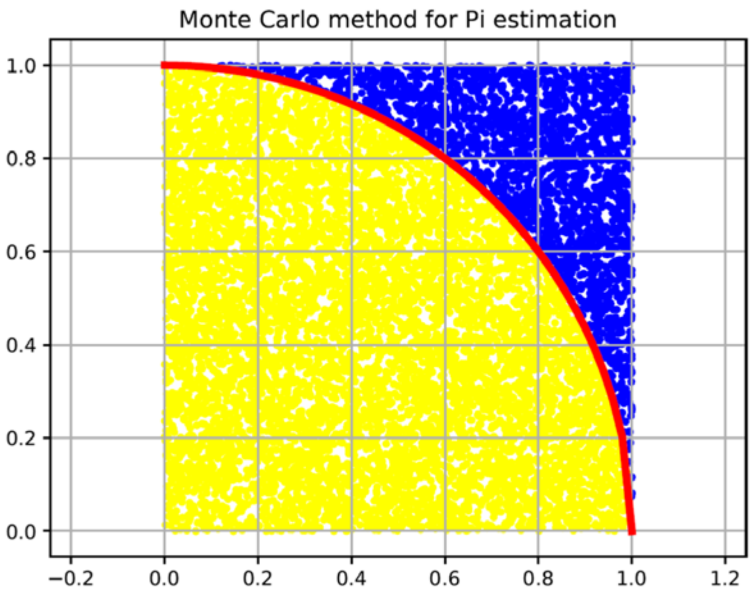


Figure 4.3 – Plot of the Pi estimation

Consistent with what we established at the beginning of this chapter, we plotted the points inside the circular sector in yellow, while those outside the circular sector are in blue. To highlight the separation line, we have drawn the circumference arc in red.

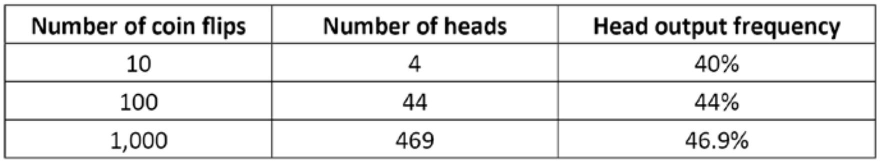
Now that we've applied the Monte Carlo method to estimate Pi, the time has come to deepen some fundamental concepts for simulation based on the generation of random numbers.

Understanding the central limit theorem

The Monte Carlo method is essentially a numerical method for calculating the expected value of random variables; that is, an expected value that cannot be easily obtained through direct calculation. To obtain this result, the Monte Carlo method is based on two fundamental theorems of statistics: the law of large numbers and the central limit theorem.

Law of large numbers

This theorem states the following: considering a very large number of variables, $x(N \rightarrow \infty)$, the integral that defines the average value is approximate to the estimate of the expected value. Let's try to give an example so that you understand this. We flip a coin 10 times, 100 times, and 1,000 times and check how many times we get heads. We can put the results we obtained into a table, as follows:



Number of coin flips	Number of heads	Head output frequency
10	4	40%
100	44	44%
1,000	469	46.9%

Figure 4.4 – Table showing the results for coin toss

Analyzing the last column of the previous table, we can see that the value of the frequency approaches the probability of 50%. We can therefore say that as the number of tests increases, the frequency value tends to the theoretical probability value. The latter value can be achieved in the hypothesis of a number of throws that tends to infinity.

Important Note

The use of the law of large numbers is different. Actually, the law of large numbers allowed us, in the Applying the *Monte Carlo method for Pi estimation* section, to equal the number of launches with the area of the circular sector. In this way, we were able to estimate the value of Pi simply by generating random numbers. Also, in this case, the greater the number of random variables generated, the closer the estimate of Pi is to the expected value.

The law of large numbers allows you to determine the centers and weights of a Monte Carlo analysis for the estimate of definite integrals but does not say how large the number *N* must be. You do not have an estimate to understand with what order of magnitude you can perform a simulation so that you can consider the numbers large enough. To answer this question, it is necessary to resort to the central limit theorem.

Central limit theorem

Monte Carlo not only allows us to obtain an estimate of the expected value, as established by the law of large numbers, but also allows us to estimate the uncertainty associated with it. This is possible thanks to the central limit theorem, which returns an estimate of the expected value and the reliability of that result.

Important Note

The central limit theorem can be summarized with the following definition: given a dataset with an unknown distribution, the sample's means will approximate the normal distribution.

If the law of large numbers tells us that the random variable allows us to evaluate the expected value, the central limit theorem provides information on its distribution.

The interesting feature of the central limit theorem is that there are no constraints on the distribution of the function used for the generation of the *N* samples, from which the random variable is formed. In fact, it is not important what the distribution associated with the random variable is, but when the average is characterized by a finite variance and is obtained for a very large number of samples, it can be described through a Gaussian distribution.

Let's take a look at a practical example. We generate 10,000 random numbers with a uniform distribution. We then extract 100 samples from this population, also taken randomly. We repeat this operation for a consistent number of times and for each time, we evaluate its average and store this value in a vector. In the end, we draw a histogram of the distribution that we have obtained. Here is the Python code:

```
import random
import numpy as np
import matplotlib.pyplot as plt
a=1
b=100
N=10000
DataPop=list(np.random.uniform(a,b,N))
```

```
plt.hist(DataPop, density=True, histtype='stepfilled',
alpha=0.2)
plt.show()

SamplesMeans = []
for i in range(0,1000):
    DataExtracted = random.sample(DataPop,k=100)
    DataExtractedMean = np.mean(DataExtracted)
    SamplesMeans.append(DataExtractedMean)
plt.figure()
plt.hist(SamplesMeans, density=True, histtype='stepfilled',
alpha=0.2)
plt.show()
```

Now, let's analyze the code line by line to understand how we have implemented the simulation procedure to understand the central limit theorem:

1. To start, we import the necessary libraries:

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

   The `random` library implements pseudo-random number generators for various distributions. The `numpy` library offers additional scientific functions of the Python language and is designed to perform operations on vectors and dimensional matrices.

   Finally, the `matplotlib` library is a Python library for printing high-quality graphics.

2. Let's move on and initialize the parameters:

   ```
   a=1
   b=100
   N=10000
   ```

   The `a` and `b` parameters are the extremes of the range and `N` is the number of values we want to generate.

Now, we can generate the uniform distribution using the `numpy random.uniform()` function, as follows:

```
DataPop=list(np.random.uniform(a,b,N))
```

3. At this point, we draw a histogram of the data in order to verify that it is a uniform distribution:

```
plt.hist(DataPop, density=True, histtype='stepfilled', 
alpha=0.2)
plt.show()
```

The `matplotlib.hist()` function draws a histogram; that is, a diagram in classes of a continuous character.

This is used in many contexts, usually to show statistical data when there is an interval of definition of the independent variable divided into subintervals.

The following diagram is printed:

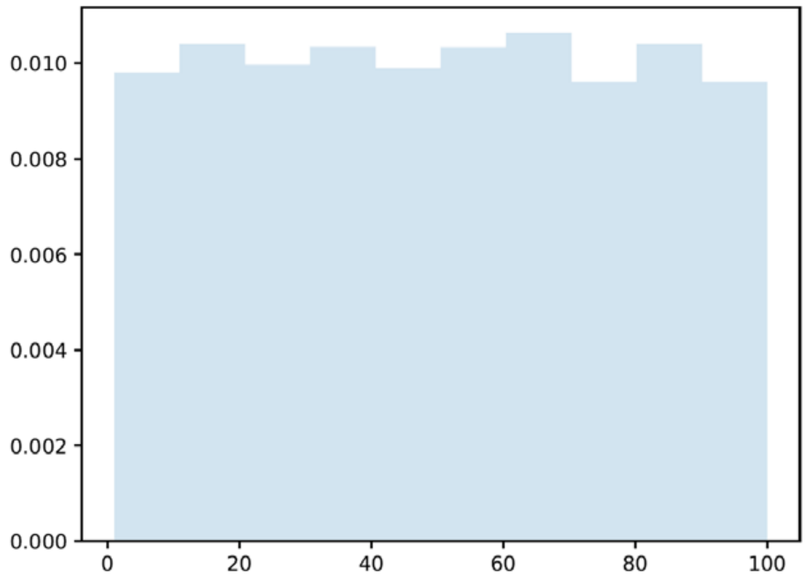


Figure 4.5 – Plot of the data distribution

The distribution appears evidently uniform—in fact, we can see that each bin is populated with an almost constant frequency.

4. Let's now pass the values to the extraction of the samples from the generated population:

```
SamplesMeans = []
for i in range(0,1000):
    DataExtracted = random.sample(DataPop,k=100)
    DataExtractedMean = np.mean(DataExtracted)
    SamplesMeans.append(DataExtractedMean)
```

   First, we initialized the vector that will contain the samples. To do this, we used a `for` loop to repeat the operations 1,000 times. At each step, we first extracted 100 samples from the population generated using the `random.sample()` function. The `random.sample()` function extracts samples without repeating the values and without changing the input sequence.

5. Next, we calculated the average of the extracted samples and added the result at the end of the vector containing the samples. Now, all we need to do is view the results:

```
plt.figure()
plt.hist(SamplesMeans, density=True,
histtype='stepfilled', alpha=0.2)
plt.show()
```

   The following histogram is printed:

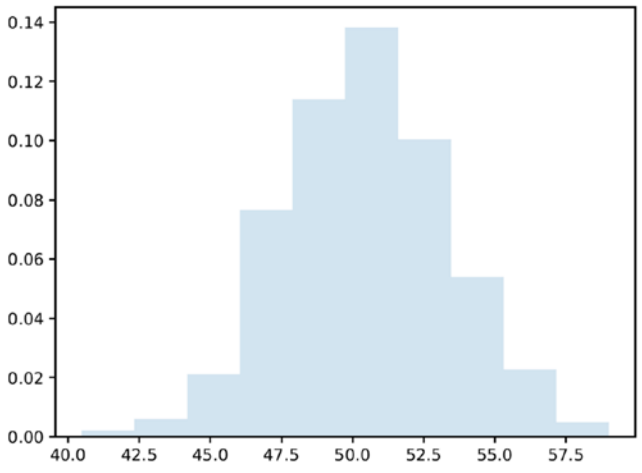


Figure 4.6 – Plot of the extracted samples

The distribution has now taken on the typical bell-shaped curve characteristic of the Gaussian distribution. This means that we have proved the central limit theorem.

Applying Monte Carlo simulation

Monte Carlo simulation used to study the response of a model to randomly generated inputs. The simulation process takes place in the following three phases:

1. *N* inputs are generated randomly.
2. A simulation is performed for each of the *N* inputs.
3. The outputs of the simulations are aggregated and examined. The most common measures include estimating the average value of an output and distributing the output values, as well as the minimum or maximum output value.

Monte Carlo simulation is widely used for the analysis of financial, physical, and mathematical models.

Generating probability distributions

The generation of probability distributions that cannot be found with analytical methods can easily be addressed with Monte Carlo methods. For example, let's say we want to estimate the probability distribution of the damage caused by earthquakes in a year in Japan.

Important Note

In this type of analysis, there are two sources of uncertainty: how many earthquakes there will be in a year and how much damage each earthquake will do. Even if it is possible to assign a probability distribution to these two logical levels, it is not always possible to put this information together with analytical methods to derive the distribution of the annual losses.

It is easier to do a Monte Carlo simulation of this type, as follows:

1. A random number is extracted from the distribution of the number of annual events.
2. If events occur from the previous point, extractions are made from the distribution of losses.
3. Finally, we add the values of the extractions we performed to obtain a value that represents an annual loss caused by events.

By cyclically repeating these three points, a sample of annual losses is generated, from which it is possible to estimate the probability distribution, which could not be obtained analytically.

Numerical optimization

There are various algorithms that can be used to find the local minima of a function. Typically, these algorithms proceed according to the following steps:

1. They start from an assigned point.
2. They control in which direction the function tends to have values smaller than the current one.
3. Moving in this direction, they find a new point where the function has a lower value than the previous one.

They keep repeating these steps until they reach a minimum. In the case of a function with only one minimum, this method allows us to achieve a result. But what if we have a function with many local minima and we want to find the point that minimizes the function globally? The following diagram shows the two cases just mentioned; that is, a distribution with only one minimum (left) and a distribution with several minimums (right):

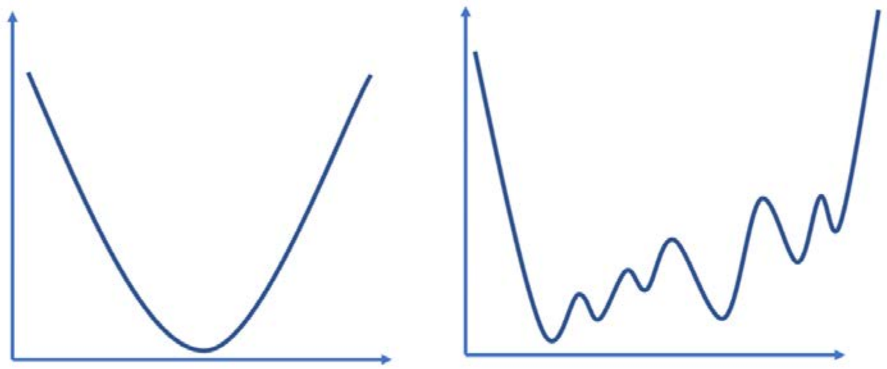


Figure 4.7 – Graphs of the two distributions

A local search algorithm could stop at any of the many local minima of the function. How would you know if you found one of the many local minimums or the global minimum? There is no way to strictly establish this. The only practical possibility is to explore different areas of the search domain to increase the probability of finding, among the various local minima, the global one.

Important Note

Different methods have been developed to explore domains, which can be very complicated, with many dimensions and with constraints to be respected.

Monte Carlo methods provide a solution to this problem; that is, an initial population of points belonging to the domain is created, which is then evolved by defining coupling algorithms between the points in which random genetic mutations also occur. When simulating different generations of points, a selection process intervenes that maintains only the best points, that is, those that give lower values of the function to be minimized.

Each generation keeps track of which point represents the best specimen ever. Continuing with this process, the points tend to move to local lows, but at the same time, they explore many areas of the optimization domain. This process can continue indefinitely, though at some point it is stopped, and the best specimen is taken as an estimate of the global minimum.

Project management

Monte Carlo methods allow you to simulate the behavior of an event of interest and, in general, return as a result a random variable whose properties, such as mean, variance, probability density function, and so on, provide us with important information on the quality of the simulation.

This is a statistical analysis technique that can be applied in all those situations in which we are faced with very uncertain project estimates, with the aim of reducing the level of uncertainty through a series of simulations. In this sense, it can be applied to the analysis of the times, costs, and risks associated with a project and, therefore, to the evaluation of the impact that this project may have on the community.

Important Note

For each of these variables, the simulations do not provide a single estimate but a range of possible estimates, along with, associated with each estimate, the level of probability that that estimate is accurate.

For example, this technique can be used to determine the overall cost of a project through a discrete series of simulation cycles. In the planning phase of a project, the activities that make up the project are identified, and the cost associated with each activity is estimated. In this way, the total cost of the project can be determined. Since, however, we rely on cost estimates, we cannot be sure that this overall cost, and therefore also the completion costs, are certain. A Monte Carlo simulation can therefore be carried out.

Performing numerical integration using Monte Carlo

Monte Carlo simulations represent a numerical solution for calculating integrals. In fact, with the use of the Monte Carlo algorithm, it is possible to adopt a numerical procedure for the solution of mathematical problems, with many variables that do not present an analytical solution. The efficiency of the numerical solution increases compared to other methods when the size of the problem increases.

Important Note

Let's analyze the problem of a definite integral. In the simplest cases, there are methods for integration that foresee the use of techniques such as integration by parts, integration by replacement, and so on. In more complex situations, however, it is necessary to adopt numerical procedures that involve the use of a computer. In these cases, the Monte Carlo simulation provides a simple solution that's particularly useful in cases of multidimensional integrals.

However, it is important to highlight that the result that's returned by this simulation approximates the integral and not its precise value.

Defining the problem

In the following equation, we denote with *I* the definite integral of the function *f* in the limited interval [a, b]:

$$I = \int_a^b F(x)dx$$

In the interval [a, b], we identify the maximum of the function *f* and indicate it with *U*. To evaluate the approximation that we are introducing, we draw a base rectangle, [a, b], and the height, *U*. The area under the function *f(x)*, which represents the integral of *f(x)*, will surely be smaller than the area of the base rectangle, [a, b], and the height, *U*. The following diagram shows the area subtended by the function *f* — which represents the integral of *f(x)* — and the area *A* of the rectangle with base [a, b] and height *U*, which represents our approximation:

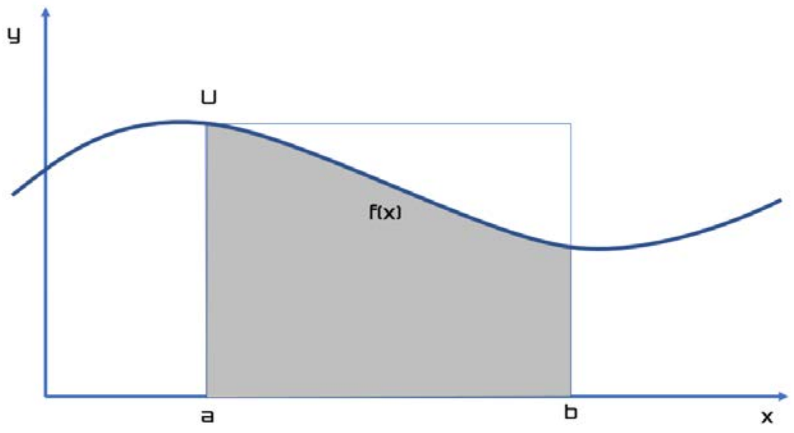


Figure 4.8 – Plot of the function

By analyzing the previous diagram, we can identify the following intervals:

- $x \in [a, b]$
- $y \in [0, U]$

In the Monte Carlo simulation, *x* and *y* both represent random numbers. At this point, we can consider a point in the plane of the Cartesian coordinates, (*x*, *y*). Our goal is to determine the probability that this point is within the area highlighted in the previous diagram; that is, that it is $y \leq f(x)$. We can identify two areas:

- The area subtended by the function *f*, which coincides with the definite integral *I*
- The area *A* of the rectangle with base [a, b] and height *U*

Let's try to write a relationship between the probability and these two areas:

$$P(y \leq f(x)) = \frac{I}{A} = \frac{I}{(b-a) * U}$$

It is possible to estimate the probability, *P (y <= f (x))*, through Monte Carlo simulation. In fact, in *Applying the Monte Carlo method for Pi estimation* section, we faced a similar case. To do this, *N* pairs of random numbers (x_i, y_i) are generated, as follows:

$$x_i \in [a, b]$$

$$y_i \in [0, U]$$

Generating random numbers in the intervals considered will certainly determine conditions in which $y_i \leq f(x_i)$ will result. If we number this quantity and denote it with the symbol M, we can analyze its variation. This is an approximation whose accuracy increases as the number of random number pairs (x_i, y_i) generated increases. The approximation of the calculation of the probability $P(y < f(x))$ will therefore be equal to the following value:

$$\mu = \frac{M}{N}$$

After calculating this probability, it will be possible to trace the value of the integral using the previous equation, as follows:

$$I \cong \mu * (b-a) * U = \frac{M}{N} * (b-a) * U = \frac{M}{N} * A$$

This is the mathematical representation of the problem. Now, let's see the numerical solution.

Numerical solution

We will begin by setting up the components that we will need for the simulation, starting from the libraries that we will use to defining the function and its domain of existence. The Python code for numerical integration through the Monte Carlo method is shown here:

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

random.seed(2)
f = lambda x: x**2
a = 0.0
b = 3.0
NumSteps = 1000000
XIntegral=[]
YIntegral=[]
XRectangle=[]
YRectangle=[]
```

Now, let's analyze the code line by line to understand how we have implemented the simulation procedure to understand the central limit theorem:

1. To start, we import the necessary libraries:

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

   The `random` library implements pseudo-random number generators for various distributions. The `numpy` library offers additional scientific functions of the Python language, designed to perform operations on vectors and dimensional matrices. Finally, the `matplotlib` library is a Python library for printing high-quality graphics. Let's set the seed:

   ```
   random.seed(2)
   ```

   The `random.seed()` function is useful if we wish to have the same set of data available to be processed in different ways as it makes the simulation reproducible. This function initializes the basic random number generator. If you use the same seed in two successive simulations, you always get the same sequence of pairs of numbers.

2. Now, we will define the function that we want to integrate:

   ```
   f = lambda x: x**2
   ```

   We know that in order to define a function in Python, we use the `def` clause, which automatically assigns a variable to it. Actually, functions can be treated like other Python objects, such as strings and numbers. These objects can be created and used at the same time (on the fly) without resorting to the creation and definition of variables that contain them.

   In Python, functions can also be used in this way, using a syntax called **lambda**. The functions that are created in this way are anonymous. This approach is often used when you want to pass a function as an argument for another function. The lambda syntax requires the `lambda` clause, followed by a list of arguments, a colon character, the expression to evaluate the arguments, and finally the input value.

3. Let's move on and initialize the parameters:

   ```
   a = 0.0
   b = 3.0
   NumSteps = 1000000
   ```

As we mentioned in the *Defining the problem* section, `a` and `b` represent the ends of the range in which we want to calculate the integral. `NumSteps` represents the number of steps in which we want to divide the integration interval. The greater the number of the steps, the better the simulation will be, even if the algorithm becomes slower.

4. Now, we will define four vectors so that we can store the pairs of generated numbers:

```
XIntegral=[]
YIntegral=[]
XRectangle=[]
YRectangle=[]
```

Whenever the generated `y` value is less than or equal to *f(x)*, this value and the relative `x` value will be added at the end of the `XIntegral`, and `YIntegral` vectors. Otherwise, they will be added at the end of the `XRectangle`, and `YRectangle` vectors.

Min-max detection

Before using the method, it is necessary to evaluate the minimum and maximum of the function:

Important Note

Recall that if the function has only one minimum/maximum, the work is simple. If there are repeated minimums/maximums, then the procedure becomes more complex.

1. In the following Python code, we are extracting the min/max of the distribution:

```
ymin = f(a)
ymax = ymin
for i in range(NumSteps):
    x = a + (b - a) * float(i) / NumSteps
    y = f(x)
    if y < ymin: ymin = y
    if y > ymax: ymax = y

```

2. To understand all these cases, even complex ones, we will look for the minimum/maximum for each step in which we have divided the interval *[a, b]*. We first initialize the minimum and maximum with the value of the function in the far left of the range `(a)`:

```
ymin = f(a)
ymax = ymin
```

3. Then, we use a `for` loop to check the value at each step:

```
for i in range(NumSteps):
    x = a + (b - a) * float(i) / NumSteps
    y = f(x)
```

4. For each step, the `x` value is obtained by increasing the left end of the interval (`a`) by a fraction of the total number of steps provided by the current value of `i`. Once this is done, the function at that point is evaluated. Now, you can check this, as follows:

```
    if y < ymin: ymin = y
    if y > ymax: ymax = y
```

5. The two `if` statements allow us to verify whether the current value of *f* is less than/greater than the value chosen so far as the minimum/maximum and, if so, to update these values. Now, we can apply the Monte Carlo method.

Monte Carlo method

Now, we will apply the Monte Carlo method, as follows:

1. Now that we've set and calculated the necessary parameters, it is time to proceed with the simulation:

```
A = (b - a) * (ymax - ymin)
N = 1000000
M = 0
for k in range(N):
    x = a + (b - a) * random.random()
    y = ymin + (ymax - ymin) * random.random()
    if y <= f(x):
        M += 1
        XIntegral.append(x)
```

```
            YIntegral.append(y)
    else:
            XRectangle.append(x)
            YRectangle.append(y)
NumericalIntegral = M / N * A
print ('Numerical integration = ' +
str(NumericalIntegral))

```

2. To start, we will calculate the area of the rectangle, as follows:

```
A = (b - a) * (ymax - ymin)
```

3. Then, we will set the numbers of random pairs we want to generate:

```
N = 1000000
```

4. Here, we initialize the `M` parameter, which represents the number of points that fall under the curve that represents *f(x)*:

```
M = 0
```

5. Now, we can calculate this value. To do this, we will use a `for` loop that iterates the process `N` times. First, we generate the two random numbers, as follows:

```
for k in range(N):
    x = a + (b - a) * random.random()
    y = ymin + (ymax - ymin) * random.random()
```

6. Both `x` and `y` fall within the rectangle of area *A*; that is, x ∊ [*a, b*] and y ∊ [*0, maxy*].

   Now, we need to determine whether the following is true:

$$y \leq f(x)$$

   We can do this with an `if` statement, as follows:

```
if y <= f(x):
            M += 1
            XIntegral.append(x)
            YIntegral.append(y)
```

7. If the condition is true, then the value of `M` is incremented by one unit and the current values of `x` and `y` are added to the `XIntegral` and `YIntegral` vectors. Otherwise, the points will be stored in the `XRectangle`, and `YRectangle` vectors:

```
    else:
            XRectangle.append(x)
            YRectangle.append(y)
```

8. After iterating for `N` times, we can estimate the integral:

```
NumericalIntegral = M / N * A
print ('Numerical integration = ' +
str(NumericalIntegral))
```

The following result is printed:

```
Numerical integration = 8.996787006398996
```

The analytical solution for this simple integral is as follows:

$$I = \int_0^3 x^2 dx = \left[\frac{x^3}{3} + c\right]_0^3 = 9$$

The percentual error we made is equal to the following:

$$\frac{9 - 8.996787006398996}{9} * 100 = 0.03\ \%$$

This is a negligible error that defines our reliable estimate.

Visual representation

We will now plot the results using the following steps:

1. Finally, we can visualize what we have achieved in the numerical integration by plotting scatter plots of the generated points. For this reason, we memorized the pairs of points that were generated in the four vectors:

```
XLin=np.linspace(a,b)
YLin=[]
for x in XLin:
    YLin.append(f(x))
```

```
plt.axis    ([0, b, 0, f(b)])
plt.plot    (XLin,YLin, color='red' , linewidth='4')
plt.scatter(XIntegral, YIntegral, color='blue', marker
='.')
plt.scatter(XRectangle, YRectangle, color='yellow',
marker   ='.')
plt.title  ('Numerical Integration using Monte Carlo
method')
plt.show()
```

2. To start, we generate the points we need in order to draw the representative curve of the function:

```
XLin=np.linspace(a,b)
YLin=[]
for x in XLin:
    YLin.append(f(x))
```

The `linspace()` function of the `numpy` library allows us to define an array composed of a series of *N* numerical elements equally distributed between two extremes (0 , 1). This will be the *x* of the function, while the *y* (`YLin`) will be obtained from the equation of the function solving them with respect to *y*, as follows:

$$y = x^2$$

3. Now that we have all the points, we can draw the graph:

```
plt.axis    ([0, b, 0, f(b)])
plt.plot    (XLin,YLin, color='red' , linewidth='4')
plt.scatter(XIntegral, YIntegral, color='blue', marker
='.')
plt.scatter(XRectangle, YRectangle, color='yellow',
marker   ='.')
plt.title  ('Numerical Integration using Monte Carlo
method')
plt.show()
```

We first set the length of the axes using the `plt.axis()` function. So, we plotted the curve of the `x2` function, which, as we know, is a convex increasing the monotone function in the range of values considered [0 , 3].

We then plotted two scatter plots:

- One for the points that are under the curve (points in blue)
- One for the points that are above the function (points in yellow)

The `scatter()` function allows us to represent a series of points not closely related to each other on two axes.

The following diagram is Numerical integration using the Monte Carlo method:

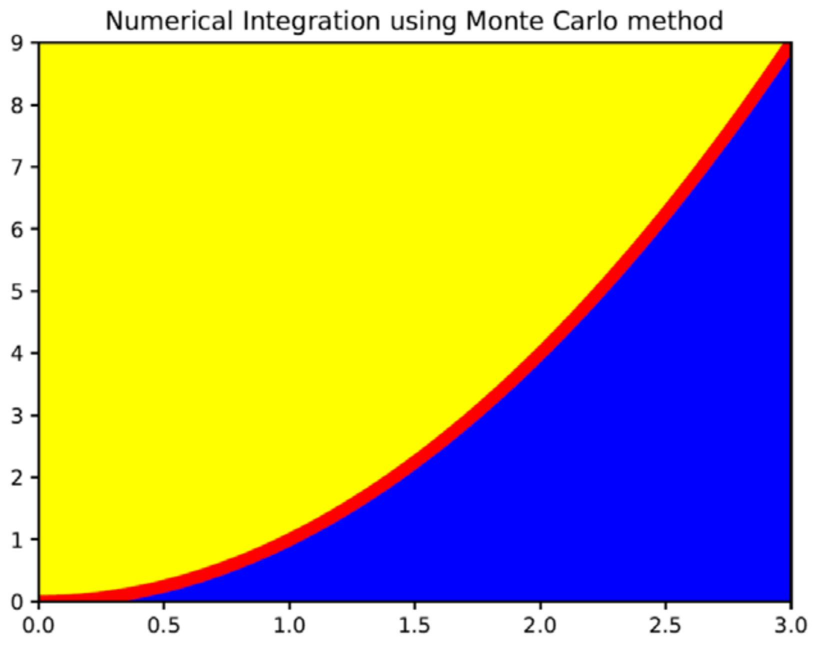


Figure 4.9 – Plot of numerical integration results

As we can see, all the points in blue are positioned below the curve of the function (curve in red), while all the points in yellow are positioned above the curve of the function.

Summary

In this chapter, we addressed the basic concepts of Monte Carlo simulation. We explored the Monte Carlo components used to obtain a simulation with satisfactory results. Hence, we used Monte Carlo methods to estimate the value of pi.

We then tackled two fundamental concepts of Monte Carlo simulation: the law of large numbers and the central limit theorem. For example, the law of large numbers allows us to determine the centers and weights of a Monte Carlo analysis for the estimate of definite integrals. The central limit theorem is of great importance and it is thanks to this that many statistical procedures work.

Next, we analyzed practical applications of using Monte Carlo methods in real life: numerical optimization and project management. Finally, we learned how to perform numerical integration using Monte Carlo techniques.

In the next chapter, we will learn the basic concepts of the Markov process. We will understand the agent-environment interaction process and how to use Bellman equations as consistency conditions for the optimal value functions to determine the optimal policy. Finally, we will learn how to implement Markov chains to simulate random walks.

5
Simulation-Based Markov Decision Processes

Markov Decision Processes (**MDPs**) model decision-making in situations where outcomes are partly random and partly under the control of a decision maker. An MDP is a stochastic process characterized by five elements: decision epochs, states, actions, transition probability, and reward. The characteristic elements of a Markovian process are the states in which the system finds itself and the available actions that the decision maker can carry out on those states. These elements identify two sets: the set of states in which the system can be found and the set of actions available for each specific state. The action chosen by the decision maker determines a random response from the system, which brings it into a new state. This transition returns a reward that the decision maker can use to evaluate the goodness of their choice. In this chapter, we will learn how to deal with decision-making processes with Markov chains. We will analyze the concepts underlying Markovian processes and then analyze some practical applications to learn how to choose the right actions for the transition between different states of the system.

In this chapter, we're going to cover the following main topics:

- Overview of Markov processes
- Introducing Markov chains
- Markov chain applications
- The Bellman equation explained
- Multi-agent simulation

Technical requirements

In this chapter, MDPs will be introduced. In order to deal with the topics in this chapter, it is necessary that you have a basic knowledge of algebra and mathematical modeling.

To work with the Python code in this chapter, you'll need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `SimulatingRandomWalk.py`
- `WeatherForecasting.py`

Overview of Markov processes

Markov's decision-making process is defined as a discrete-time stochastic control process. In *Chapter 2, Understanding Randomness and Random Numbers*, we said that stochastic processes are numerical models used to simulate the evolution of a system according to random laws. Natural phenomena, both by their very nature and by observation errors, are characterized by random factors. These factors introduce a random number into the observation of the system. This random factor determines an uncertainty in the observation since it is not possible to predict with certainty what the result will be. In this case, we can only say that it will assume one of the many possible values with a certain probability.

If starting from an instant *t* in which an observation of the system is made, the evolution of the process will depend only on *t*, while it will not be influenced by the previous instants. Here, we can say that the stochastic process is Markovian.

Important note

A process is called **Markovian** when the future evolution of the process depends only on the instant of observation of the system and does not depend in any way on the past.

Characteristic elements of a Markovian process include the states in which the system finds itself and the available actions that the decision maker can carry out on those states. These elements identify two sets: the set of states in which the system can be found and the set of actions available for each specific state. The action chosen by the decision maker determines a random response from the system, which brings it into a new state. This transition returns a reward that the decision maker can use to evaluate the goodness of their choice.

The agent-environment interface

A Markovian process takes on the characteristics of an interaction problem between two elements in order to achieve a goal. The two characteristic elements of this interaction are the **agent** and the **environment**. The agent is the element that must reach the goal, while the environment is the element that the agent must interact with. The environment corresponds to everything that is external to the agent.

The agent is a piece of software that performs the services necessary for another piece of software in a completely automatic and intelligent way. They are known as intelligent agents.

The essential characteristics of an agent are listed here:

- The agent continuously monitors the environment, and this action causes a change in the state of the environment.
- The available actions belong to a continuous or discrete set.
- The agent's choice of action depends on the state of the environment and this choice requires a certain degree of intelligence as it is not trivial.
- The agent has a memory of the choices made – intelligent memory.

The agent's behavior is characterized by attempting to achieve a specific goal. To do this, it performs actions on an environment it does not know a priori, or at least not completely. This uncertainty is filled through the interaction between the agent and the environment. In this phase, the agent learns to know the states of the environment by measuring it, in this way planning its future actions.

The strategy adopted by the agent is based on the principles of error theory: proof of the actions and memory of the possible mistakes made in order to make repeated attempts until the goal is achieved. These actions by the agent are repeated continuously, causing changes in the environment that change their state.

Important Note

Crucial to the agent's future choices is the concept of reward, which represents the environment's response to the action taken. This response is proportional to the weight that the action determines in achieving the objective: it will be positive if it leads to correct behavior, while it will be negative in the case of an incorrect action.

The decision-making process that leads the agent to achieving their objective can be summarized in three essential points:

- Objective of the agent
- Interaction with the environment
- Total or partial uncertainty of the environment

In this process, the agent receives stimuli from the environment through the measurements made by sensors. The agent decides what actions to take based on the stimuli received from the environment. As a result of the agent's actions, determining a change in the state of the environment will receive a reward.

The crucial elements in the decision-making process are shown in the following diagram:

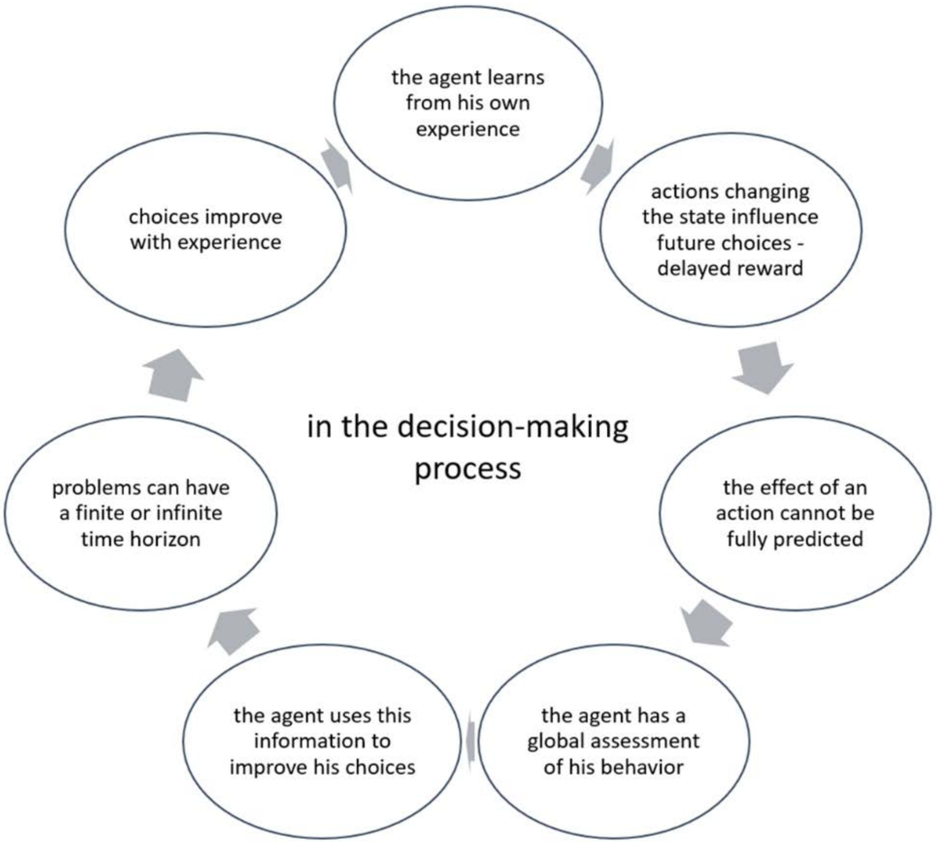

the agent learns from his own experience
actions changing the state influence future choices - delayed reward
the effect of an action cannot be fully predicted
the agent has a global assessment of his behavior
the agent uses this information to improve his choices
problems can have a finite or infinite time horizon
choices improve with experience
in the decision-making process

Figure 5.1 – The agent's decision-making process

While choosing an action, it becomes crucial to have a formal description of the environment. This description must return essential information regarding the properties of the environment, and not a precise representation of the environment.

Exploring MDPs

The agent-environment interaction, which we discussed in the previous section, is approached as a Markov decision-making process. This choice is dictated by loading problems and computational difficulties. As anticipated in the *Overview of Markov processes* section, a Markov decision-making process is defined as a discrete-time stochastic control process.

Here, we need to perform a sequence of actions, with each action leading to a non-deterministic change regarding the state of the environment. By observing the environment, we know its state after performing an action. On the other hand, if the observation of the environment is not available, we do not know the state, even after performing the action. In this case, the state is a probability distribution of all the possible states of the environment. In such cases, the change process can be viewed as a snapshot sequence.

The state at time *t* is represented by a random variable s_t. Decision-making is interpreted as a discrete-time stochastic process. A discrete-time stochastic process is a sequence of random variables x_t, with t ∈ N. We can define some elements as follows:

- **State space**: Set of values that random variables can assume
- **History of a stochastic process (path)**: Realization of the sequence of random variables

The response of the environment to a certain action is represented by the reward. The agent-environment interaction in a Markov decision process can be summarized by the following diagram:

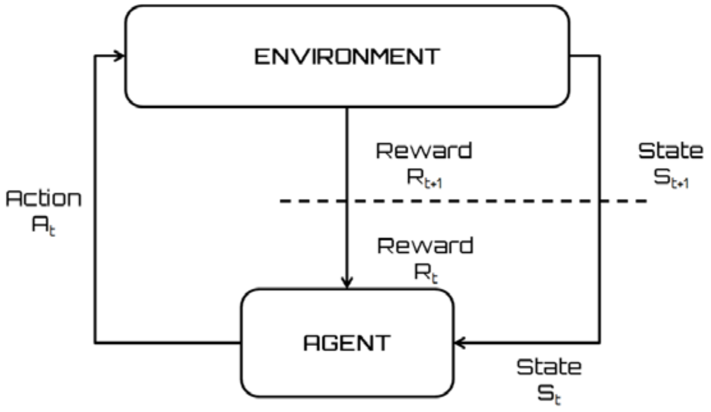


Figure 5.2 – The agent-environment interaction in MDP

The essential steps of the agent-environment interaction, schematically represented in the previous diagram, are listed here:

1. The interaction between the agent and the environment occurs at discrete instants over time.
2. In every instant, the agent monitors the environment by obtaining its state *st* ∈ *S*, where *S* is the set of possible states.

3. The agent performs an action $a \in A(s_t)$, where $A(s_t)$ is the set of possible actions available for the state *st*.
4. The agent chooses one of the possible actions according to the objective to be achieved.
5. This choice is dictated by the policy π *(s, a)*, which represents the probability that the action *a* is performed in the state *s*.
6. At time $t + 1$, the agent receives a numerical reward $r_{t+1} \in R$ corresponding to the action previously chosen.
7. Because of the choice, the environment passes into the new state.
8. The agent must monitor the state of the environment and perform a new action.
9. This iteration repeats until the goal is achieved.

In the iterative procedure we have described, the state s_{t+1} depends on the previous state and the action taken. This feature defines the process as an MDP, which can be represented by the following equation:

$$s_{t+1} = \delta(s_t, a)$$

In the previous equation, δ represents the state function. We can summarize an MDP as follows:

1. The agent monitors the state of the environment and has a series of actions.
2. At a discrete time *t*, the agent detects the current state and decides to perform an action *at* $\in A$.
3. The environment reacts to this action by returning a reward $r_t = r(s_t, a_t)$ and moving to the state $s_{t+1} = \delta(s_t, a_t)$.

Important Note

The r and δ functions are characteristics of the environment that depend only on the current state and action. The goal of MDP is to learn a policy that, for each state of the system, provides the agent with an action that maximizes the total reward accumulated during the entire sequence of actions.

Now, let's analyze some of the terms that we introduced previously. They represent crucial concepts that help us understand Markovian processes.

The reward function

A reward function identifies the target in a Markovian process. It maps the states of the environment detected by the agent by enclosing them in a single number, which represents the reward. The purpose of this process is to maximize the total reward that the agent receives over the long term as a result of their choices. Then, the reward function collects the positive and negative results obtained from the actions chosen by the agent and uses them to modify the policy. If an action selected based on the indications provided by the policy returns a low reward, then the policy will be modified to select other actions. The reward function performs two functions: it stimulates the efficiency of the decisions and determines the degree of risk aversion of the agent.

Policy

A policy determines the agent's behavior in terms of decision-making. It maps both the states of the environment and the actions to be chosen in those states, which represent a set of rules or associations that respond to a stimulus. A policy is a fundamental part of a Markovian agent as it determines its behavior. In a Markov decision-making model, a policy provides a solution that associates a recommended action with each state potentially achievable by the agent. If the policy provides the highest expected utility among the possible actions, it is called an optimal policy (π^*). In this way, the agent does not have to keep their previous choices in memory. To make a decision, the agent only needs to execute the policy associated with the current state.

The state-value function

The state-value function provides us with the information necessary to evaluate the quality of a state for an agent. It returns the value of the expected goal that was obtained following the policy of each state, which is represented by the total expected reward. The agent depends on the policy in order to choose the actions to be performed.

Understanding the discounted cumulative reward

The goal of MDP is to learn a policy that guides an agent in choosing the actions to be performed for each state of the environment. This policy aims to maximize the total reward received during the entire sequence of actions performed by the agent. Let's learn how to maximize this total reward. The total reward that's obtained from adopting a policy is calculated as follows:

$$R_T = \sum_{i=0}^{T} r_{t+1} = r_t + r_{t+1} + \cdots + r_T$$

In the preceding equation, r_T is the reward of the action that brings the environment into the terminal state s_T.

To get the maximum total reward, we can select the action that provides the highest reward for each individual state, which leads to the choice of the optimal policy that maximizes the total reward.

Important Note

This solution is not applicable in all cases; for example, when the goal or terminal state is not achieved in a finite number of steps. In this case, both r_t and the sum of the rewards you want to maximize tend to infinity.

An alternative technique uses the discounted cumulative reward, which tries to maximize the following amount:

$$R_T = \sum_{i=0}^{\infty} \gamma^i * r_{t+1} = r_t + \gamma * r_{t+1} + \gamma^2 * r_{t+2} + \cdots$$

In the previous equation, γ is called the discount factor and represents the importance of future rewards. The discount factor is $0 \leq \gamma \leq 1$ and has the following conditions:

- **γ <1**: The sequence *rt* converges to a finite value.
- **γ = 0**: The agent does not consider future rewards, thereby trying to maximize the reward only for the current state.
- **γ = 1**: The agent will favor future rewards over immediate rewards.

The value of the discount factor may vary during the learning process to take special actions or states into account. An optimal policy may include individual actions that return low rewards, provided that the total reward is higher.

Comparing exploration and exploitation concepts

Upon reaching the goal, the agent looks for the most rewarded behavior. To do this, they must link each action to the reward returned. In the case of complex environments with many states, this approach is not feasible due to many action-reward pairs.

Important Note

This is the well-known exploration-exploitation dilemma: for each state, the agent explores all possible actions, exploiting the action most rewarded in achieving the objective.

Decision-making requires a choice between the two available approaches:

- **Exploitation**: The best decision is made based on current information
- **Exploration**: The best decision is made by gathering more information

The best long-term strategy can impose short-term sacrifices as this approach requires collecting adequate information to reach the best decisions.

In everyday life, we often find ourselves having to choose between two alternatives that, at least theoretically, lead to the same result: this approach is the exploration-exploitation dilemma. For example, let's say we need to decide whether to choose what we already know (exploitation) or choose something new (exploration). Exploitation keeps our knowledge unchanged, while exploration makes us learn more about the system. It is obvious that exploration exposes us to the risk of wrong choices.

Let's look at an example of using this approach in a real-life scenario– we must choose the best path to reach our trusted restaurant:

- **Exploitation**: Choose the path you already know.
- **Exploration**: Try a new path.

In complex problems, converging toward an optimal strategy can be too slow. In these cases, a solution to this problem is represented by a balance between exploration and exploitation.

An agent who acts exclusively based on exploration will always behave randomly in each state with a convergence to an optimal strategy that is practically impossible. On the contrary, if an agent acts exclusively based on exploitation, they will always use the same actions, which may not be optimal.

Introducing Markov chains

Markov chains are discrete dynamic systems that exhibit characteristics attributable to Markovian processes. These are finite state systems – finite Markov chains – in which the transition from one state to another occurs on a probabilistic, rather than deterministic, basis. The information available about a chain at the generic instant *t* is provided by the probabilities that it are in any of the states, and the temporal evolution of the chain is specified by specifying how these probabilities update by going from the instant *t* at instant *t + 1*.

Important Note

A Markov chain is a stochastic model in which the system evolves over time in such a way that the past affects the future only through the present: Markov chains have no memory of the past.

A random process characterized by a sequence of random variables $X = X_0, \ldots, X_n$ with values in a set $j_0, j_1, \ldots, j_n$ is given. This process is Markovian if the evolution of the process depends only on the current state, that is, the state after *n* steps. Using conditional probability, we can represent this process with the following equation:

$$P(X_{n+1} = j | X_0 = i_0, \ldots, X_n = i_n) = P(X_{n+1} = j | X_n = i_n)$$

If a discrete-time stochastic process *X* has a Markov property, it is called a Markov chain. A Markov chain is said to be homogeneous if the following transition probabilities do not depend on *n*, and only on *i* and *j*:

$$P(X_{n+1} = j | X_n = i)$$

In such hypotheses, let's assume we have the following:

$$p_{ij} = P(X_{n+1} = j | X_n = i)$$

All joint probabilities can be calculated by knowing the numbers p_{ij} and the following initial distribution:

$$p_i^0 = P(X_0 = i)$$

This probability represents the distribution of the process at zero time. The probabilities p_{ij} are called transition probabilities, and p_{ij} is the probability of transition from *i* to *j* in each time phase.

Transition matrix

The application of homogeneous Markov chains is easy by adopting the matrix representation. Through this, the formula expressed by the previous equation becomes much more readable. We can represent the structure of a Markov chain through the following transition matrix:

$$\begin{matrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n1} & p_{2n} & \cdots & p_{nn} \end{matrix}$$

This is a positive matrix in which the sum of the elements of each row is unitary. In fact, the elements of the *i*th row are the probabilities that the chain, being in the state S_i at the instant t, passes through S_1 *or* S_2,. . . *or* S_n at the next instant. Such transitions are mutually exclusive and exhaustive of all possibilities. Such a positive matrix with unit sum lines is stochastic. We will call each vector positive line x stochastic such that $T = [x_1\ x_2 . . . x_n]$, in which the sum of the elements assumes a unit value:

$$\sum_{i=1}^{n} x_i = 1$$

The transition matrix has the position (i, j) to pass from result i to result j by performing a single experiment.

Transition diagram

The transition matrix is not the only solution for describing a Markov chain. An alternative is an oriented graph called a transition diagram, in which the vertices are labeled by the states S_1, S_2, ..., S_n, and there is a direct edge connecting the vertex S_i to the vertex S_j if and only if the probability of transition from S_i to S_j is positive.

Important Note

The transition matrix and the transition diagram contain the same information necessary for representing the same Markov chain.

Let's take a look at an example: consider a Markov chain with three possible states – 1, 2, and 3 – represented by the following transition matrix:

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & 0 & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{4} & \frac{2}{4} \end{bmatrix}$$

As mentioned previously, the transition matrix contains the same information as the transition diagram. Let's learn how to draw this diagram. There are three possible states – 1, 2, and 3 – and the direct boundary from each state to other states shows the probabilities of transition p_{ij}. When there is no arrow from state *i* to state *j*, this means that $p_{ij} = 0$:

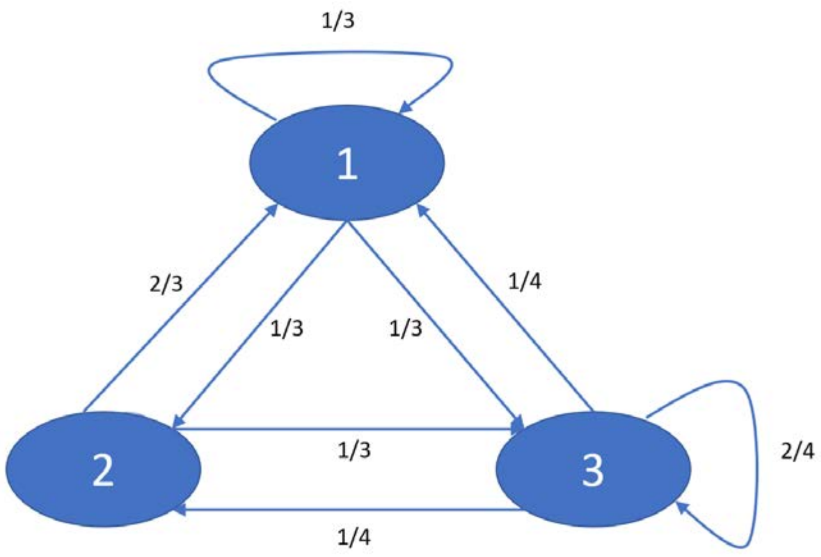


Figure 5.3 – Diagram of the transition matrix

In the preceding transition diagram, the arrows that come out of a state always add up to exactly 1, just like for each row in the transition matrix.

Markov chain applications

Now, let's look at a series of practical applications that can be made using Markov chains. We will introduce the problem and then analyze the Python code that will allow us to simulate how it works.

Introducing random walks

Random walks identify a class of mathematical models used to simulate a path consisting of a series of random steps. The complexity of the model depends on the system features we want to simulate, which are represented by the number of degrees of freedom and the direction. The authorship of the term is attributed to Karl Pearson who, in 1905, first referred to the term casual walk. In this model, each step has a random direction that evolves through a random process involving known quantities that follow a precise statistical distribution. The path that's traced over time will not necessarily be descriptive of real motion: it will simply return the evolution of a variable over time. This is the reason for the widespread use of this model in all areas of science: chemistry, physics, biology, economics, computer science, and sociology.

One-dimensional random walk

The one-dimensional casual walk simulates the movement of a punctual particle that is bound to move along a straight line, thus having only two movements: right and left. Each movement is associated with a random shift of one step to the right with a fixed probability *p* or to the left with a probability *q*. Each single step is the same length and is independent of the others. The following diagram shows the path to which the punctual particle is bound, along with the direction and the two vertices allowed:

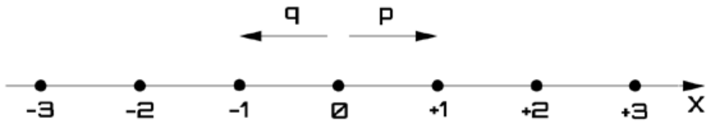


Figure 5.4 – One-dimensional walk

After n passes, the position of the point will be identified by its abscissa *X(n)*, characterized by a random term. Our aim is to calculate the probability with which the particle will return to the starting point, after *n* steps.

Important Note

The idea that the point will actually return to its starting position is not assured. To represent the position of the point on the straight line, we will adopt the X(n) variable, which represents the abscissa of the line after the particle has moved n steps: this variable is a discrete random variable with a binomial distribution.

The path of the point particle can be summarized as follows: for each instant, the particle moves one step to the right or left according to the value returned by a random variable *Z (n)*. This random variable takes only two dichotomous values:

- +1 with probability p > 0
- -1 with probability q

The two probabilities are related to each other through the following equation:

$$p + q = 1$$

Let's consider random variables Z_n with *n* = 1, 2, …. Suppose that these variables are independent and with equal distribution. The position of the particle at instant *n* will be represented by the following equation:

$$X_n = X_{n-1} + Z_n \quad ; \quad n = 1,2, \ldots$$

In the previous formula, X_n is the next value in the walk, X_{n-1} is the observation in the previous time phase, and Z_n is the random fluctuation in that step.

Important Note

The X_n variable identifies a Markov chain; that is, the probability that the particle in the next moment is in a certain position depends only on the current position, even if we know all the moments preceding the current one.

Simulating a one-dimensional random walk

The simulation of a casual walk does not represent a trivial succession of random numbers since the next step to the current one represents its evolution. The dependence between the next two steps guarantees a certain consistency from one passage to the next. This is not guaranteed in a banal generation of independent random numbers, which instead return big differences from one number to another. Let's learn how to represent the sequence of actions to be performed in a simple casual walking model through the following pseudocode:

1. Start from the 0 position.
2. Randomly select a dichotomous value (-1, 1).
3. Add this value to the previous time step.
4. Repeat *step 2* onward.

This simple iterative process can be implemented in Python by processing a list of 1,000 time steps for the random walk. Let's take a look:

1. Let's start by loading the necessary libraries:

```
from random import seed
from random import random
from matplotlib import pyplot
```

   The `random` module implements pseudo-random number generators for various distributions. The random module is based on the Mersenne Twister algorithm. Mersenne Twister is a pseudo-random number generator. Originally developed to produce inputs for Monte Carlo simulations, almost uniform numbers are generated via Mersenne Twister, making them suitable for a wide range of applications.

From the random module, two libraries were imported: seed and random. In this code, we will generate random numbers. To do this, we will use the random() function, which produces different values each time it is invoked. It has a very long period before any number is repeated. This is useful for producing unique values or variations, but there are times when it is useful to have the same dataset available for processing in different ways. This is necessary to ensure the reproducibility of the experiment. To do this, we can use the random.seed() function contained in the seed library. This function initializes the basic random number generator.

The matplotlib library is a Python library for printing high-quality graphics. With matplotlib, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and so on with a few commands. This is a collection of command-line functions like those provided by the MATLAB software.

2. Now, we will investigate the individual operations. Let's start with setting the seed:

```
seed(1)
```

The random.seed() function is useful if we wish to have the same set of data available to be processed in different ways as this makes the simulation reproducible.

Important Note

This function initializes the basic random number generator. If you use the same seed in two successive simulations, you will always get the same sequence of pairs of numbers.

3. Let's move on and initialize the crucial variable of the code:

```
RWPath= list()
```

The RWPath variable represents a list that will contain the sequence of values representative of the random walk. A list is an ordered collection of values, which can be of various types. It is an editable container, meaning that we can add, delete, and modify existing values. For our purposes, where we want to continuously update our values through the subsequent steps of the path, a list represents the most suitable solution. The list() function accepts a sequence of values and converts them into lists. With the preceding command, we simply initialized the list that is currently empty, and with the following code, we start to populate it:

```
RWPath.append(-1 if random() < 0.5 else 1)
```

The first value that we add to our list is a dichotomous value. It is simply a matter of deciding whether the value to be added is `1` or `-1`. The choice, however, is made on a random basis. Here, we generate a random number between 0 and 1 using the `random()` function and then check whether it is `<0.5`. If it is, then we add `-1`; otherwise, we add `1`. At this point, we will use an iterative cycle with a `for` loop, which will repeat the procedure for 1,000 steps:

```
for i in range(1, 1000):
```

At each step, we will generate a random term, as follows:

```
        ZNValue = -1 if random() < 0.5 else 1
```

Important Note

As we did when we chose the first value to add to the list, we generate a random value with the `random()` function, so if the value that's returned is lower than `0.5`, the `ZNValue` variable assumes the value `-1`; otherwise, `1`.

4. Now, we can calculate the value of the random walk at the current step:

```
XNValue = RWPath[i-1] + ZNValue
```

The `XNValue` variable represents the value of the abscissa on the current step. It is made up of two terms: the first represents the value of the abscissa in the previous state, while the second is the result of generating the random value. This value must be added to the list:

```
RWPath.append(XNValue)
```

This procedure will be repeated for the 1,000 steps that we want to perform. At the end of the cycle, we will have the entire sequence stored in the list.

5. Finally, we can visualize it through the following piece of code:

```
pyplot.plot(RWPath)
pyplot.show()
```

The `pyplot.plot()` function plots the values contained in the `RWPath` list on the *y* axis using *x* as an index array with the following value: *0..N-1*. The `plot()` function is extremely versatile and will take an arbitrary number of arguments.

Finally, the `pyplot.show()` function displays the graph that's created, as follows:

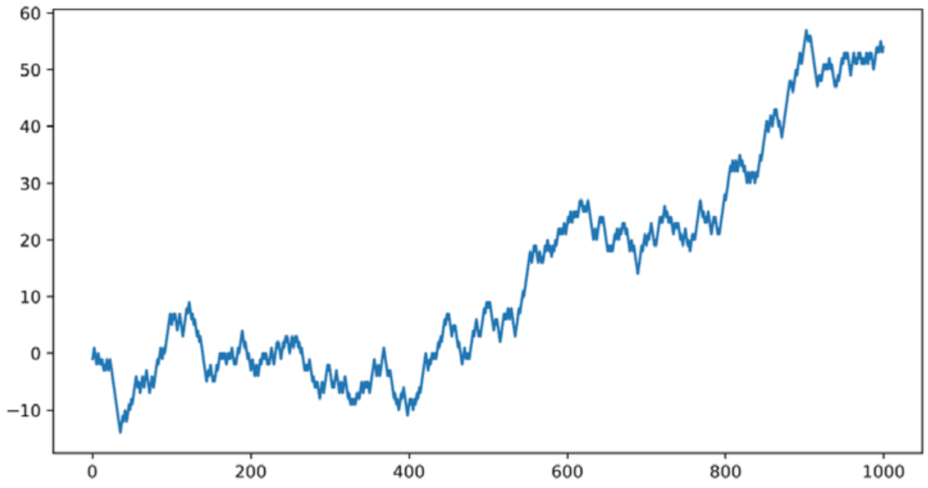


Figure 5.5 – The trend plot of the random walk path

In the previous graph, we can analyze the path followed by the point particle in a random process. This curve can describe the trends of a generic function, not necessarily associated with a road route. As anticipated, this process is configured as a Markovian process in that the next step is independent of the position from the previous step and depends only on the current step. The casual walk is a mathematical model widely used in finance. In fact, it is widely used to simulate the efficiency of information deriving from the markets: the price varies for the arrival of new information, which is independent of what we already know.

Simulating a weather forecast

Another potential application of Markov chains is in the development of a weather forecasting model. Let's learn how to implement this algorithm in Python. To start, we can work with a simplified model: we will consider only two climatic conditions/states, that is, sunny and rainy. Our model will assume that tomorrow's weather conditions will be affected by today's weather conditions, making the process take on Markovian characteristics. This link between the two states will be represented by the following transition matrix:

$$P = \begin{bmatrix} 0.80 & 0.20 \\ 0.25 & 0.75 \end{bmatrix}$$

The transition matrix returns the conditional probabilities P (A | B), which indicate the probability that event A occurs after event B has occurred. This matrix therefore contains the following conditional probabilities:

$$P = \begin{bmatrix} P(Sunny|Sunny) & P(Sunny|Rainy) \\ P(Rainy|Sunny) & P(Rainy|Rainy) \end{bmatrix}$$

In the previous transition matrix, each row contains a complete distribution. Therefore, all the numbers must be non-negative and the sum must be equal to 1. The climatic conditions show a tendency to resist change. For this reason, after a sunny day, the probability of another sunny – *P (Sunny | Sunny)* – day is greater than a rainy – *P (Sunny | Rainy)* day. The climatic conditions of tomorrow are not directly related to those of yesterday; it follows that the process is Markovian. The previous transition matrix is equivalent to the following:

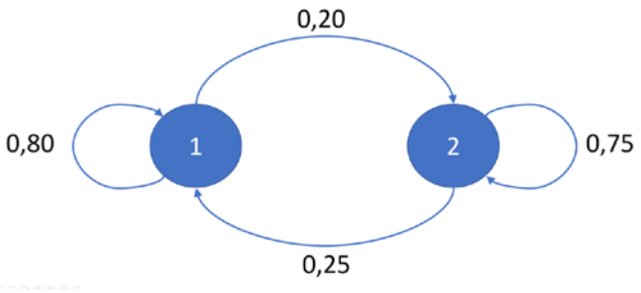


Figure 5.6 – Transition diagram

The simulation model we want to elaborate on will have to calculate the probability that it will rain in the next few days. It will also have to allow you to recover a statistic of the proportion of sunny and rainy days in a certain period of time. The process, as mentioned previously, is Markovian, and the tools we analyzed in the previous sections allow us to obtain the requested information. Let's get started:

1. Let's see the Python code that alternates sunny and rainy days, starting from a specific initial condition. As always, we will analyze it line by line, starting with loading the necessary libraries:

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

   The `numpy` library is a Python library that contains numerous functions that can help us manage multidimensional matrices. Furthermore, it contains a large collection of high-level mathematical functions we can use on these matrices. We will use two functions: `random.seed()` and `random.choose()`.

The `matplotlib` library is a Python library for printing high-quality graphics. With `matplotlib`, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and so on with a few commands. It is a collection of command-line functions like those provided by the MATLAB software. Let's move on and illustrate the code:

```
np.random.seed(3)
```

The `random.seed()` function initializes the seed of the random number generator. In this way, the simulation that uses random numbers will be reproducible. The reproducibility of the experiment will be guaranteed by the fact that the random numbers that will be generated will always be the same.

2. Now, let's define the possible states of the weather conditions:

```
StatesData = ['Sunny','Rainy']
```

Two states are provided: sunny and rainy. The transitions matrix representing the transition between the weather conditions will be set as follows:

```
TransitionStates = [['SuSu','SuRa'],['RaRa','RaSu']]
TransitionMatrix = [[0.80,0.20],[0.25,0.75]]
```

The transition matrix returns the conditional probabilities $P(A \mid B)$, which indicate the probability that event A occurs after event B has occurred. All the numbers in a row must be non-negative and the sum must be equal to 1. Let's move on and set the variable that will contain the list of state transitions:

```
WeatherForecasting = list()
```

The `WeatherForecasting` variable will contain the results of the weather forecast. This variable will be of the `list` type.

Important Note

A list is an ordered collection of values and can be of various types. It is an editable container and allows us to add, delete, and modify existing values.

For our purposes, which is to continuously update our values through the subsequent steps of the path, the list represents the most suitable solution. The `list()` function accepts a sequence of values and converts them into lists.

3. Now, we decide on the number of days for which we will predict the weather conditions:

```
NumDays = 365
```

4. For now, we have decided to simulate the weather forecast for a 1-year time horizon; that is, 365 days. Let's fix a variable that will contain the forecast of the current day:

```
TodayPrediction = StatesData[0]
```

5. Furthermore, we also initialized it with the first vector value containing the possible states. This value corresponds to the `Sunny` condition. We print this value on the screen:

```
print('Weather initial condition =',TodayPrediction)
```

6. At this point, we can predict the weather conditions for each of the days set by the `NumDays` variable. To do this, we will use a `for` loop that will execute the same piece of code several times equal to the number of days that we have set in advance:

```
for i in range(1, NumDays):
```

7. Now, we will analyze the main part of the entire program. Within the `for` loop, the forecast of the time for each consecutive day occurs through an additional conditional structure: the `if` statement. Starting from a meteorological condition contained in the `TodayPrediction` variable, we must predict that of the next day. We have two conditions: sunny and rainy. In fact, there are two control conditions, as shown in the following code:

```
if TodayPrediction == 'Sunny':
        TransCondition = np.random.
choice(TransitionStates[0],replace=True,
p=TransitionMatrix[0])
        if TransCondition == 'SuSu':
            pass
        else:
            TodayPrediction = 'Rainy'

 elif TodayPrediction == 'Rainy':
        TransCondition = np.random.
choice(TransitionStates[1],replace=True,
p=TransitionMatrix[1])
```

```
            if TransCondition == 'RaRa':
                pass
            else:
                TodayPrediction = 'Sunny'
```

8. If the current state is `Sunny`, we use the `numpy random.choice()` function to forecast the weather condition for the next state. A common use for random number generators is to select a random element from a sequence of enumerated values, even if these values are not numbers. The `random.choice()` function returns a random element of the non-empty sequence passed as an argument. Three arguments are passed:

   `TransitionStates[0]`: The first row of the transition states

   `replace=True`: The sample is with a replacement

   `p=TransitionMatrix[0]`: The probabilities associated with each entry in the state passed

   The `random.choise()` function returns random samples of the `SuSu`, `SuRa`, `RaRa`, and `RaSu` types, according to the values contained in the `TransitionStates` matrix. The first two will be returned starting from sunny conditions and the remaining two starting from rain conditions. These values will be stored in the `TransCondition` variable.

   Within each `if` statement, there is an additional `if` statement. This is used to determine whether to update the current value of the weather forecast or to leave it unchanged. Let's see how:

```
  if TransCondition == 'SuSu':
                pass
            else:
                TodayPrediction = 'Rainy'
```

   If the `TransCondition` variable contains the `SuSu` value, the weather conditions of the current day remain unchanged. Otherwise, it is replaced by the `Rainy` value. The `elif` clause performs a similar procedure, starting from the rain condition. At the end of each iteration of the `for` loop, the list of weather forecasts is updated, and the current forecast is printed:

```
  WeatherForecasting.append(TodayPrediction)
  print(TodayPrediction)
```

   Now, we need to predict the weather forecast for the next 365 days.

9. Let's draw a graph with the sequence of forecasts for the next 365 days:

```
plt.plot(WeatherForecasting)
plt.show()
```

The following graph is printed:

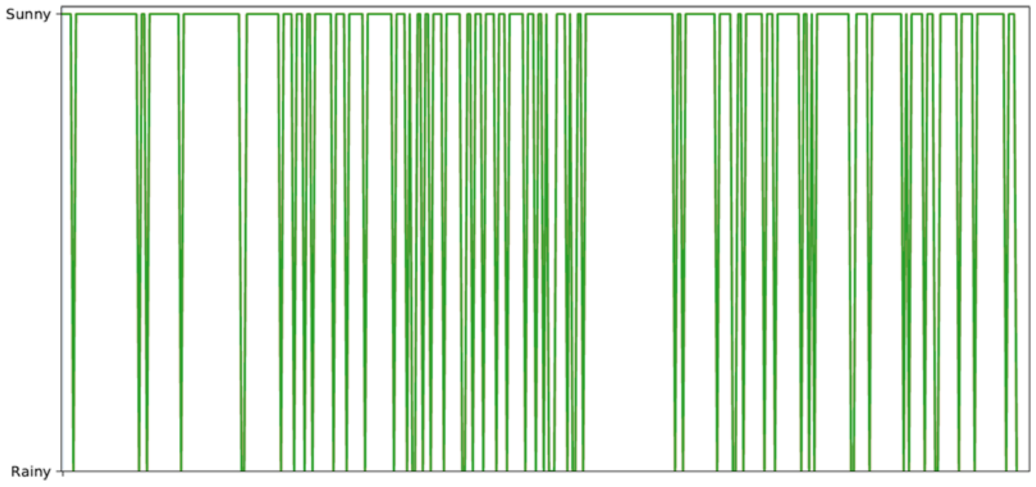


Figure 5.7 – Plot of the weather forecast

Here, we can see that the forecast of sunny days prevails over the rainy ones.

Important Note

The flat points at the top represent all the sunny days, while the dips in-between are the rainy days.

10. To quantify this prevalence, we can draw a histogram. In this way, we will be able to count the occurrences of each condition:

```
plt.figure()
plt.hist(WeatherForecasting)
plt.show()
```

The following is a histogram of the weather condition for the next 365 days:

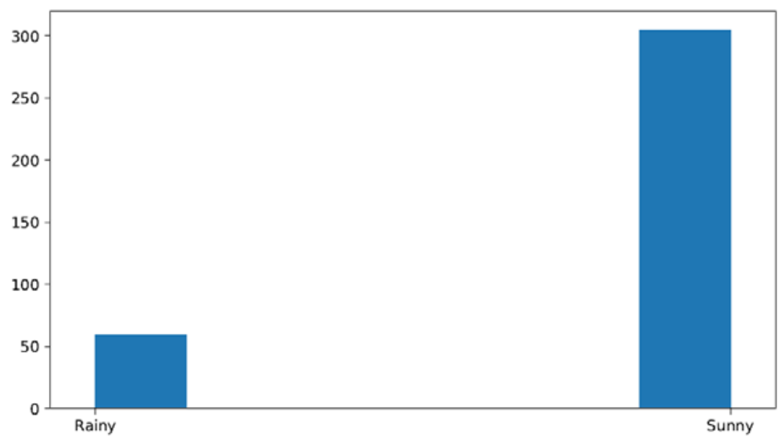


Figure 5.8 – Histogram of the weather forecast

With this, we can confirm the prevalence of sunny days. The result we've obtained derives from the transition matrix. In fact, we can see that the probability of the persistence of a solar condition is greater than that of rain. In addition, the initial condition has been set to the sunny condition. We can also try to see what happens when the initial condition is set to the rain condition.

The Bellman equation explained

In 1953, Richard Bellman introduced the principles of dynamic programming in order to efficiently solve sequential decision problems. In this type of problem, decisions are periodically implemented and influence the size of the model. In turn, these influence future decisions. The principle of optimality, enunciated by Bellman, allows, through an intelligent application, you to efficiently deal with the complexity of the interaction between the decisions and the sizes of the model. Dynamic programming techniques were also applied from the outset to problems in which there is no temporal or sequential aspect.

Important Note

Although dynamic programming can be applied to a wide range of problems by providing a common abstract model, from a practical point of view, many problems require models of such dimensions to preclude, then as now, any computational approach. This inconvenience was then called the 'curse of dimensionality' and was an anticipation, in still informal terms, of concepts of computational complexity.

The greatest successes of dynamic programming have been obtained in the context of sequential decision models, especially of the stochastic type, such as Markovian decision processes, but also in some combinatorial models.

Dynamic programming concepts

Dynamic Programming (**DP**) is a programming technique designed to calculate an optimal policy based on a perfect model of the environment in the form of an **MDP**. The basis of dynamic programming is the use of state values and action values in order to identify good policies.

DP methods are applied to Markov decision-making processes using two processes called policy evaluation and policy improvement, which interact with each other:

- **Policy evaluation** is done through an iterative process that seeks to solve Bellman's equation. The convergence of the process for $k \rightarrow \infty$ imposes approximation rules, thus introducing a stop condition.
- **Policy improvement** improves the policy based on current values.

In the policy iteration technique, the two phases just described alternate, and each concludes before the other begins.

> **Important Note**
> The iterative process when evaluating policies obliges us to evaluate a policy at each step through an iterative process whose convergence is not known a priori and depends on the starting policy. To address this problem, we can stop evaluating the policy at some point, while still ensuring that we converge to an optimal value.

Principle of optimality

The validity of the dynamic optimization procedure is ensured by Bellman's principle of optimality: *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Based on this principle, it is possible to divide the problem into stages and solve the stages in sequence, using the dynamically determined values of the objective function, regardless of the decisions that led to them. This allows us to optimize one stage at a time, reducing the initial problem to a sequence of smaller subproblems therefore easier to solve.

The Bellman equation

Bellman's equation helps us solve MDP by finding the optimal policy and value functions. The optimal value function V * (S) is the one that returns the maximum value of a state. This maximum value is the one corresponding to the action that maximizes the reward value of the optimal action in each state. It then adds a discount factor multiplied by the value of the next state by the Bellman equation, through a recursive procedure. The following is an example of a Bellman equation:

$$V(s) = max_a(R(s,a) + \gamma * V(s'))$$

In the previous equation, we have the following:

- $V(s)$ is the function value at state *s*.
- $R(s,a)$is the reward we get after acting a in state *s*.
- γ is the discount factor.
- $V(s^{'})$ is the function value at the next state.

For a stochastic system, when we take an action, it is not said that we will end up in a later state, but that we can only indicate the probability of ending up in that state.

Multi-agent simulation

An agent can be defined as anything that is able to perceive an environment through sensors and act in it through actuators. Artificial intelligence focuses on the concept of a rational agent, or an agent who always tries to optimize an appropriate performance measure. A rational agent can be a human agent, a robotic agent, or a software agent. In the following diagram, we can see the interaction between the agent and the environment:

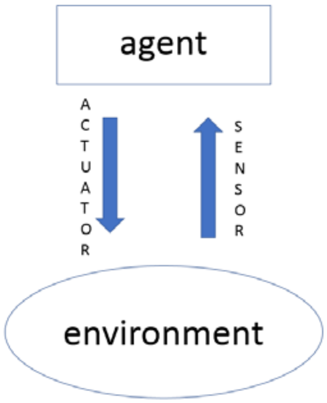


Figure 5.9 – Interaction between the agent and the environment

An agent is considered autonomous when it can flexibly and independently choose the actions to be taken to achieve its goals, without constantly resorting to the intervention of an external decision system. Note that, in most complex domains, an agent can only partially obtain information and have control in the environment that it has been inserted into, thus exerting, at most, a certain influence on it.

An agent can be considered autonomous and intelligent if it has the following characteristics:

- **Reactivity**: It must perceive the environment, managing to adapt in good time to the changes that take place.
- **Proactivity**: It must show behavior oriented toward achieving the objectives set with initiative.
- **Social skills**: It must be able to interact with other agents for the pursuit of their goals.

There are numerous situations where multiple agents coexist in the same environment and interact with each other in different ways. In fact, it is quite rare for an agent to represent an isolated system. We can define a **Multi-agent System** (**MAS**) as a group of agents that can potentially interact with each other. A MAS can be competitive, which is where each agent tries to exclusively maximize their own interests, even at the expense of those of others, rather than cooperative, which is where agents are willing to give up part of their objectives in an attempt to maximize the global utility of the system.

The types of possible interactions are as follows:

- **Negotiation**: This occurs when agents must seek an agreement on the value to be assigned to some variables.
- **Cooperation**: This occurs when there are common goals for which agents try to align and coordinate their actions.
- **Coordination**: This is a type of interaction aimed at avoiding situations of conflict between agents.

The use of MAS systems introduces a series of advantages:

- **Efficiency and speed**: Thanks to the possibility of performing computations in parallel.
- **Robustness**: The system can overcome single-agent failures.

- **Flexibility**: Adding new agents to the system is extremely easy.
- **Modularity**: Extremely useful in the software design phase due to the possibility of reusing the code.
- **Cost**: The single unit-agent has a very low cost compared to the overall system.

The growing attention that's being paid to multi-agent systems for the treatment of problems based on decision-making processes is linked to some characteristics that distinguish them, such as flexibility and the possibility of representing independent entities through distinct computational units that interact with each other. The various stakeholders of a decision-making system can, in fact, be modeled as autonomous agents. Several practical real-world applications have recently adopted an approach based on problems such as satisfying and optimizing distributed constraints and identifying regulations that have an intermediate efficiency between the centralized (optimal) and the non-coordinated (bad) ones.

Summary

In this chapter, we learned the basic concepts of the Markov process. This is where the future evolution of the process depends only on the instant of observation of the system and in no way depends on the past. We have seen how an agent and the surrounding environment interact and the elements that characterize its actions. We now understand the reward and policy concepts behind decision-making. We then went on to explore Markov chains by analyzing the matrices and transition diagrams that govern their evolution.

Then, we addressed some applications in order to put the concepts we'd learned about into practice. We dealt with a casual walk and a forecast model of weather conditions by adopting an approach based on Markov chains. Next, we studied Bellman equations as coherence conditions for optimal value functions to determine optimal policy. Finally, we introduced multi-agent systems, which allow us to consider different stakeholders in a decision-making process.

In the next chapter, we will understand how to obtain robust estimates of confidence intervals and standard errors of population parameters, as well as how to estimate the distortion and standard error of a statistic. We will then discover how to perform a test for statistical significance and how to validate a forecast model.

6 Resampling Methods

Resampling methods are one of the most interesting inferential applications of stochastic simulations and random numbers. They are particularly useful in the nonparametric field, where the traditional inference methods cannot be correctly applied. They generate random numbers to be assigned to random variables or random samples. They require machine time related to the growth of repeated operations. They are very simple to implement and once implemented, they are automatic. Selecting the required elements must provide a sample that is, or at least can be, representative of the population. To achieve this, all the characteristics of the population must be included in the sample. In this chapter, we will try to extrapolate the results obtained from the representative sample of the entire population. Given the possibility of making mistakes in this extrapolation, it will be necessary to evaluate the degree of accuracy of the sample and the risk of arriving at incorrect predictions. In this chapter, we will learn how to apply resampling methods to approximate some characteristics of the distribution of a sample in order to validate a statistical model. We will analyze the basics of the most common resampling methods and learn how to use them by solving some practical cases.

In this chapter, we're going to cover the following main topics:

- Introducing resampling methods
- Exploring the jackknife technique
- Demystifying bootstrapping

- Explaining permutation tests
- Approaching the cross-validation technique

Technical requirements

In this chapter, we will address resampling method technologies. In order to deal with the topics in this chapter, it is necessary that you have a basic knowledge of algebra and mathematical modeling. To work with the Python code in this chapter, you'll need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `JackknifeEstimator.py`
- `BootstrapEstimator.py`
- `KfoldCrossValidation.py`

Introducing resampling methods

Resampling methods are a set of techniques based on the use of subsets of data, which can be extracted either randomly or according to a systematic procedure. The purpose of this technology is to approximate some characteristics of the sample distribution – a statistic, a test, or an estimator – to validate a statistical model.

Resampling methods are one of the most interesting inferential applications of stochastic simulations and the generation of random numbers. These methods became widespread during the 1960s, originating from the basic concepts of Monte Carlo methods. The development of Monte Carlo methods took place mainly in the 1980s, following the progress of information technology and the increase in the power of computers. Their usefulness is linked to the development of non-parametric methods, in situations where the methods of classical inference cannot be correctly applied.

The following details can be observed from resampling methods:

- They repeat simple operations many times.
- They generate random numbers to be assigned to random variables or random samples.
- They require more machine time as the number of repeated operations grows.
- They are very simple to implement and once implemented, they are automatic.

Over time, various resampling methods have been developed and can be classified based on some characteristics.

Important note

A first classification can be made between methods based on randomly extracting subsets of sample data and methods in which resampling occurs according to a non-randomized procedure.

Further classification can be performed as follows:

- The bootstrap method and its variants, such as subsampling, belong to the random extraction category.
- Procedures such as Jackknife and cross-validation fall into the non-randomized category.
- Statistical tests, called permutation or exact tests, are also included in the family of resampling methods.

Sampling concepts overview

Sampling is one of the fundamental topics of all statistical research. Sampling generates a group of elementary units, that is, a subset of a population, with the same properties as the entire population, at least with a defined risk of error.

By population, we mean the set, finite or unlimited, of all the elementary units to which a certain characteristic is attributed, which identifies them as homogeneous.

Important Note

For example, this could be the population of temperature values in each place, in a time span that can be daily, monthly, or yearly.

Sampling theory is an integral and preparatory part of statistical inference, along with the resulting sampling techniques, and allows us to identify the units whose variables are to be analyzed.

Statistical sampling is a method used to randomly select items so that every item in a population has a known, non-zero probability of being included in the sample. Random selection is considered a powerful means of building a representative sample that, in its structure and diversity, reflects the population under consideration. Statistical sampling allows us to obtain an objective sample: the selection of an element does not depend on the criteria defined for reasons of research convenience or availability and does not systematically exclude and favor any group of elements within a population.

In random sampling, associated with the calculation of probabilities, the following actions are performed:

- Extrapolation of results through mathematical formulas and an estimate of the associated error
- Control over the risk of reaching an opposite conclusion to reality
- Calculation – through a formula – of the minimum sample size necessary to obtain a given level of accuracy and precision

Reasoning about sampling

Now, let's learn why it may be preferable to analyze the data of a sample rather than that of the entire population:

- Consider a case in which the statistical units do not present variability. Here, it is useless to make many measurements because the population parameters are determined with few measurements. For example, if we wanted to determine the average of 1,000 identical statistical units, this value would be equal to that obtained if we only considered 10 units.

> **Important Note**
> Sampling is used if not all the elements of the population are available. For example, investigations into the past can only be done on available historical data, which is often incomplete.

- Sampling is indicated when there is a considerable amount of time being saved when achieving results. This is because even if electronic computers are used, the data-entry phase is significantly reduced if the investigation is limited to a few elements of the overall population.

Pros and cons of sampling

When information is collected, a survey is performed on all the units that make up the population under study. When an analysis is carried out on the information collected, it is possible to use it only on part of the units that make up the population.

The pros of sampling are as follows:

- Cost reduction
- Reduction of time
- Reduction of the organizational load

The disadvantage of sampling is as follows:

- The sampling base is not always available or easy to know

Sampling can be performed by forced choice in cases where the reference population is partially unknown in terms of composition or size. Sampling cannot always replace a complete investigation, such as in the case of surveys regarding the movement of marital status, births, and deaths: all individual cases must be known.

Probability sampling

In probability sampling, the probability that each unit of the population must be extracted is known. In contrast, in non-probability sampling, the probability that each unit of the population must be extracted is not known.

Let's take a look at an example. If we extract a sample of university students by drawing lots from those present on any day in university, we do not get a probabilistic sample for the following reasons: non-attending students have no chance of entering, and the students who attend the most are more likely to be extracted than the other students of the following years.

How sampling works

The sampling procedure involves a series of steps that need to be followed appropriately in order to extract data that can adequately represent the population. Sampling is carried out as follows:

1. Define the objective population in the detection statistics.
2. Define the sampling units.
3. Establish the size of the sample.
4. Choose the sample or samples on which the load will be statistically detected according to a method of sampling.
5. Finally, formulate a judgment on the goodness of the sample.

Exploring the Jackknife technique

This method is used to estimate characteristics such as the distortion and the standard deviation of a statistic. This technique allows us to obtain the desired estimates without necessarily resorting to parametric assumptions. Jackknife is based on calculating the statistics of interest for the sub-samples we've obtained, leaving out one sample observation at a time. The jackknife estimate is consistent for various sample statistics, such as mean, variance, correlation coefficient, maximum likelihood estimator, and others.

Defining the Jackknife method

The Jackknife method was proposed in 1949 by M. H. Quenouille who, due to the low computational power of the time, created an algorithm that requires a fixed number of accounts.

Important Note

The main idea behind this method is to cut a different observation from the original sample each time and to re-evaluate the parameter of interest. The estimate will be compared with the same one that was calculated on the original sample.

Since the distribution of the variable is not known, the distribution of the estimator is not known either.

Jackknife samples are constructed by leaving an observation x_i out of the original sample each time, as shown in the following equation:

$$x_i = (x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$$

Then, *n* samples of size *m* = n-1 are obtained. Let's take a look at an example. Consider a sample of size n = 5 that produces five Jackknife samples of size m = 4, as follows:

$$\begin{aligned}
x_{(1)} &= (x_2, x_3, x_4, x_5)\\
x_{(2)} &= (x_1, x_3, x_4, x_5)\\
x_{(3)} &= (x_1, x_2, x_4, x_5)\\
x_{(4)} &= (x_1, x_2, x_3, x_5)\\
x_{(5)} &= (x_1, x_2, x_3, x_4)
\end{aligned}$$

The pseudo-value $\hat{\theta}$ is recalculated on the generic ith sample Jackknife. The procedure is iterated n times on each of the available Jackknife samples:

$$x_{(1)} = (x_2, x_3, x_4, x_5) \rightarrow \hat{\theta}_{(1)}$$
$$x_{(2)} = (x_1, x_3, x_4, x_5) \rightarrow \hat{\theta}_{(2)}$$
$$x_{(3)} = (x_1, x_2, x_4, x_5) \rightarrow \hat{\theta}_{(3)}$$
$$x_{(4)} = (x_1, x_2, x_3, x_5) \rightarrow \hat{\theta}_{(4)}$$
$$x_{(5)} = (x_1, x_2, x_3, x_4) \rightarrow \hat{\theta}_{(5)}$$

The following diagram shows this preliminary procedure:

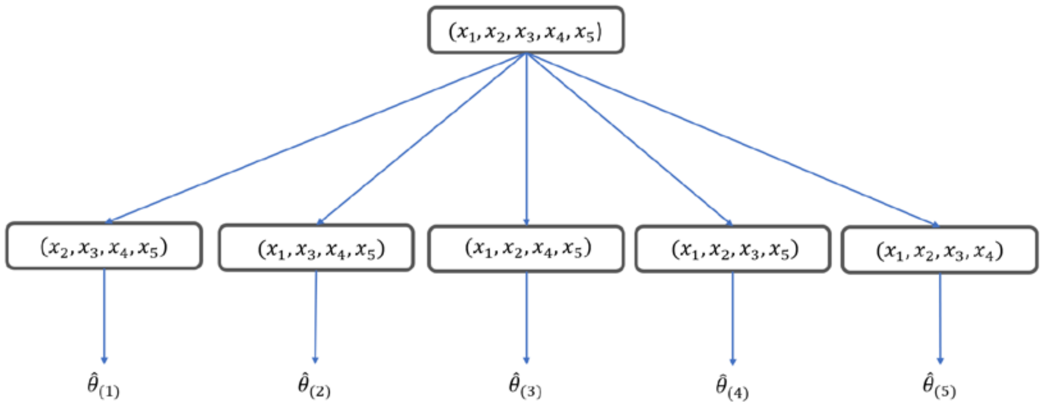


Figure 6.1 – Representation of the Jackknife method

To calculate the variance of the Jackknife estimate, the following equation will be used:

$$Variance_{Jakknife} = \sqrt{\frac{n-1}{n} \sum_{i=1}^{n} (\hat{\theta}_{(i)} - \hat{\theta}_{(.)})^2}$$

In the previous equation, the term $\hat{\theta}_{(.)}$ is defined as follows:

$$\hat{\theta}_{(.)} = \frac{1}{n}\sum_{i=1}^{n} \hat{\theta}_{(i)}$$

The calculated standard deviation will be used for building confidence intervals for the parameters.

With the aim of evaluating, and possibly reducing, the estimator distortion, the Jackknife estimate of the distortion is calculated as follows:

$$\hat{\theta}_i^* = n * \hat{\theta} - (n-1) * \hat{\theta}_i$$

Essentially, the Jackknife method reduces bias and evaluates variance for an estimator.

Estimating the coefficient of variation

To make comparisons regarding variability between different distributions, we can use the **coefficient of variation** (**CV**) since it considers the average of the distribution. The variation coefficient is a relative measure of dispersion and is a dimensionless magnitude. It allows us to evaluate the dispersion of the values around the average, regardless of the unit of measurement.

Important Note

For example, the standard deviation of a sample of income expressed in dollars is completely different from the standard deviation of the same income expressed in euros, while the dispersion coefficient is the same in both cases.

The coefficient of variation is calculated using the following equation:

$$CV = \frac{\sigma}{|\mu|} * 100$$

In the previous equation, we use the following parameters:

- σ is the standard deviation of the distribution.
- $|\mu|$ is the absolute value of the mean of the distribution.

The variance is the average of the differences squared between each of the observations in a group of data and the arithmetic mean of the data:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

So, it represents the squared error that we commit, on average, replacing a generic observation x_i with the average μ. The standard deviation is the square root of the variance and therefore represents the square root of the mean squared error:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

The CV, which can be defined starting from the average and standard deviation, is the appropriate index for comparing the variability of two characters. CV
is particularly useful when you want to compare the dispersion of data with different units of measurement or with different ranges of variation.

Applying Jackknife resampling using Python

Now, let's look at some Python code that compares the CV of a distribution and the one obtained with resampling according to the Jackknife method:

1. Let's see the code step by step, starting with loading the necessary libraries:

```
import random
import statistics
import matplotlib.pyplot as plt
```

The `random` module implements pseudo-random number generators for various distributions. The `random` module is based on the Mersenne Twister algorithm. Mersenne Twister is a pseudo-random number generator. Originally developed to produce inputs for Monte Carlo simulations, almost uniform numbers are generated via Mersenne Twister, making them suitable for a wide range of applications.

The `statistics` module contains numerous functions for calculating mathematical statistics from numerical data. With the tools available in this module, it will be possible to calculate the averages and make measurements of the central position and diffusion measures.

The `matplotlib` library is a Python library for printing high-quality graphics. With `matplotlib`, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and so on with a few commands. This is a collection of command-line functions like those provided by the MATLAB software.

2. We now generate a distribution that represents our data population. We will use this data to extract samples using the sampling methods we are studying. To do this, we first create an empty list that will contain such data:

```
PopData = list()
```

A list is an ordered collection of values and can be of various types. It is an editable container – in fact, it allows us to add, delete, and modify existing values. For our purpose, which is to continuously update our values, the list represents the most suitable solution. The `list()` function accepts a sequence of values and converts them into lists. With this command, we simply initialized the list, which is currently empty.

3. The list will be populated through the generation of random numbers. Then, to make the experiment reproducible, we will fix the seed in advance:

```
random.seed(5)
```

The `random.seed ()` function is useful if we want to have the same set of data available to be processed in different ways as this makes the simulation reproducible.

Important Note

This function initializes the basic random number generator. If you use the same seed in two successive simulations, you always get the same sequence of pairs of numbers.

4. Now, we can populate the list with 100 randomly generated values:

```
for i in range(100):
    DataElem = 10 * random.random()
    PopData.append(DataElem)
```

In the previous piece of code, we generated 100 random numbers between 0 and 1 using the `random()` function. Then, for each step of the `for` loop, this number was multiplied by 10 to obtain a distribution of numbers between 0 and 10.

5. Now, let's define a function that calculates the coefficient of variation, as follows:

```
def CVCalc(Dat):
    CVCalc = statistics.stdev(Dat)/statistics.mean(Dat)
    return CVCalc
```

   As indicated in the *Estimating the coefficient of variation* section, this coefficient is simply the ratio between the standard deviation and the mean. To calculate the standard deviation, we used the `statistics.stdev()` function. This function calculates the sample standard deviation, which represents the square root of the sample variance. To calculate the mean of the data, we used the `statistics.mean` function. This function calculates the sample arithmetic mean of the data. We can immediately use the newly created function to calculate the variation coefficient of the distribution that we have created:

```
CVPopData = CVCalc(PopData)
print(CVPopData)
```

   The following result is returned:

```
0.6569398125747403
```

   For now, we leave this result out, but we will use it later to compare the results we obtained by resampling.

6. Now, we can move on and resample according to the Jackknife method. To begin, we fix the variables that we will need in the following calculations:

```
N = len(PopData)
JackVal = list()
PseudoVal = list()
```

   `N` represents the number of samples present in the starting distribution. The `JackVal` list will contain the `Jackknife` sample, while the `PseudoVal` list will contain the `Jackknife` pseudo values.

7. The two newly created lists must be initialized to zero to avoid problems in subsequent calculations:

```
for i in range(N-1):
    JackVal.append(0)
for i in range(N):
    PseudoVal.append(0)
```

The `JackVal` list has a length of *N-1* and relates to what we discussed in the *Defining the Jackknife method* section.

8. At this point, we have all the tools necessary to apply the Jackknife method. We will use two `for` loops to extract the samples from the initial distribution by calculating the pseudo value at each step of the external loop:

```
for i in range(N):
    for j in range(N):
        if(j < i):
            JackVal[j] = PopData[j]
        else:
            if(j > i):
                JackVal[j-1]= PopData[j]
    PseudoVal[i] = N*CVCalc(PopData)-
                        (N-1)*CVCalc(JackVal)
```

Jackknife samples (`JackVal`) are constructed by leaving an observation xi out of the original sample at each step of the external loop (for *i* in `range(N)`). At the end of each step of the external cycle, the pseudo value is evaluated using the following equation:

$$\hat{\theta}_i^* = n * \hat{\theta} - (n-1) * \hat{\theta}_i$$

9. To analyze the distribution of pseudo values, we can draw a histogram:

```
plt.hist(PseudoVal)
plt.show()
```

The following graph is printed:

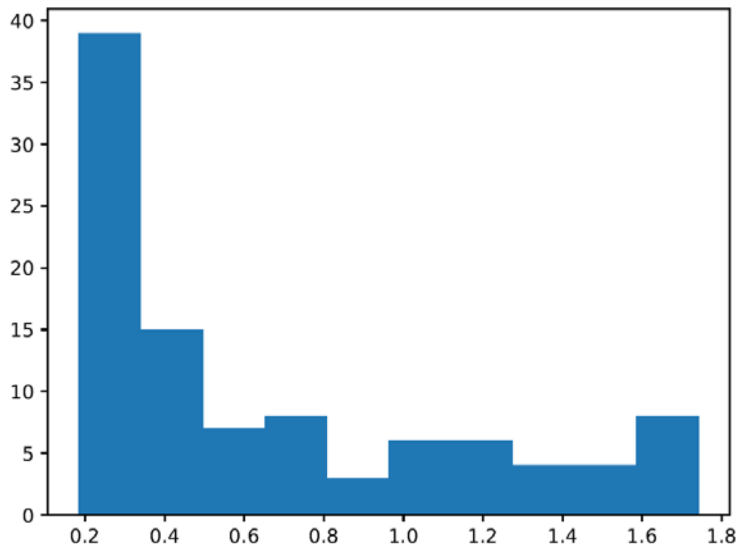


Figure 6.2 – Distribution of pseudo values

10. Now, let's calculate the average of the pseudo values that we have obtained:

```
MeanPseudoVal=statistics.mean(PseudoVal)
print(MeanPseudoVal)
```

The following result is returned:

```
0.6545985339842991
```

As we can see, the value we've obtained is comparable with what we obtained from the starting distribution. Now, we will calculate the variance of the pseudo values:

```
VariancePseudoVal=statistics.variance(PseudoVal)
print(VariancePseudoVal)
```

The following result is returned:

```
0.2435929299444099
```

Finally, let's evaluate the variance of the Jackknife estimator:

```
VarJack = statistics.variance(PseudoVal)/N
print(VarJack)
```

The following result is returned:

```
0.002435929299444099
```

We will use these results to compare the different resampling methods.

Demystifying bootstrapping

The most well-known resampling technique is the one defined as bootstrapping, as introduced by B. Efron in 1993. The logic of the bootstrap method is to build samples that are not observed, but statistically like those observed. This is achieved by resampling the observed series through an extraction procedure where we reinsert the observations.

Introducing bootstrapping

This procedure is like extracting a number from an urn, with subsequent reinsertion of the number before the next extraction. Once a statistical test has been chosen, it is calculated both on the observed sample and on a large number of samples of the same size as that observed and obtained by resampling. The *N* values of the test statistic then allow us to define the sample distribution; that is, the empirical distribution of the chosen statistic.

Important Note

A statistical test is a rule for discriminating samples that, if observed, lead to the rejection of an initial hypothesis, from those which, if observed, lead to accepting the same hypothesis until proven otherwise.

Since the bootstrapped samples derive from a random extraction process with reintegration from the original series, any temporal correlation structure of the observed series is not preserved. It follows that bootstrapped samples have properties such as the observed sample, but respect, at least approximately, the hypothesis of independence. This makes them suitable for calculating test statistics distributions, assuming there's a null hypothesis for the absence of trends, change points, or of a generic systematic temporal trend.

Once the sample distribution of the generic test statistic under the null hypothesis is known, it is possible to compare the value of the statistic itself, as calculated on the observed sample with the quantiles, deduced from the sample distribution, and check whether the value falls into critical regions with a significance level of 5% and 10%, respectively. Alternatively, you can define the percentage of times that the value of the statistic calculated on the observed sample is exceeded by the values coming from the N samples. This value is the statistic p-value for the observed sample and checks whether this percentage is far from the commonly adopted meaning of 5% and 10%.

Bootstrap definition problem

Bootstrap is a statistical resampling technique with reentry so that we can approximate the sample distribution of a statistic. It therefore allows us to approximate the mean and variance of an estimator so that we can build confidence intervals and calculate test p-values when the distribution of the statistics of interest is not known.

Important Note

Bootstrap is based on the fact that the only available sample is used to generate many more samples and to build the theoretical reference distribution. Use the data from the original sample to calculate a statistic and estimate its sample distribution without making any assumptions about the distribution model.

The plug-in principle is used to generate the distribution; that is, the estimate of θ is constructed by substituting the empirical equivalent of the unknown distribution function of the population. The distribution function of the sample is obtained by constructing a distribution of frequencies of all the values it can assume in that experimental situation.

In the simple case of simple random sampling, the operation is as follows. Consider an observed sample with *n* elements, as described by the following equation:

$$x = (x_1, \ldots, x_n)$$

From this distribution, m other samples of a constant number equal to *n*, say *x* * 1, ..., *x* * *m*, are resampled. In each bootstrap extraction, the data from the first element of the sample can be extracted more than once. Each one that's provided has a probability equal to 1 / n of being extracted.

Let E be the estimator of θ that interests us to study, say, E(x) = θ. This quantity is calculated for each bootstrap sample, E(x * 1),…, E(x * m). In this way, m estimates of θ are available, from which it is possible to calculate the bootstrap mean, the bootstrap variance, the bootstrap percentiles, and so on. These values are approximations of the corresponding unknown values and carry information on the distribution of E(x). Therefore, starting from these estimated quantities, it is possible to calculate confidence intervals, test hypotheses, and so on.

Bootstrap resampling using Python

We proceed in a similar way to what we did for Jackknife resampling. We will generate a random distribution, carry out a resampling according to the bootstrap method, and then compare the results. Let's see the code step by step in order to understand the procedure:

1. Let's start by importing the necessary libraries:

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

   NumPy is a Python library that contains numerous functions that help us manage multidimensional matrices. Furthermore, it contains a large collection of high-level mathematical functions that we can use on these matrices.

2. Now, we will generate a distribution that represents our data population. We will use this data to start extracting samples using the sampling methods we have studied. To do this, we will create an empty list that will contain such data:

   ```
   PopData = list()
   ```

   This list will be populated through generating random numbers.

3. To make the experiment reproducible, we will fix the seed in advance:

   ```
   random.seed(7)
   ```

   The `random.seed ()` function is useful if we want to have the same set of data available to be processed in different ways as it makes the simulation reproducible.

4. Now, we can populate the list with 1,000 randomly generated values:

   ```
   for i in range(1000):
       DataElem = 50 * random.random()
       PopData.append(DataElem)
   ```

   In the previous piece of code, we generated 1,000 random numbers between 0 and 1 using the `random()` function. Then, for each step of the `for` loop, this number was multiplied by 50 to obtain a distribution of numbers between 0 and 50.

5. At this point, we can start extracting a sample of the initial population. The first sample can be extracted using the `random.choices()` function, as follows:

```
PopSample = random.choices(PopData, k=100)
```

   This function extracts a sample of size *k* elements chosen from the population with substitution. We extracted a sample of 100 elements from the original population of 1,000 elements.

6. Now, we can apply the bootstrap method, as follows:

```
PopSampleMean = list()
for i in range(10000):
    SampleI = random.choices(PopData, k=100)
    PopSampleMean.append(np.mean(SampleI))
```

   In this piece of code, we created a new list that will contain the sample. Here, we used a `for` loop with 1,000 steps. At each step, a sample of 100 elements was extracted using the `random.choices ()` function from the initial population. Then, we obtained the average of this sample. This value was then added to the end of the list.

Important Note

We resampled the data with the replacement, thereby keeping the resampling size equal to the size of the original dataset.

7. We now print a histogram of the sample we obtained to visualize its distribution:

```
plt.hist(PopSampleMean)
plt.show()
```

The following graph is printed:

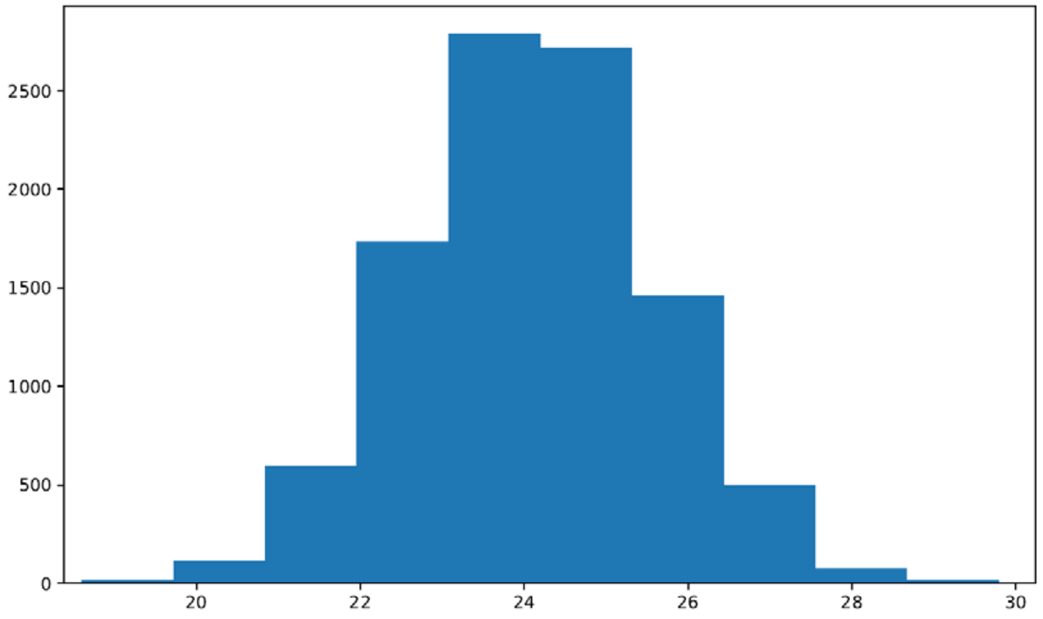


Figure 6.3 – Histogram of the sample distribution

Here, we can see that the sample has a normal distribution.

8. We can now calculate the mean of the three distributions that we have generated. Let's start with the bootstrap estimator:

```
MeanPopSampleMean = np.mean(PopSampleMean)
print("The mean of the Bootstrap estimator is
",MeanPopSampleMean)
```

The following result is returned:

```
The mean of the Bootstrap estimator
is  24.105354873028915
```

We can then calculate the mean of the initial population:

```
MeanPopData = np.mean(PopData)
print("The mean of the population is ",MeanPopData)
```

The following result is returned:

```
The mean of the population is  24.087053989747968
```

Finally, we calculate the mean of the simple sample that was extracted from the initial population:

```
MeanPopSample = np.mean(PopSample)
print("The mean of the simple random sample is
",MeanPopSample)
```

The following result is returned:

```
The mean of the simple random sample
is  23.140472976536497
```

We can now compare the results. Here, the population and bootstrap sample means are practically identical, while the generic sample mean deviates from these values. This tells us that the bootstrap sample is more representative of the initial population than a generic sample that was extracted from it.

Comparing Jackknife and bootstrap

In this section, we will compare the two sampling methods that we have studied by highlighting their strengths and weaknesses:

- Bootstrap requires approximately 10 times more computational effort. Jackknife can, at least theoretically, be done by hand.
- Bootstrap is conceptually simpler than Jackknife. Jackknife requires *n* repetitions for a sample of n, while bootstrap requires a certain number of repetitions. This leads to choosing a number to use, which is not always an easy task. A general rule of thumb is that this number is 1,000 unless you have access to a great deal of computing power.
- Bootstrap introduces errors due to having additional sources of variation due to the finished resampling. Note that this error is reduced for large sizes or where only specific bootstrap sample sets are used.
- Jackknife is more conservative than bootstrap as it produces slightly larger estimated standard errors.
- Jackknife always provides the same results, due to the small differences between the replicas. Bootstrap, on the other hand, provides different results each time it is run.
- Jackknife tends to work best for estimating the confidence interval for pair agreement measures.
- Bootstrap performs better for distorted distributions.
- Jackknife is best suited for small samples of original data.

Explaining permutation tests

When observing a phenomenon belonging to a set of possible results, we ask ourselves what the law of probability is that we can assign to this set. Statistical tests provide a rule that allows us to decide whether to reject a hypothesis based on the sample observations.

Parametric approaches are very uncertain about the experiment plan and the population model. When these assumptions are not respected, particularly when the data law does not conform to the needs of the test, the parametric results are less reliable. When the hypothesis is not based on knowledge of the data distribution and assumptions have not been verified, nonparametric tests are used. Nonparametric tests offer a very important alternative since they need fewer hypotheses.

Permutation tests are a special case of randomization tests that use series of random numbers formulated from statistical inferences. The computing power of modern computers has made their widespread application possible. These methods do not require that their assumptions about data distribution are met.

A permutation test is performed through the following steps:

1. A statistic is defined whose value is proportional to the intensity of the process or relationship being studied.
2. A null hypothesis H_0 is defined.
3. A dataset is created based on the scrambling of those actually observed. The mixing mode is defined according to the null hypothesis.
4. The reference statistics are recalculated, and the value is compared with the one that was observed.
5. The last two steps are repeated many times.
6. If the observed statistic is greater than the limit obtained in 95% of the cases based on shuffling, H_0 is rejected.

Two experiments use values in the same sample space under the respective distributions P_1 and P_2, both of which are members of an unknown population distribution. Given the same dataset x, if the inference conditional on x, which is obtained using the same test statistic, is the same, assuming that the exchangeability for each group is satisfied in the null hypothesis. The importance of permutation tests lies in their robustness and flexibility. The idea of using these methods is to generate a reference distribution from the data and recalculate the test statistics for each permutation of the data with reference to the resulting discrete law.

Approaching cross-validation techniques

Cross-validation is a method used in model selection procedures based on the principle of predictive accuracy. A sample is divided into two subsets, of which the first (training set) is used for construction and estimation, while the second (validation set) is used to verify the accuracy of the predictions of the estimated model. Through a synthesis of repeated predictions, a measure of the accuracy of the model is obtained. A cross-validation method is like jackknife, in that it leaves one observation out at a time. In another method, known as K-fold validation, the sample is divided into K subsets and, in turn, each of them is left out as a validation set.

Important Note

Cross validation can be used to estimate the **Mean Squared Error** (**MSE**) test (or, in general, any measure of precision) of a statistical learning technique in order to evaluate its performance or select its level of flexibility.

Cross validation can be used for both regression and classification problems. The three main validation techniques of a simulation model are the validation set approach, **Leave-One-Out Cross Validation** (**LOOCV**), and k-fold cross validation. In the following sections, we will learn about these concepts in more detail.

The validation set approach

This technique consists of randomly dividing the available dataset into two parts:

- A training set
- A validation set, called the hold-out set

A statistical learning model is adapted to the training data and subsequently used for predicting with the data of the validation set.

The measurement of the resulting test error, which is typically the MSE in the case of regression, provides an estimate of the real test error. In fact, the validation set is the result of a sampling procedure and therefore different samplings result in different estimates of the test error.

This validation technique has various pros and cons. Let's take a look at a few:

- The method tends to have high variability; that is, the results can change substantially as the selected test set changes.
- Only a part of the available units is used for function estimates. This can lead to less precision in function estimating and over-estimating the test error.

The **LOOCV** and k-fold cross validation techniques try to overcome these problems.

Leave-one-out cross validation

LOOCV also divides the observation set into two parts. However, instead of creating two subsets of a comparable size, we do the following:

1. A single observation (x_1, y_1) is used for validation and the remaining observations make up the training set.
2. The function is estimated based on the *n-1* observations of the training set.
3. The prediction $\hat{y}_1$ is made using x_1. Since (x_1, y_1) was not used in the function estimate, an estimate of the test error is as follows:

   $$MSE_1 = (y_1 - \hat{y}_1)^2$$

   But even if MSE_1 is impartial to the test error, it is a poor estimate because it is very variable. This is because it is based on a single observation (x_1, y_1).
4. The procedure is repeated by selecting for validation (x_2, y_2), where a new estimate of the function is made based on the remaining *n-1* observations, and calculating the test error again, as follows:

   $$MSE_2 = (y_2 - \hat{y}_2)^2$$
5. Repeating this approach n times produces *n* test errors.
6. The LOOCV estimate for the MSE test is the average of the *n* MSEs available, as follows:

   $$CV_n = \frac{1}{n}\sum_{i=1}^{n} MSE_i$$

LOOCV has some advantages over the validation set approach:

- Using an *n-1* unit to estimate the function has less bias. Consequently, the LOOCV approach does not tend to overestimate the test error.
- As there is no randomness in the choice of the test set, there is no variability in the results for the same initial dataset.

LOOCV can be computationally intensive, so for large datasets, it takes a long time to calculate. In the case of linear regression, however, there are direct computational formulas with low computational intensity.

K-fold cross validation

In **k-fold cross-validation** (**k-fold CV**), the set of observations is randomly divided into *k* groups, or folders, of approximately equal size. The first folder is considered as a validation set and the function is estimated on the remaining k-1 folders. The mean square error MSE_1 is then calculated on the observations of the folder that's kept out. This procedure is repeated k times, each time choosing a different folder for validation, thus obtaining *k* estimates of the test error. The k-fold CV estimate is calculated by averaging these values, as follows:

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

This method has the advantage of being less computationally intensive if k << n. Furthermore, the k-fold CV tends to have less variability than the LOOCV on different size datasets *n*.

Choosing *k* is crucial in k-fold cross-validation. What happens when *k* changes in cross validation? Let's see what an extreme choice of *k* entails:

- A high *k* value results in larger training sets and therefore less bias. This implies small validation sets, and therefore greater variance.
- A low *k* value results in smaller training sets and therefore greater bias. This implies larger validation sets, and therefore low variance.

Cross-validation using Python

In this section, we will look at an example of the application of cross-validation. First, we will create an example dataset that contains simple data to identify in order to verify the procedure being performed by the algorithm. Then, we will apply k-fold cross-validation and analyze the results:

1. As always, we start by importing the necessary libraries:

```
import numpy as np
from sklearn.model_selection import KFold
```

`numpy` is a Python library that contains numerous functions that help us manage multidimensional matrices: Furthermore, it contains a large collection of high-level mathematical functions we can use on these matrices.

Scikit-learn is an open source Python library that provides multiple tools for machine learning. In particular, it contains numerous classification, regression, and clustering algorithms; support vector machines; logistic regression; and much more. Since it was released in 2007, Scikit-learn has become one of the most used libraries in the field of machine learning, both supervised and unsupervised, thanks to the wide range of tools it offers, but also thanks to its API, which is documented, easy to use, and versatile.

Important Note

Application programming interfaces (**APIs**) are sets of definitions and protocols that application software is created and integrated with. They allow products or services to communicate with other products or services without knowing how they are implemented, thus simplifying app development and allowing a net saving of time and money. When creating new tools and products or managing existing ones, APIs offer flexibility; simplify design, administration, and use; and provide opportunities for innovation.

The scikit-learn API combines a functional user interface with an optimized implementation of numerous classification and meta-classification algorithms. It also provides a wide variety of data preprocessing, cross-validation, optimization, and model evaluation functions. Scikit-learn is particularly popular for academic research since developers can use the tool to experiment with different algorithms by changing only a few lines of code.

2. Now, let's generate the starting dataset:

```
StartedData=np.arange(10,110,10)
print(StartedData)
```

Here, we generated a vector containing 10 integers, starting from the value 10 up to 100 with a step equal to 10. To do this, we used the NumPy `arange()` function. This function generates equidistant values within a certain range. Three arguments have been passed, as follows:

`10`: Start of the interval. This value is included. If this value is omitted, the default value of 0 is used.

`110`: End of range. This value is not included in the range except in cases of floating-point numbers.

`10`: Spacing between values. This is the distance between two adjacent values. By default, this value is equal to 1.

The following array was returned:

```
[ 10  20  30  40  50  60  70  80  90 100]
```

3. Now, we can set the function that will allow us to perform k-fold cross validation:

```
kfold = KFold(5, True, 1)
```

Sklearn's `KFold()` function performs k-fold cross validation by dividing the dataset into k consecutive folds without shuffling by default. Each fold is then used once as validation, while the remaining k - 1 folds form the training set. Three arguments were passed, as follows:

`5`: Number of folds required. This number must be at least 2.

`True`: Optional Boolean value. If it is equal to `True`, the data is mixed before it's divided into batches.

`1`: Seed used by the random number generator.

4. Finally, we can resample the data by using k-fold cross validation:

```
for TrainData, TestData in kfold.split(StartedData):
     print("Train Data :", StartedData[TrainData],"Test
Data :", StartedData[TestData])
```

To do this, we used a loop for the elements generated by the `kfold.split()` method, which returns the indexes that the dataset is divided into. Then, for each step,
which is equal to the number of folds, the elements of the subsets that were drawn are printed.

The following results are returned:

```
Train Data : [ 10  20  40  50  60  70  80  90]
Test Data : [ 30 100]
Train Data : [ 10  20  30  40  60  80  90 100]
Test Data : [50 70]
Train Data : [ 20  30  50  60  70  80  90 100]
Test Data : [10 40]
Train Data : [ 10  30  40  50  60  70  90 100]
```

```
  Test Data : [20 80]
  Train Data : [ 10  20  30  40  50  70  80 100]
  Test Data : [60 90]
```

These pairs of data (`Train Data`, `Test Data`) will be used in succession to train the model and to validate it. This way, you can avoid overfitting and bias problems. Every time you evaluate the model, the extracted part of the dataset is used and the remaining part of the dataset is used for training.

Summary

In this chapter, we learned how to resample a dataset. We analyzed several techniques that approach the problem differently. First, we analyzed the basic concepts of sampling and learned about the reasons that push us to use a sample extracted from a population. We then examined the pros and cons of this choice. We also analyzed how a resampling algorithm works.

We then tackled the first resampling method: the Jackknife method. We first defined the concepts behind the method and then moved on to the procedure, which allows us to obtain samples from the original population. To put the concepts we learned into practice, we applied Jackknife resampling to a practical case.

We then explored the bootstrap method, which builds unobserved but statistically, like the observed samples. This is accomplished by resampling the observed series through an extraction procedure where we reinsert the observations. After defining the method, we worked through an example to highlight the characteristics of the procedure. Furthermore, a comparison between Jackknife and bootstrap was made.

After analyzing the concepts underlying permutation tests, we concluded this chapter by looking at various cross-validation methods. Our knowledge of the k-fold cross-validation method was deepened through an example.

In the next chapter, we will learn about the basic concepts of various optimization techniques and how to implement them. We will understand the difference between numerical and stochastic optimization techniques, and we will learn how to implement stochastic gradient descent. We will then discover how to estimate missing or latent variables and optimize model parameters. Finally, we will discover how to use optimization methods in real-life applications.

7
Using Simulation to Improve and Optimize Systems

Simulation models allow us to obtain a lot of information using few resources. As often happens in life, simulation models are also subject to improvements in order to increase their performance. Through optimization techniques, we try to modify the performance of a model to obtain improvements both in terms of the results and when trying to exploit resources. Optimization problems are usually so complex that a solution cannot be determined analytically. Complexity is determined first by the number of variables and constraints, which define the size of the problem, and then by the possible presence of non-linear functions. To solve an optimization problem, it is necessary to use an iterative algorithm that, given a current approximation of the solution, determines, with an appropriate sequence of operations, updates to this approximation. Starting from an initial approximation, a sequence of approximations that progressively improve the solution is determined.

In this chapter, we will learn how to use the main optimization techniques to improve the performance of our simulation models. We will learn how to use the gradient descent technique, the Newton-Raphson method, and stochastic gradient descent. We will also learn how to apply these techniques with practical examples.

In this chapter, we're going to cover the following main topics:

- Introducing numerical optimization techniques
- Exploring the gradient descent technique
- Facing the Newton-Raphson method
- Deepening our knowledge of stochastic gradient descent
- Discovering multivariate optimization applications in Python

Technical requirements

In this chapter, we will learn how to use simulation models to improve and optimize systems. To deal with the topics in this chapter, it is necessary that you have a basic knowledge of algebra and mathematical modeling. To work with the Python code in this chapter, you'll need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`:

- `gradient_descent.py`
- `newton_raphson.py`
- `scipy_optimize.py`

Introducing numerical optimization techniques

In real life, optimizing means choosing the best option among several available alternatives. Each of us optimizes an itinerary to reach a destination, organizes our day, how we use savings, and so on. In mathematics, optimizing means determining the value of the variables of a function so that it assumes its minimum or maximum. Optimization is the discipline that deals with the formulation of useful models in applications, thereby using efficient methods to identify an optimal solution.

Optimization models have great practical interest for the many applications offered. In fact, there are numerous decision-making processes that require you to determine the choice that minimizes the cost or maximizes the gain and are therefore attributable to optimization models. In optimization theory, a relevant position is occupied by mathematical optimization models, for which the evaluation function and the constraints that characterize the permissible alternatives are expressed through equations and inequalities. Mathematical optimization models come in different forms:

- Linear optimization
- Integer optimization
- Nonlinear optimization

Defining an optimization problem

An optimization problem consists of trying to determine the points that belong to a set F in which a function f takes values that are as low as possible. This problem is represented in the following form:

$$min\, f(x) \quad \forall\, x \in F$$

Here, we have the following:

- f is called the objective function
- F is called the feasible set and contains all the admissible choices for x

Important Note

If you have a maximization problem, that is, if you have to find a point where the function f takes on the highest possible value, you can always go back to the minimal problem, thus changing the sign of the objective function.

The elements that minimize the function f by satisfying the previous relationship are called global optimal solutions, also known as **optimal solutions** or **minimum solutions**. The corresponding value of the objective function f is called the **global optimum value**, also known as the **optimal** or **minimum**.

The complexity of the optimization problem, that is, its difficulty of resolution, obviously depends on the characteristics of the objective function and the structure of the flexible set. Usually, an optimization problem is characterized by whether there is complete freedom in the choice of the vector *x*. We can therefore state that there are two types of problems, as follows:

- Unconstrained minimization problem, if $F = R_n$; that is, if the flexible set *F* coincides with the whole set R_n
- Constrained minimization problem, if $F \subset R_n$; that is, if the flexible set *F* is only a part of the set R_n

While solving an optimization problem, the first difficulty we face is understanding whether the value is well placed, in the sense that there may not be a point F where the function f(x) takes the value of p_i with the least decimal value. In fact, one of the following conditions could occur:

- The flexible set *F* may be empty.
- The flexible set *F* may not be empty but the objective function could have a lower limit equal to $-\infty$.
- The flexible set *F* may not be empty and the objective function could have a lower limit equal to $-\infty$ but, also in this case, there could be no global minimum points of *f* on *F*.

A sufficient but not necessary condition for the existence of a global minimum point in an optimization problem is that expressed by the Weierstrass theorem through the following proposition: let $F \subset R_n$ be a non-empty and compact set. Let *f* be a continuous function on *F*. If so, a global minimum point of *f* exists in *F*.

The previous proposition applies only to the class of constrained problems in which the flexible set is compact. To establish existence results for problems with non-compact flexible sets, that is, in the case where $F = R_n$, it is necessary to try to characterize some subset of *F* containing the optimal solutions to the problem.

Important Note

A compact space is a topological space where every open covering of it contains a finished sub-covering.

In general, there isn't always an optimal solution for the problem at hand and, where it exists, it isn't always unique.

Explaining local optimality

Unfortunately, all global optimality conditions have limited application interest. In fact, they are linked to the overall behavior of the objective function on the admissible set and, therefore, are necessarily described by complex conditions from a computational point of view. Next to the notion of global optimality, as introduced by defining the optimization model, it is appropriate to define the concept of local optimality.

We can define a local optimum as the best solution to a problem in a small neighborhood of possible solutions. In the following graph, we can identify four local minimum conditions for the function *f(x)*, which therefore represent the local optimum:

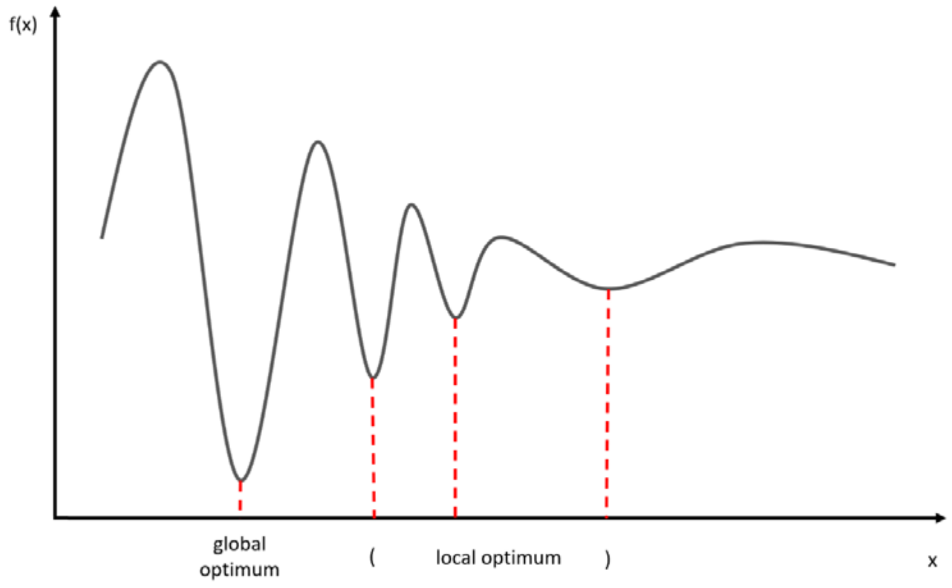


Figure 7.1 – Local minimum conditions for the f(x) function

However, only one of these is a global optimum, while the other three remain local optima. Local optimality conditions are more useful from an application point of view. These are nothing but necessary conditions, but in general, they are not sufficient. This is because an assigned point is a local minimum point of a minimization problem. Therefore, from a theoretical point of view, they do not give a satisfactory characterization of the local minima of the optimization problem, but they do play an important role in the definition of minimization algorithms.

Many problems that are faced in real life can be represented as nonlinear optimization problems. This motivates the increasing interest, from a technical and scientific point of view, in the study and development of methods that can solve this class of difficult mathematical problem.

Exploring the gradient descent technique

The goal of any simulation algorithm is to reduce the difference between the values predicted by the model and the actual values returned by the data. This is because a lower error between the actual and expected values indicates that the algorithm has done a good simulation job. Reducing this difference simply means minimizing the objective function that the model being built is based on.

Defining the descent methods

Descent methods are iterative methods that, starting from an initial point $x_0 \in R^n$, generate a sequence of points $\{x_n\}$ $n \in N$ defined by the following equation:

$$x_{n+1} = x_n + \gamma_n * g_n$$

Here, the g_n vector is a search direction and the γ_n scalar is a positive parameter called step length, which indicates the distance by which we move in the g_n direction.

In a descent method, the g_n vector and the γ_n parameter are chosen to guarantee the decrease of the objective *f* function at each iteration, as follows:

$$f_{x_{n+1}} < f_{x_n} \; \forall \, n \geq 0$$

Using the g_n vector, we take a direction of descent, which is such that the line $x = x_n + \gamma_n * g_n$ forms an obtuse angle with the gradient vector $\nabla f\,(x_n)$. In this way, it is possible to guarantee the decrease of *f*, provided that γ_n is sufficiently small.

Depending on the choice of g_n there are different descent methods. The most common are as follows:

- Gradient descent method
- Newton-Raphson method

Let's start by analyzing the gradient descent algorithm.

Approaching the gradient descent algorithm

A gradient is a vector-valued function that represents the slope of the tangent of the function graph, indicating the direction of the maximum rate of increase of the function. Let's consider the convex function represented in the following diagram:

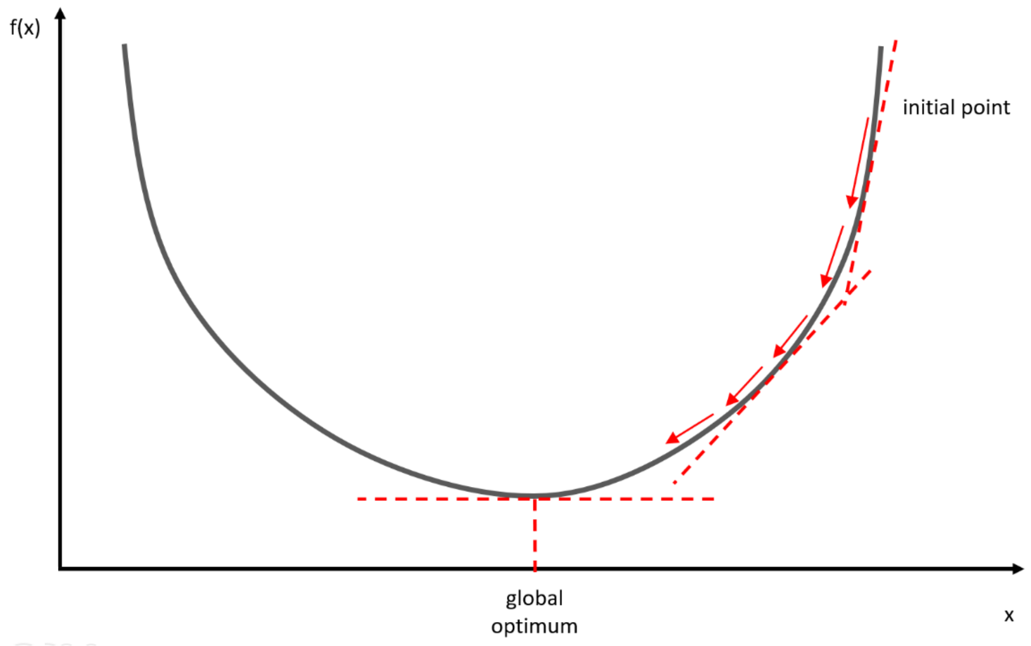


Figure 7.2 – The convex function

The goal of the gradient descent algorithm is to reach the lowest point of this function. In more technical terms, the gradient represents a derivative that indicates the slope or inclination of the objective function.

To understand this better, let's assume we got lost in the mountains at night with poor visibility. We can only feel the slope of the ground under our feet. Our goal is to reach the lowest point of the mountain. To do this, we will have to walk a few steps and move toward the direction of the highest slope. We will do this iteratively, moving one step at a time until we finally reach the mountain valley.

In mathematics, the derivative is the rate of change or slope of a function at a given point. So, the value of the derivative is the incline of the slope at a specific point. The gradient represents the same thing, with the addition that it is a vector value function that stores partial derivatives. This means that the gradient is a vector and that each of its components is a partial derivative with respect to a specific variable.

Let's analyze a function, $f(x, y)$, that is, a two-variable function, x and y. Its gradient is a vector containing the partial derivatives of f: the first with respect to x and the second with respect to y. If we calculate the partial derivatives of f, we get the following:

$$\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}$$

The first of these two expressions is called a partial derivative with respect to x, while the second partial derivative is with respect to y. The gradient is the following vector:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \end{bmatrix}$$

The preceding equation is a function that represents a point in a two-dimensional space, or a two-dimensional vector. Each component indicates the steepest climbing direction for each of the function variables. Hence, the gradient points in the direction that the function increases the most in.

Similarly, if we have a function with five variables, we would get a gradient vector with five partial derivatives. Generally, a function with n variables results in an n-dimensional gradient vector, as follows:

$$\nabla f(x,y,\ldots.z) = \begin{bmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \\ \cdots \\ \cdots \\ \frac{\delta f}{\delta z} \end{bmatrix}$$

For gradient descent, however, we don't want to maximize *f* as fast as possible. Instead, we want to minimize it, that is, find the smallest point that minimizes the function.

Suppose we have a function *y = f(x)*. Gradient descent is based on the observation that if the function f is defined and differentiable in a neighborhood of *x*, then this function decreases faster if we move in the direction of the negative gradient. Starting from a value of *x*, we can write the following:

$$x_{n+1} = x_n - \gamma * \nabla f(x_n)$$

Here, we have the following:

- γ is the learning rate
- ∇ is the gradient

For sufficiently small γ values, the algorithm converges to the minimum value of the function *f* in a finite number of iterations.

Important Note

Basically, if the gradient is negative, the objective function at that point is decreasing, which means that the parameter must move toward larger values to reach a minimum point. On the contrary, if the gradient is positive, the parameters move toward smaller values to reach the lower values of the objective function.

Understanding the learning rate

The gradient descent algorithm searches for the minimum of the objective function through an iterative process. At each step, an estimate of the gradient is performed to direct the descent in the direction that minimizes the objective function. In this procedure, the choice of the **learning rate** parameter becomes crucial. This parameter determines how quickly or slowly we will move to the optimal values of the objective function:

- If it is too small, we will need too many iterations to converge to the best values.
- If the learning rate is very high, we will skip the optimal solution.

In the following diagram, you can see the two possible scenarios imposed by the value of the learning rate:

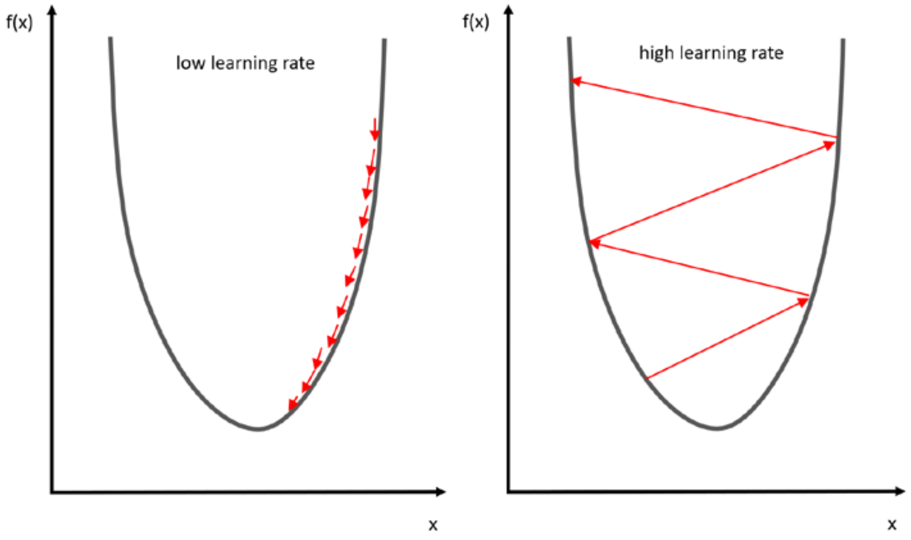


Figure 7.3 – The scenarios for the learning rate

Due to this, it is essential to use a good learning rate. The best way to identify the optimal learning rate is through trial and error.

Explaining the trial and error method

The term **trial and error** defines a heuristic method that aims to find a solution to a problem by attempting it and checking if it has produced the desired effect. If so, the attempt constitutes a solution to the problem; otherwise, we continue with a different attempt.

Let's analyze the essential characteristics of the method:

- **It is oriented toward the solution**: It does not aim to find out why an attempt works, but simply seeks it.
- **It is specific to the problem in question**: It has no claim to generalize to other problems.
- **It is not optimal**: It usually limits itself to finding a single solution that will usually not be the best possible one.
- **It does not require having thorough knowledge of it**: It proposes to find a solution to a problem of which little or nothing is known about it.

The trial and error method can be used to find all the solutions to the problem or the best solution among them if there is more than one. In this case, instead of stopping at the first attempt that provided a desired result, we take note of it and continue in the attempts until all the solutions are found. At the end, these are compared based on a given criterion, which will determine which of them is to be considered the best.

Implementing gradient descent in Python

In this section, we will apply what we have learned so far on the gradient descent by completing a practical example. We will define a function and then use that method to find the minimum point of the function. As always, we will analyze the code line by line:

1. Let's start by importing the necessary libraries:

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

   `numpy` is a Python library that contains numerous functions that help us manage multidimensional matrices. Furthermore, it contains a large collection of high-level mathematical functions we can use on these matrices.

`matplotlib` is a Python library for printing high-quality graphics. With matplotlib, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and so on with a few commands. It is a collection of command-line functions like those provided by the Matlab software.

2. Now, let's define the function:

```
x = np.linspace(-1,3,100)
y=x**2-2*x+1
```

First, we defined an interval for the dependent variable `x`. We will only need this to visualize the function and draw a graph. To do this, we used the `linspace()` function. This function creates numerical sequences. Then, we passed three arguments: the starting point, the ending point, and the number of points to be generated. Next, we defined a parabolic function.

3. Now, we can draw a graph and display it:

```
fig = plt.figure()
axdef = fig.add_subplot(1, 1, 1)
axdef.spines['left'].set_position('center')
axdef.spines['bottom'].set_position('zero')
axdef.spines['right'].set_color('none')
axdef.spines['top'].set_color('none')
axdef.xaxis.set_ticks_position('bottom')
axdef.yaxis.set_ticks_position('left')
plt.plot(x,y, 'r')
plt.show()
```

First, we defined a new figure and then set the axes so that the x-axis coincides with the minimum value of the function and the y-axis coincides with the center of the parabola. This will make it easier to visually locate the minimum point of the function. The following diagram is printed:

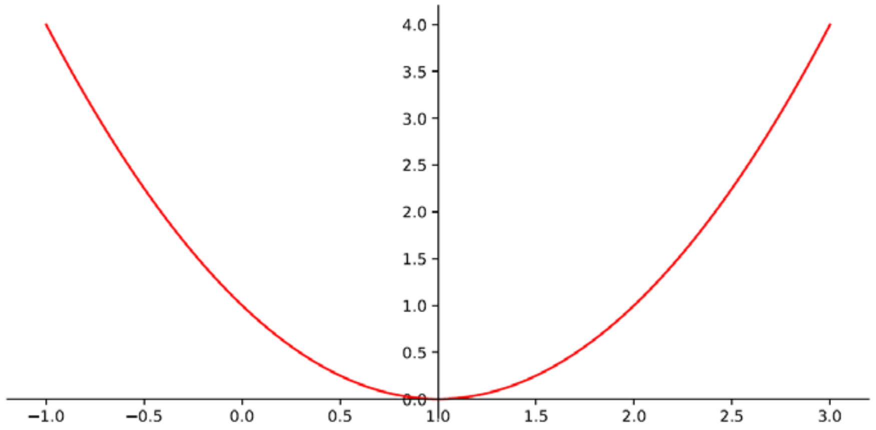


Figure 7.4 – The minimum point of the function

As we can see, the minimum of the function corresponding to `y = 0` occurs for a value of `x` equal to `1`. This will be the value that we will have to determine through the gradient descent method.

4. Now, let's define the gradient function:

```
Gradf = lambda x: 2*x-2
```

Recall that the gradient of a function is its derivative. In this case, doing this is easy since it is a single-variable function.

5. Before applying the iterative procedure, it is necessary to initialize a series of variables:

```
actual_X  = 3
learning_rate  = 0.01
precision_value = 0.000001
previous_step_size = 1
max_iteration = 10000
iteration_counter = 0
```

As seen here:

- The `actual_X` variable will contain the current value of the independent variable `x`. To start, we initialize it at `x = 3`, which represents the far-right value of the display range of the function in the graph.
- The `learning_rate` variable contains the learning rate. As explained in the *Understanding the learning rate* section, it is set at `0.01`. We can try to see what happens if we change that value.
- The `precision_value` variable will contain the value that defines the degree of precision of our algorithm. Being an iterative procedure, the solution is refined at each iteration and tends to converge. But this convergence may come after a very large number of iterations, so to save resources, it is advisable to stop the iterative procedure once adequate precision has been reached.
- The `previous_step_size` variable will contain the calculation of this precision and will be initialized to `1`.
- The `max_iteration` variable contains the maximum number of iterations that we have provided for our algorithm. This value will be used to stop the procedure if it does not converge.
- Finally, the `iteration_counter` variable will be the iteration counter.

6. Now, we are ready for the iteration procedure:

```
while previous_step_size  > precision_value  and
iteration_counter  < max_iteration :
    PreviousX = actual_X
    actual_X  = actual_X  - learning_rate  *
Gradf(PreviousX)
    previous_step_size  = abs(actual_X  - PreviousX)
    iteration_counter  = iteration_counter +1
    print('Number of iterations = ',iteration_counter ,'\
nActual value of x  is = ',actual_X )
    print('X value of f(x) minimum = ', actual_X )
```

The iterative procedure uses a `while` loop, which will repeat itself until both conditions are verified (`TRUE`). When at least one of the two assumes a `FALSE` value, the cycle will be stopped. The two conditions provide a check on the precision and the number of iterations.

This procedure, as anticipated in the *Defining the gradient descent* section, requires that we update the current value of *x* in the direction of the gradient's descent. We do this using the following equation:

$$x_{n+1} = x_n - \gamma * \nabla f(x_n)$$

At each step of the cycle, the previous value of `x` is stored so that we can calculate the precision of the previous step as the absolute value of the difference between the two `x` values. In addition, the step counter is increased at each step. At the end of each step, the number of iterations and the current value of the `x` are printed, as follows:

```
Number of iterations =  520
Actual value of x  is =  1.0000547758790321
Number of iterations =  521
Actual value of x  is =  1.0000536803614515
Number of iterations =  522
Actual value of x  is =  1.0000526067542224
Number of iterations =  523
Actual value of x  is =  1.000051554619138
Number of iterations =  524
Actual value of x  is =  1.0000505235267552
Number of iterations =  525
Actual value of x  is =  1.0000495130562201
Number of iterations =  526
Actual value of x  is =  1.0000485227950957
```

As we can see at each step, the value of `x` progressively approaches the exact value. Here, `526` iterations were executed.

7. At the end of the procedure, we can print the result:

```
print('X value of f(x) minimum = ', actual_X )
```

The following result is returned:

```
X value of f(x) minimum =  1.0000485227950957
```

As we can verify, the returned value is very close to the exact value, which is equal to `1`. It differs precisely from the value of the precision that we imposed as the term for the iterative procedure.

Facing the Newton-Raphson method

Newton's method is the main numerical method for the approximation of roots of nonlinear equations. The function is linearly approximated at each iteration to obtain a better estimate of the zero point.

Using the Newton-Raphson algorithm for root-finding

Given a nonlinear function f and an initial approximation x_0, Newton's method generates a sequence of approximations $\{x_k\}$ *k > 0* by constructing, for each *k*, a linear model of the function f in a neighborhood of x_k and approximating the function with the model itself. This model can be constructed starting from Taylor's development of the function *f* at a point *x* belonging to a neighborhood of the iterated current point x_k, as follows:

$$f(x) = f(x_k) + (x - x_k) * f'(x_k) + (x - x_k)^2 * \frac{f''(x_k)}{2!} + \cdots$$

Truncating Taylor's first-order development gives us the following linear model:

$$f(x) = f(x_k) + (x - x_k) * f'(x_k)$$

The previous equation remains valid in a sufficiently small neighborhood of x_k.

Given x_0 as the initial data, the first iteration consists of calculating x_1 as the zero of the previous linear model with *k = 0*, that is, to solve the following scalar linear equation:

$$f(x) = 0$$

The previous equation leads to the next iterated x_1 in the following form:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Similarly, the subsequent equation iterates x_2, where x_3 is constructed so that we can elaborate on a general validity equation, as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The form of the update equation is like the generic formula of descent methods. From a geometric point of view, the previous update equation represents the line tangent to the function *f* at the point $(x_k, f(x_k))$. It is for this reason that the method is also called the tangent method.

Geometrically, we can define this procedure through the following steps:

- The tangent of the function is plotted at the starting point x_0.
- The intercept of this line is identified with the x-axis. This point represents the new value x_1.
- This procedure is repeated until convergence.

The following diagram shows this procedure:

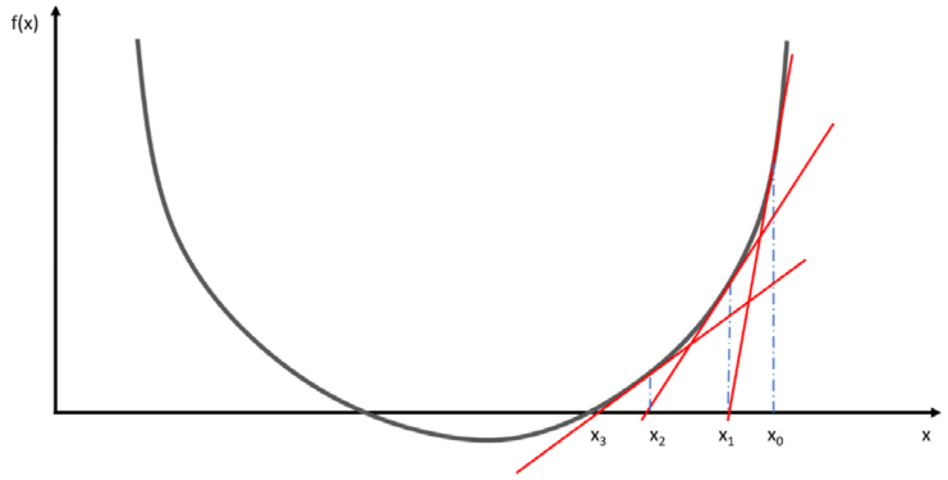


Figure 7.5 – The procedure of finding a tangent

This algorithm is well-defined if $f'(x_k) = 0$ for every *k*. With regards to the computational cost, it can be noted that, at each iteration, the evaluation of the function *f* and its derivative before in point x_k are required.

Approaching Newton-Raphson for numerical optimization

The Newton-Raphson method is also used for solving numerical optimization problems. In this case, the method takes the form of Newton's method for finding the zeros of a function, but applied to the derivative of the function *f*. This is because determining the minimum point of the function *f* is equivalent to determining the root of the first derivative *f'*.

In this case, the update formula takes the following form:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

In the previous equation, we have the following:

- $f'(x_n)$ is the first derivative of the function *f*
- $f''(x_n)$ is the second derivative of the function *f*

Important Note

The Newton-Raphson method is usually preferred over the descending gradient method due to its speed. However, it requires knowledge of the analytical expression of the first and second derivatives and converges indiscriminately to the minima and maxima.

There are variants that bring this method to global convergence and that lower the computational cost by avoiding having to determine the direction of the research with direct methods.

Applying the Newton-Raphson technique

In this section, we will apply what we have learned so far about the Newton-Raphson method by completing a practical exercise. We'll define a function and then use that method to find the minimum point of the function. As always, we will analyze the code line by line:

1. Let's start by importing the necessary libraries:

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

2. Now, let's define the function:

```
x = np.linspace(0,3,100)
y=x**3 -2*x**2 -x + 2
```

   First, we defined an interval for the dependent variable x. We will only need this to visualize the function in order to draw a graph. To do this, we used the `linspace()` function. This function creates numerical sequences. Then, we passed three arguments: the starting point, the ending point, and the number of points to be generated. Next, we defined a cubic function.

3. Now, we can draw a graph and display it:

```
fig = plt.figure()
axdef = fig.add_subplot(1, 1, 1)
axdef.spines['left'].set_position('center')
axdef.spines['bottom'].set_position('zero')
axdef.spines['right'].set_color('none')
axdef.spines['top'].set_color('none')
axdef.xaxis.set_ticks_position('bottom')
axdef.yaxis.set_ticks_position('left')
plt.plot(x,y, 'r')
plt.show()
```

First, we defined a new figure and then we set the axes so that the x-axis coincides with the minimum value of the function and the y-axis coincides with the center of the parabola. This will make it easier to visually locate the minimum point of the function. The following graph is printed:

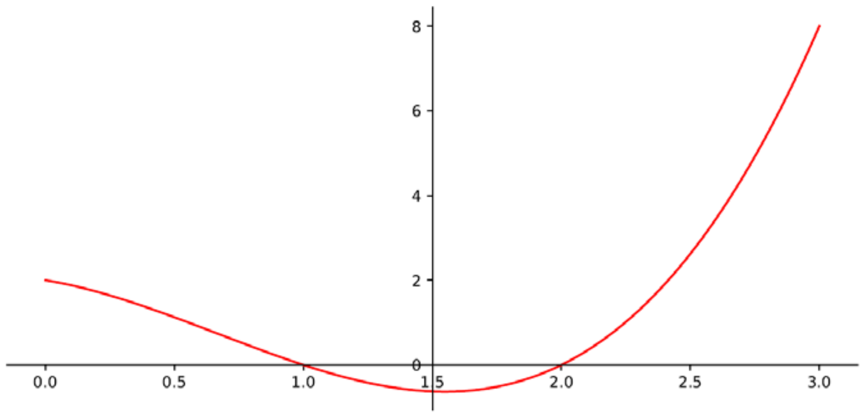


Figure 7.6 – The minimum point of the function

Here, we can see that the minimum of the function occurs for a value of `x` roughly equal to `1.5`. This will be the value that we will have to determine through the gradient descent method. But to have the precise value so that we can compare it with what we will get later, we need to extract this value:

```
print('Value of x at the minimum of the function', x[np.
argmin(y)])
```

To determine this value, we used NumPy's `argmin()` function. This function returns the position index of the minimum element in a vector. The following result is returned:

```
Value of x at the minimum of the function
1.5454545454545454
```

4. Now, let's define the first and second derivative functions:

```
FirstDerivative = lambda x: 3*x**2-4*x -1
SecondDerivative = lambda x: 6*x-4
```

5. Now, we will initialize some parameters:

```
actual_X  = 3
precision_value  = 0.000001
previous_step_size  = 1
max_iteration  = 10000
iteration_counter  = 0
```

These parameters have the following meaning:

- The `actual_X` variable will contain the current value of the independent variable `x`. To start, we initialize it at `x = 3`, which represents the far-right value of the display range of the function in the graph.
- The `precision_value` variable will contain the value that defines the degree of precision of our algorithm. Being an iterative procedure, the solution is refined at each iteration and tends to converge. But this convergence may come after a very large number of iterations, so to save resources, it is advisable to stop the iterative procedure once adequate precision has been reached.
- The `previous_step_size` variable will contain the calculation of this precision and will be initialized to `1`.
- The `max_iteration` variable contains the maximum number of iterations that we have provided for our algorithm. This value will be used to stop the procedure if it does not converge.
- Finally, the `iteration_counter` variable will be the iteration counter.

6. Now, we can apply the Newton-Raphson method, as follows:

```
while previous_step_size  > precision_value  and
iteration_counter  < max_iteration :
    PreviousX = actual_X
    actual_X  = actual_X  - FirstDerivative(PreviousX)/
SecondDerivative(PreviousX)
    previous_step_size  = abs(actual_X  - PreviousX)
    iteration_counter  = iteration_counter +1
    print('Number of iterations = ',iteration_counter ,'\
nActual value of x  is = ',actual_X )
```

This procedure is similar to what we adopted to solve our problem in the *Implementing gradient descent in Python* section. A `while` loop, which will repeat itself until both conditions are verified (`TRUE`), was used here. When at least one of the two assumes a `FALSE` value, the cycle will be stopped. These two conditions provide a check on the precision and the number of iterations.

The Newton-Raphson method updates the current value of x as follows:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

At each step of the cycle, the previous value of `x` is stored in order to calculate the precision of the previous step as the absolute value of the difference between the two `x` values. In addition, the step counter is increased at each step. At the end of each step, the number of the iteration and the current value of `x` are printed, as follows:

```
Number of iterations =  1
Actual value of x  is =  2.0
Number of iterations =  2
Actual value of x  is =  1.625
Number of iterations =  3
Actual value of x  is =  1.5516304347826086
Number of iterations =  4
Actual value of x  is =  1.5485890147300967
Number of iterations =  5
Actual value of x  is =  1.5485837703704566
Number of iterations =  6
Actual value of x  is =  1.5485837703548635
```

As we mentioned in the *Approaching Newton-Raphson for numerical optimization* section, the number of iterations necessary to reach the solution is drastically skewed. In fact, we went from the 526 iterations necessary to bring the method based on the gradient's descent to convergence, to only 6 iterations for the Newton-Raphson method.

7. Finally, we will print the result:

```
print('X value of f(x) minimum = ', actual_X )
```

The following result is returned:

```
X value of f(x) minimum =  1.5485837703548635
```

As we can verify, the returned value is very close to the exact value, which is equal to `1.5454545454545454`. It differs precisely in terms of the value of the precision that we imposed as the term for the iterative procedure.

Deepening our knowledge of stochastic gradient descent

As we mentioned in the *Exploring the gradient descent technique* section, the implementation of the gradient descent method consists of initially evaluating both the function and its gradient, starting from a configuration chosen randomly in the space of dimensions.

From here, we try to move in the direction indicated by the gradient. This establishes a direction of descent in which the function tends to a minimum and examines whether the function actually takes on a value lower than that calculated in the previous configuration. If so, the procedure continues iteratively, recalculating the new gradient. This can be totally different from the previous one. After this, it starts again in search of a new minimum.

This iterative procedure requires that, at each step, the entire system status is updated. This means that all the parameters of the system must be recalculated. From a computational point of view, this equates to an extremely expensive operating cost and greatly slows down the estimation procedure. With respect to the standard gradient descent method, in which the weights are updated after calculating the gradient for the entire dataset, in the stochastic method, the system parameters are updated after a certain number of examples. These are chosen randomly in order to speed up the process and to try and avoid any local minimum situations.

Consider a dataset that contains n observations of a phenomenon. Here, let *f* be an objective function that we want to minimize with respect to a series of parameters *x*. Here, we can write the following equation:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i\,(x)$$

From the analysis of the previous equation, we can deduce that the evaluation of the objective function *f* requires n evaluations of the function *f*, one for each value contained in the dataset.

In the classic gradient descent method, at each step, the function gradient is calculated in correspondence with all the values of the dataset through the following equation:

$$x_{n+1} = x_n - \gamma * \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x_n)$$

In some cases, the evaluation of the sum present in the previous equation can be particularly expensive, such as when the dataset is particularly large and there is no elementary expression for the objective function. The stochastic descent of the gradient solves this problem by introducing an approximation of the gradient function. At each step, instead of the sum of the gradients being evaluated in correspondence to the data contained in the dataset, the evaluation of the gradient is used only in a random subset of the dataset.

So, the previous equation replaces the following:

$$x_{n+1} = x_n - \gamma * \nabla f_i(x_n)$$

In the previous equation, $\nabla f_i(x_n)$ is the gradient of one of the observations in the dataset, chosen randomly.

The pros of this technique are as follows:

- Based only on a part of the observations, the algorithm allows a wider exploration of the parametric space, with the greater possibility of finding new and potentially better points of the minimum.
- Taking a step of the algorithm is computationally much faster, which ensures faster convergence toward the minimum point.
- The parameter estimates can also be calculated by loading only a part of the dataset into memory at a time, allowing this method to be applied to large datasets.

Discovering the multivariate optimization methods in Python

In this section, we will analyze some numerical optimization methods contained in the Python SciPy library. SciPy is a collection of mathematical algorithms and functions based on NumPy. It contains a series of commands and high-level classes that can be used to manipulate and display data. With SciPy, functionality is added to Python, making it a data processing and system prototyping environment, similar to commercial systems such as MATLAB.

Scientific applications that use SciPy benefit from the development of add-on modules in numerous fields of numerical computing made by developers around the world. Numerical optimization problems are also covered among the available modules.

The SciPy `optimize` module contains numerous functions for the minimization/maximization of objective functions, both constrained and unconstrained. It treats nonlinear problems with support for both local and global optimization algorithms. In addition, problems regarding linear programming, constrained and nonlinear least squares, search for roots, and the adaptation of curves are dealt with. In the following sections, we will analyze some of them.

The Nelder–Mead method

Most of the well-known optimization algorithms are based on the concept of derivatives and on the information that can be deduced from the gradient. However, many optimization problems deriving from real applications are characterized by the fact that the analytical expression of the objective function is not known, which makes it impossible to calculate its derivatives, or because is particularly complex, so coding the derivatives may take too long. To solve this type of problem, several algorithms have been developed that do not attempt to approximate the gradient but rather use the values of the function in a set of sampling points to determine a new iteration by other means.

The Nelder-Mead method tries to minimize a nonlinear function by evaluating test points that constitute a geometric form called a simplex.

Important Note

A simplex is defined as a set of closed and convex points of a Euclidean space that allow us to find the solution to the typical optimization problem of linear programming.

The choice of geometric figure for the simplex is mainly due to two reasons: the ability of the simplex to adapt its shape to the trend in the space of the objective function deforming itself, and the fact that it requires the memorization of only *n + 1* points. Each iteration of a direct search method based on the simplex begins with a simplex, specified by its *n + 1* vertices and the values of the associated functions. One or more test points and the respective values of the function are calculated, and the iteration ends with a new simplex so that the values of the function in its vertices satisfy some form of descent condition with respect to the previous simplex.

The Nelder-Mead algorithm is particularly sparing in terms of its evaluation of the function at each iteration, given that, in practice, it typically requires only one or two evaluations of the function to build a new simplex. However, since it does not use any gradient assessment, it may take longer to find the minimum.

This method is easily implemented in Python using the `minimize` routine of the SciPy `optimize` module. Let's look at a simple example of using this method:

1. Let's start by loading the necessary libraries:

   ```
   import numpy as np
   from scipy.optimize import minimize
   import matplotlib.pyplot as plt
   from matplotlib import cm
   from matplotlib.ticker import LinearLocator,
   FormatStrFormatter
   from mpl_toolkits.mplot3d import Axes3D
   ```

   The library that's needed to generate 3D graphics is imported (`Axes3D`).

2. Now, let's define the function:

   ```
   def matyas(x):
       return 0.26*(x[0]**2+x[1]**2)-0.48*x[0]*x[1]
   ```

The Matyas function is continuous, convex, unimodal, differentiable, and non-separable, and is defined on two-dimensional space. The `matyas` function is defined as follows:

$$f(x, y) = 0.26 * (x^2 + y^2) - 0.48 * x * y$$

This function is defined on a x, y E [-10, 10]. This function has one global minimum in f(0, 0) = 0.

3. Let's visualize the `matyas` function:

```
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
x, y = np.meshgrid(x, y)
z = matyas([x,y])

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1,
                      cmap=cm.RdBu,linewidth=0,
antialiased=False)

ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

fig.colorbar(surf, shrink=0.5, aspect=10)

plt.show()
```

To start, we defined the independent variables, x and y, in the range that we have already specified `[-10.10]`. So, we created a grid using the `numpy meshgrid()` function. This function creates an array in which the rows and columns correspond to the values of `x` and `y`. We will use this matrix to plot the corresponding points of the `z` variable, which corresponds to the Matyas function. After defining the `x`, `y`, and `z` variables, we traced a three-dimensional graph to represent the function. The following graph is plotted:

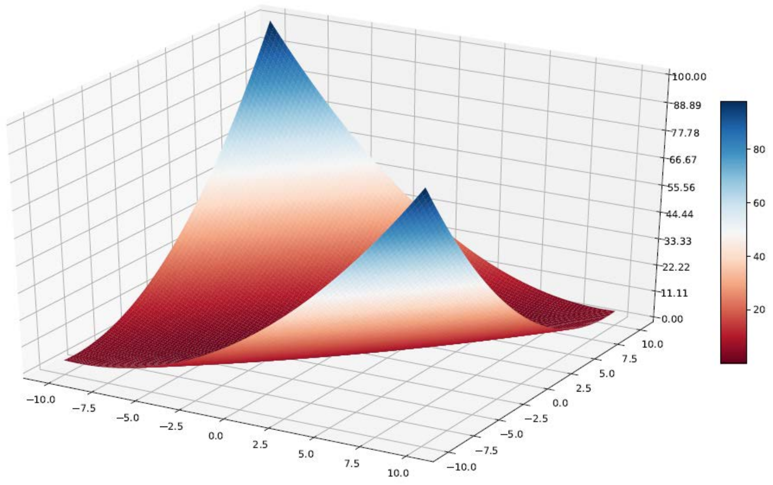


Figure 7.7 – Meshgrid plot to represent the function

4. As we already mentioned, the Nelder-Mead method does not require us to calculate a derivative as it is limited to evaluating the function. This means that we can directly apply the method:

```
x0 = np.array([-10, 10])
NelderMeadOptimizeResults = minimize(matyas, x0,
            method='nelder-mead',
            options={'xatol': 1e-8, 'disp': True})
print(NelderMeadOptimizeResults.x)
```

To do this, we first defined an initial point to start the search procedure from for the minimum of the function. So, we used the `minimize()` function of the SciPy optimize module. This function finds the minimum of the scalar functions of one or more variables. The following parameters have been passed:

- `matyas`: The function you want to minimize
- `x0`: The initial vector

- `method = 'nelder-mead'`: The method used for the minimization procedure

  Additionally, the following two options have been added:

- `'xatol': 1e-8`: Defines the absolute error acceptable for convergence
- `'disp': True`: Set to True to print convergence messages

5. Finally, we printed the results of the optimization method, as follows:

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 77
         Function evaluations: 147
[3.17941614e-09 3.64600127e-09]
```

   The minimum was identified in the value `0`, as already anticipated. Furthermore, this value was identified in correspondence with the following values:

```
X = 3.17941614e-09
Y = 3.64600127e-09
```

   These are values that are very close to zero, as we expected. The deviation from this value is consistent with the error that we set for the method.

Powell's conjugate direction algorithm

Conjugate direction methods were originally introduced as iterative methods for solving linear systems with a symmetric and positive definite coefficient matrix, and for minimizing strictly convex quadratic functions.

The main feature of conjugated direction methods for minimizing quadratic functions is that of generating, in a simple way, a set of directions that, in addition to being linearly independent, enjoy the further important property of being mutually conjugated.

The idea of Powell's method is that if the minimum of a quadratic function is found along each of the p(p < n) directions in a stage of the research, then when taking a step along each direction, the final displacement from the beginning up to the *p*-th step is conjugated with respect to all the *p* subdirections of research.

For example, if points 1 and 2 are obtained from one-dimensional searches in the same direction but from different starting points, then the line formed by 1 and 2 will be directed toward the maximum. The directions represented by these lines are called conjugate directions.

Let's analyze a practical case of applying the Powell method. We will use the `matyas` function, which we defined in the *Nelder-Mead method* section:

1. Let's start by loading the necessary libraries:

```
import numpy as np
from scipy.optimize import minimize
```

2. Now, let's define the function:

```
def matyas (x):
    return 0.26 * (x [0] ** 2 + x [1] ** 2) -0.48 * x [0]
* x [1]
```

3. Now, let's apply the method:

```
x0 = np.array([-10, 10])
PowellOptimizeResults = minimize(matyas, x0,
           method='Powell',
           options={'xtol': 1e-8, 'disp': True})
print(PowellOptimizeResults.x)
```

The `minimize()` function of the SciPy optimize module was used here. This function finds the minimum of the scalar functions of one or more variables. The following parameters were passed:

- `matyas`: The function we want to minimize
- `x0`: The initial vector
- `method = 'Powell'`: The method used for the minimization procedure

Additionally, the following two options have been added:

- `'xtol': 1e-8`: Defines the absolute error acceptable for convergence
- `'disp': True`: Set to *True* to print convergence messages

4. Finally, we printed the results of the optimization method. The following results are returned:

```
Optimization terminated successfully.
        Current function value: 0.000000
        Iterations: 3
        Function evaluations: 66
[-6.66133815e-14 -1.32338585e-13]
```

The minimum was identified in the value 0, as specified in the *Nelder-Mead method* section. Furthermore, this value was identified in correspondence with the following values:

```
X = -6.66133815e-14
Y = -1.32338585e-13
```

These are values very close to zero, as we expected. We can now make a comparison between the two methods we applied to the same function. We can note that the number of iterations necessary to reach convergence is equal to `3` for the Powell method, while it is equal to `77` for the Nelder-Mead method. A drastic reduction in the number of evaluations of the function is also noted; `66` against `147`. Finally, the difference between the calculated value and the expected value is reduced by the Powell method.

Summarizing other optimization methodologies

The `minimize()` routine from the SciPy optimize package contains numerous methods for unconstrained and constrained minimization. We analyzed some of them in detail in the previous sections. In the following list, we have summarized the most used methods provided by the package:

- **Newton-Broyden-Fletcher-Goldfarb-Shanno** (**BFGS**): This is an iterative unconstrained optimization method used to solve nonlinear problems. This method looks for the points where the first derivative is zero.
- **Conjugate Gradient** (**CG**): This method belongs to the family of conjugate gradient algorithms and performs a minimization of the scalar function of one or more variables. This method requires that the system matrix be symmetric and positive definite.
- **Dog-leg trust-region** (**dogleg**): The method first defines a region around the current best solution, where the original objective function can be approximated. The algorithm therefore takes a step forward within the region.
- **Newton-CG**: This method is also called truncated Newton's method. It is a method that identifies the direction of research by adopting a procedure based on the conjugate gradient, to roughly minimize the quadratic function.
- **Limited-memory BFGS** (**L-BFGS**): This is part of the family of quasi-Newton methods. It uses the BFGS method for systematically saving computer memory.

- **Constrained Optimization By Linear Approximation** (**COBYLA**): The operating mechanism is iterative and uses the principles of linear programming to refine the solution found in the previous step. Convergence is achieved by progressively reducing the pace.

Summary

In this chapter, we learned how to use different numerical optimization techniques to improve the solutions offered by a simulation model. We started by introducing the basic concepts of numerical optimization, defining a minimization problem, and learning to distinguish between local and global minimums. We then moved on and looked at the optimization techniques based on gradient descent. We defined the mathematical formulation of the technique and gave it a geometric representation. Furthermore, we deepened our knowledge of the concepts surrounding the learning rate and trial and error. By doing this, we addressed a practical case in order to reinforce the concepts we learned by solving the problem of searching for the minimum of a quadratic function.

Subsequently, we learned how to use the Newton-Raphson method to search for the roots of a function and then how to exploit the same methodology for numerical optimization. We also analyzed a practical case for this technology to immediately put the concepts we learned into practice. We did this by looking for the local minimum of a convex function.

We then went on to study the stochastic gradient descent algorithm, which allows us to considerably reduce the computational costs of a numerical optimization problem. This result is obtained by using a single estimate of the gradient at each step, which is chosen in a stochastic way among those available.

Finally, we explored the multivariate numerical optimization algorithms contained in the Python SciPy package. For some of them, we defined the mathematical formulation and proposed a practical example of using the method. For the others, a summary was drawn up to list their characteristics.

In the next chapter, we will learn how to use simulation models to handle financial problems. We will explore how the geometric Brownian motion model works, and we will discover how to use Monte Carlo methods for stock price prediction. Finally, we will learn how to model credit risks using Markov chains.

Section 3: Real-World Applications

In this section, we will use the techniques that we introduced in the previous chapters to deal with practical cases. By the end of this section, you will be well versed in the real-world applications of simulation modeling.

This section contains the following chapters:

8 Using Simulation Models for Financial Engineering

The explosive entry of systems based on artificial intelligence and machine learning has opened up new scenarios for the financial sector. These methods can bring benefits such as user rights protections, as well as macroeconomic benefits.

Monte Carlo methods find a natural application in finance for the numerical resolution of pricing and problems in covered call options. Essentially, these methods consist of simulating a given process or phenomenon using a given mathematical law and a sufficiently large set of data, created randomly from distributions that adequately represent real variables. The idea is that, if an analytical study is not possible, or adequate experimental sampling is not possible or convenient, the numerical simulation of the phenomenon is used. In this chapter, we will look at practical cases of using simulation methods in a financial context. You will learn how to use Monte Carlo methods to predict stock prices and how to assess the risk associated with a portfolio of sharess.

In this chapter, we're going to cover the following topics:

- Understanding the geometric Brownian motion model
- Using Monte Carlo methods for stock price prediction
- Studying risk models for portfolio management

Technical requirements

In this chapter, we will learn how to use simulation models for financial engineering. In order to understand these topics, basic knowledge of algebra and mathematical modeling is needed.

To work with the Python code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `StandardBrownianMotion.py`
- `AmazonStockMontecarloSimulation.py`
- `ValueAtRisk.py`

Understanding the geometric Brownian motion model

The name **Brownian** comes from the Scottish botanist Robert Brown who, in 1827, observed, under the microscope, how pollen particles suspended in water moved continuously in a random and unpredictable way. In 1905, it was Einstein who gave a molecular interpretation of the phenomenon of movement observed by Brown. He suggested that the motion of the particles was mathematically describable, assuming that the various jumps were due to the random collisions of pollen particles with water molecules.

Today, Brownian motion is, above all, a mathematical tool in the context of probability theory. This mathematical theory has been used to describe an ever-widening set of phenomena, studied by disciplines that are very different from physics. For instance, the prices of financial securities, the spread of heat, animal populations, bacteria, illness, sound, and light are modeled using the same instrument.

Important note

Brownian motion is a phenomenon that consists of the uninterrupted and irregular movement made by small particles or grains of colloidal size, that is, particles that are far too small to be observed with the naked eye but are significantly larger than atoms when immersed in a fluid.

Defining a standard Brownian motion

There are various ways of constructing a Brownian motion model and various equivalent definitions of Brownian motion. Let's start with the definition of a standard Brownian motion (the Wiener process). The essential properties of a standard Brownian motion include the following:

- The standard Brownian motion starts from zero.
- The standard Brownian motion takes a continuous path.
- The increases suffered by the Brownian process are independent.
- The increases suffered by the Brownian process in the time interval, *dt*, indicate a Gaussian distribution, with an average that is equal to zero and a variance that is equal to the time interval, *dt*.

Based on these properties, we can consider the process as the sum of a large number of extremely small increments. After choosing two instants, *t* and *s*, the random variable, *Y (s) -Y (t)*, follows a normal distribution, with a mean of μ (s-t) and variance of σ^2 *(s-t)*, which we can represent using the following equation:

$$Y(s) - Y(t) \sim \mathcal{N}(\mu(s-t), \sigma^2(s-t))$$

The hypothesis of normality is very important in the context of linear transformations. In fact, the standard Brownian motion takes its name from the type of distribution that is a standard normal distribution, with parameters of μ = 0 and σ^2= 1.

Therefore, it can be said that the Brownian motion, *Y (t)*, with a unit mean and variance can be represented as a linear transformation of a standard Brownian motion, according to the following equation:

$$Y(t) = Y(0) + \mu * t + \sigma * Z(t)$$

In the previous equation, we can observe the following:

- $Z(t)$ is the standard Brownian motion.

The weak point of this equation lies in the fact that the probability that *Y (t)* assuming a negative value is positive; in fact, since *Z (t)* is characterized by independent increments, which can assume a negative sign, the risk of the negativity of *Y (t)* is not zero.

Now, consider the Brownian motion (the Wiener process) for sufficiently small time intervals. An infinitesimal increment of this process is obtained in the following form:

$$Z_{(t+dt)} - Z_{(t)} = \delta Z_t = N * \sqrt{dt}$$

The previous equation can be rewritten as follows:

$$\frac{Z_{(t+dt)} - Z_{(t)}}{dt} = \frac{N}{\sqrt{dt}}$$

This process is not limited in variation, and, therefore, cannot be differentiated in the context of classical analysis. In fact, the previous one tends to infinity for the interval, *dt*.

Addressing the Wiener process as random walk

A Wiener process can be considered a borderline case of random walk. We dealt with a random walk in *Chapter 5, Simulation-Based Markov Decision Processes*. We have seen that the position of a particle at instant *n* will be represented by the following equation:

$$Y_n = Y_{n-1} + Z_n \quad ; \quad n = 1,2, \ldots$$

In the previous formula, we can observe the following:

- Y_n is the next value in the walk.
- Y_{n-1} is the observation in the previous time phase.
- Z_n is the random fluctuation in that step.

If the *n* random numbers, Z_n, have a mean equal to zero and a variance equal to 1, then, for each value of *n*, we can define a stochastic process using the following equation:

$$Y_n(t) = \frac{1}{\sqrt{n}} * \sum_k Z_k$$

The preceding formula can be used in an iterative process. For very large values of *n*, we can write the following:

$$Y_n(s) - Y_n(t) \sim \mathcal{N}(0, (s - t))$$

The previous formula is due to the central limit theorem that we covered in *Chapter 4, Monte Carlo Simulations*.

Implementing a standard Brownian motion

So, let's demonstrate how to generate a simple Brownian motion in the Python environment. Let's start with the simplest case, in which we define the time interval, the number of steps to be performed, and the standard deviation:

1. We start by importing the following libraries:

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

   The `numpy` library is a Python library containing numerous functions that can help us in the management of multidimensional matrices. Additionally, it contains a large collection of high-level mathematical functions that we can use to operate on these matrices.

   The `matplotlib` library is a Python library used for printing high-quality graphics. With `matplotlib`, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and more using just a few commands. This includes a collection of command-line functions similar to those provided by the MATLAB software.

2. Now, let's proceed with some initial settings:

```
np.random.seed(4)
n = 1000
SQN = 1/np.math.sqrt(n)
ZValues = np.random.randn(n)
Yk = 0
SBMotion=list()
```

In the first line of code, we used the `random.seed()` function to initialize the seed of the random number generator. In this way, the simulation that uses random numbers will be reproducible. The reproducibility of the experiment will be guaranteed by the fact that the random numbers that are generated will always be the same. We set the number of the iterations (*n*), and we calculated the first term of the following equation:

$$Y_n(t) = \frac{1}{\sqrt{n}} * \sum_k Z_k$$

Then, we generated the *n* random numbers using the `random.randn()` function. This function returns a standard normal distribution of *n* samples with a mean of 0 and a variance of 1. Finally, we set the first value of the Brownian motion as required from the properties, $(Y_{(0)}=0)$, and we initialized the list that will contain the Brownian motion location coordinates.

3. At this point, we will use a `for` loop to calculate all of the *n* positions:

```
for k in range(n):
    Yk = Yk + SQN*ZValues[k]
    SBMotion.append(Yk)
```

We simply added the current random number, multiplied by SQN, to the variable that contains the cumulative sum. The current value is then appended to the SBMotion list.

4. Finally, we draw a graph of the Brownian motion created:

```
plt.plot(SBMotion)
plt.show()
```

The following graph is printed:

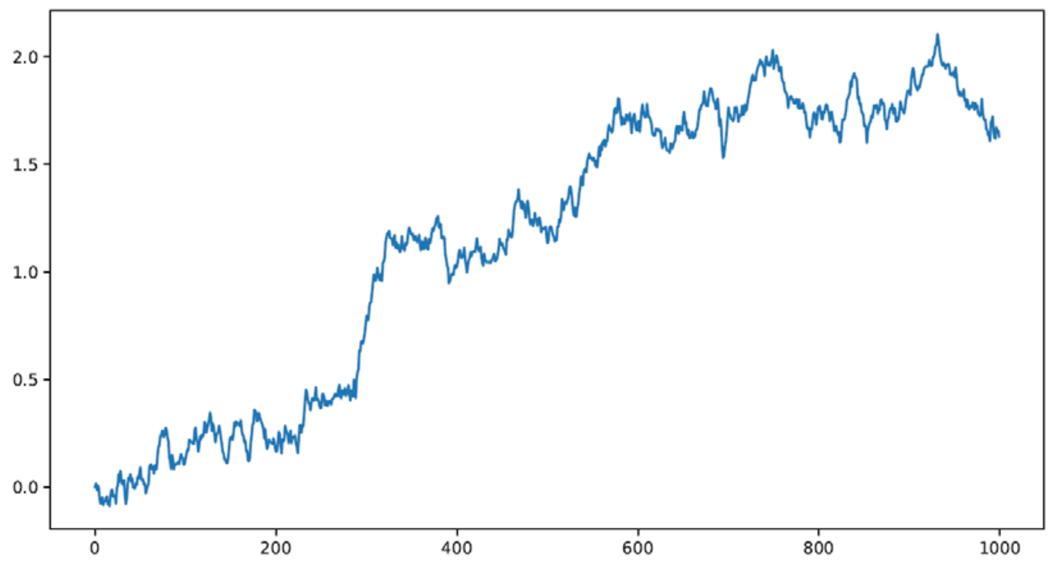


Figure 8.1 – Brownian motion graph

So, we have created our first simulation of Brownian motion. Its use is particularly suited to financial simulations. In the next section, we will demonstrate how this is done.

Using Monte Carlo methods for stock price prediction

As we explored in *Chapter 4, Monte Carlo Simulations*, Monte Carlo methods simulate different evolutions of the process under examination using different probabilities that the event may occur under certain conditions. These simulations explore the entire parameter space of the phenomenon and return a representative sample. For each sample obtained, measures of the quantities of interest are carried out to evaluate their performance. A correct simulation means that the average value of the result of the process converges to the expected value.

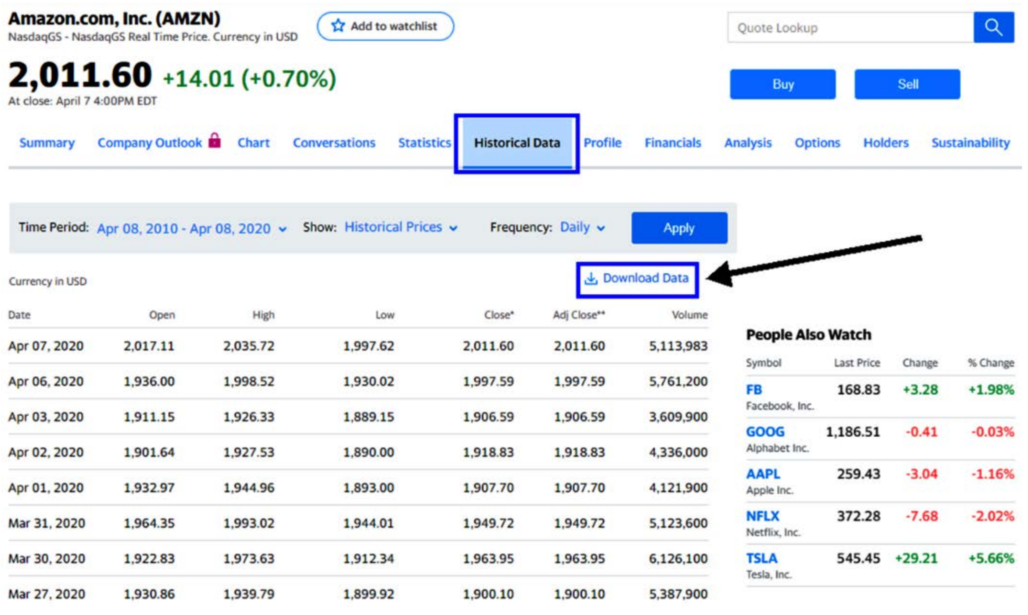
Exploring the Amazon stock price trend

The stock market provides an opportunity to quickly earn large amounts of money, that is, in the eyes of an inexperienced user at least. Exchanges on the stock market can cause large fluctuations in price attracting the attention of speculators from all over the world. In order to obtain revenues from investments in the stock market, it is necessary to have solid knowledge obtained from years of in-depth study of the phenomenon. In this context, the possibility of having a tool to predict stock market securities represents a need felt by all.

Let's demonstrate how to develop a simulation model of the stock of one of the most famous companies in the world. Amazon was founded by Jeff Bezos in the 1990s, and it was one of the first companies in the world to sell products via the internet. Amazon stock has been listed on the stock exchange since 1997 under the symbol AMZN. The historical values of AMZN stock can be obtained from various internet sites that have been dealing with the stock market over the past 10 years. We will refer to the performance of AMZN stock on the NASDAQ GS stock quote from **2010-04-08** to **2020-04-07**. In order to get the data from **2020-04-07**, we need to select **2020-04-08** on the Yahoo website as the end date.

Data can be downloaded in `.csv` format from the Yahoo Finance website at `https://finance.yahoo.com/quote/AMZN/history/`.

In the following screenshot, you can see the Yahoo Finance section for AMZN stock with a highlighted button to download the data:



Amazon.com, Inc. (AMZN)
NasdaqGS - NasdaqGS Real Time Price. Currency in USD

2,011.60 +14.01 (+0.70%)
At close: April 7 4:00PM EDT

Time Period: Apr 08, 2010 - Apr 08, 2020 Show: Historical Prices Frequency: Daily

Currency in USD

Date	Open	High	Low	Close*	Adj Close**	Volume
Apr 07, 2020	2,017.11	2,035.72	1,997.62	2,011.60	2,011.60	5,113,983
Apr 06, 2020	1,936.00	1,998.52	1,930.02	1,997.59	1,997.59	5,761,200
Apr 03, 2020	1,911.15	1,926.33	1,889.15	1,906.59	1,906.59	3,609,900
Apr 02, 2020	1,901.64	1,927.53	1,890.00	1,918.83	1,918.83	4,336,000
Apr 01, 2020	1,932.97	1,944.96	1,893.00	1,907.70	1,907.70	4,121,900
Mar 31, 2020	1,964.35	1,993.02	1,944.01	1,949.72	1,949.72	5,123,600
Mar 30, 2020	1,922.83	1,973.63	1,912.34	1,963.95	1,963.95	6,126,100
Mar 27, 2020	1,930.86	1,939.79	1,899.92	1,900.10	1,900.10	5,387,900

People Also Watch

Symbol	Last Price	Change	% Change
FB Facebook, Inc.	168.83	+3.28	+1.98%
GOOG Alphabet Inc.	1,186.51	-0.41	-0.03%
AAPL Apple Inc.	259.43	-3.04	-1.16%
NFLX Netflix, Inc.	372.28	-7.68	-2.02%
TSLA Tesla, Inc.	545.45	+29.21	+5.66%

Figure 8.2 – Amazon data on Yahoo Finance

The downloaded AMZN.csv file contains a lot of features, but we will only use two of them, as follows:

- **Date**: Date of quote
- **Close**: Close price

We will analyze the code, line by line, to fully understand the whole process, which will lead us to the simulation of a series of predicting scenarios of the performance of the Amazon stock price:

1. As always, we start by importing the libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm
from pandas.plotting import register_matplotlib_
converters
register_matplotlib_converters()
```

   The following libraries were imported:

   The pandas library is an open source BSD-licensed library that contains data structures and operations to manipulate high-performance numeric values for the Python programming language.

   SciPy is a collection of mathematical algorithms and functions based on NumPy. It has a series of commands and high-level classes to manipulate and display data. With SciPy, functionality is added to Python, making it a data processing and system prototyping environment similar to commercial systems such as MATLAB.

2. Now, let's import the data contained in the AMZN.csv file:

```
AmznData = pd.read_csv('AMZN.csv',header=0,
           usecols = ['Date',Close'],parse_dates=True,
           index_col='Date')
```

We used the `read_csv` module of the `pandas` library, which loads the data in a `pandas` object called `DataFrame`. The following arguments are passed:

`'AMZN.csv'`: The name of the file.

`header=0`: The row number containing the column names and the start of the data. By default, if a non-header row is passed (`header=0`), the column names are inferred from the first line of the file.

`usecols=['Date',Close']`: This argument extracts a subset of the dataset by specifying the column names.

`parse_dates=True`: A Boolean value; if `True`, try parsing the index.

`index_col='Date'`: This allows us to specify the name of the column that will be used as the index of the DataFrame.

3. Now we will explore the imported dataset to extract preliminary information. To do this, we will use the `info()` function, as follows:

```
print(AmznData.info())
```

The following information is printed:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2518 entries, 2010-04-08 to 2020-04-07
Data columns (total 1 columns):
Close    2518 non-null float64
dtypes: float64(1)
memory usage: 39.3 KB
None
```

Here, a lot of useful information is returned: the object class, the number of records present (2,518), the start and end values of the index (`2010-04-08` to `2020-04-07`), the number of columns and the type of data they contain, and other information.

We can also print the first five lines of the dataset, as follows:

```
print(AmznData.head())
```

The following data is printed:

```
                 Close
Date
2010-04-08  140.960007
2010-04-09  140.059998
```

```
2010-04-12  141.199997
2010-04-13  140.160004
2010-04-14  144.279999
```

If we wanted to print a different number of records, it would be enough to specify it by indicating the number of lines to be printed. Similarly, we can print the last 10 records of the dataset:

```
print(AmznData.tail())
```

The following records are printed:

```
                  Close
Date
2020-04-01  1907.699951
2020-04-02  1918.829956
2020-04-03  1906.589966
2020-04-06  1997.589966
2020-04-07  2011.599976
```

An initial quick comparison between the head and the tail allows us to verify that the Amazon stock in the last 10 years has gone from a value of about $140 to about $2,011. This is an excellent deal for Amazon sharesholders.

Using the `describe()` function, we will extract a preview of the data using basic statistics:

```
print(AmznData.describe())
```

The following results are returned:

```
             Close
count  2518.000000
mean    723.943074
std     607.588565
min     108.610001
25%     244.189995
50%     398.995011
75%    1006.467514
max    2170.219971
```

We can confirm the significant increase in value in the last 10 years, but we can also see how the stock has undergone significant fluctuations given the very high value of the standard deviation. This tells us that the sharesholders who were loyal to the shares, and maintained the shares over time, benefited the most from the increase in the shares.

4. After analyzing the preliminary data statistics, we can take a look at the performance of the Amazon shares in the last 10 years by drawing a simple graph:

```
plt.figure(figsize=(10,5))
plt.plot(AmznData)
plt.show()
```

The following `matplotlib` functions were used:

`figure()`: This function creates a new figure, which is empty for now. We set the size of the frame using the `figsize` parameter, which sets the width and height in inches.

`plot()`: This function plots the `AmznData` dataset.

`show()`: This function, when running in IPython with the PyLab mode, displays all the figures and returns to the IPython prompt.

The following graph is printed:

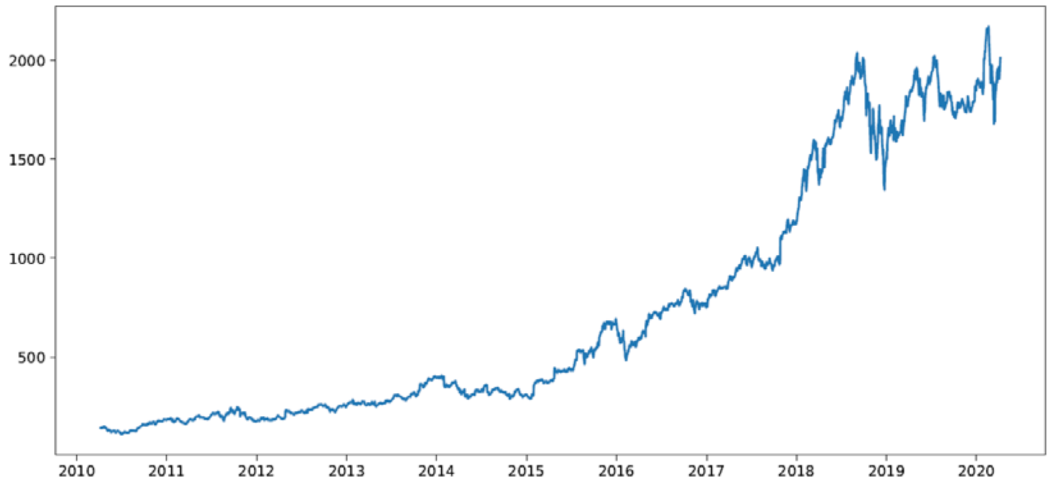


Figure 8.3 – Amazon shares graph

The significant increase undergone by Amazon stock over the past 10 years is evident. Furthermore, it should be noted that the greatest increase has been recorded since 2015, but we'll try to extract more information from the data.

Handling the stock price trend as time series

The trend over time of the Amazon stock price, represented in the previous graph, is configured as a sequence of ordered data. This type of data can be conveniently handled as a time series. Let's consider a simple definition: a time series contains a chronological sequence of experimental observations of a variable. This variable can relate to data of different origins. Very often, it concerns financial data such as unemployment rates, spreads, stock market indices, and stock price trends.

Dealing with the problem as a time series will allow us to extract useful information from the data in order to develop predictive models for the management of future scenarios. It may be useful to compare the trend of stock prices in the same periods for different years or, more simply, between contiguous periods.

Let $Y_1, \ldots, Y_t, \ldots, Y_n$ be the elements of a time series. Let's start by comparing the data for two different times indicated with t and $t + 1$. It is, therefore, two contiguous periods; we are interested in evaluating the variation undergone by the phenomenon under observation, which can be defined by the following ratio:

$$\frac{Y_{t+1} - Y_t}{Y_t} * 100$$

This percentage ratio is called a percentage change. It can be defined as the percentage change rate of Y of time $t + 1$ compared to the previous time, t. This descriptor returns information about how the data underwent a change over a period. The percentage change allows you to monitor both the stock prices and the market indices, not just comparing currencies from different countries:

1. To evaluate this useful descriptor, we will use the `pct_change()` function contained in the `pandas` library:

   ```
   AmznDataPctChange = AmznData.pct_change()
   ```

   This function returns the percentage change between the current element and a previous element. By default, the function calculates the percentage change from the immediately preceding row.

The concept of the percentage variation of a time series is linked to the concept of the return of a stock price. The returns-based approach allows the normalization of data, which is an operation of fundamental importance when evaluating the relationships between variables characterized by different metrics.

We will deal with the return on a logarithmic scale as this choice will give us several advantages: normally distributed results; values returned (logarithm of the return) very close to the initial one (the return), at least for very small values; and additive results over time.

2. To pass the return on a logarithmic scale, we will use the `log()` function of the `numpy` library, as follows:

```
AmznLogReturns = np.log(1 + AmznDataPctChange)
print(AmznLogReturns.tail(10))
```

The following results are printed:

```
                  Close
Date
2020-03-25 -0.028366
2020-03-26  0.036267
2020-03-27 -0.028734
2020-03-30  0.033051
2020-03-31 -0.007272
2020-04-01 -0.021787
2020-04-02  0.005817
2020-04-03 -0.006399
2020-04-06  0.046625
2020-04-07  0.006989
```

3. To better understand how the return is distributed over time, let's draw a graph:

```
plt.figure(figsize=(10,5))
plt.plot(AmznLogReturns)
plt.show()
```

The following graph is printed:

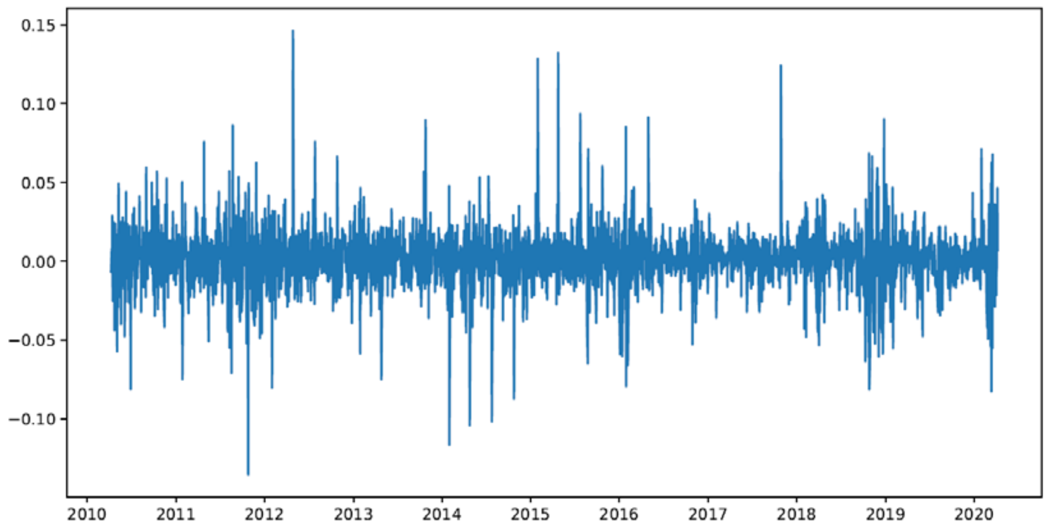


Figure 8.4 – Logarithmic values of the returns

The previous graph shows us that the logarithmic return is normally distributed over the entire period and the mean is stable.

Introducing the Black-Scholes model

The **Black-Scholes** (**BS**) model certainly represents the most important and revolutionary work in the history of quantitative finance. In traditional financial literature, it is assumed that almost all financial asset prices (stocks, currencies, and interest rates) are driven by a Brownian drift motion.

This model assumes that the expected return of an asset is equal to the non-risky interest rate, *r*. This approach is capable of simulating returns on a logarithmic scale of an asset. Suppose we observe an asset in the instants: *t(0), t(1)...,t(n)*. We note, using *s(i) = S(ti)*, the value of an asset at *t(i)*. Based on these hypotheses, we can calculate the return using the following equation:

$$y(i) = \frac{[s(i) - s(i-1)]}{s(i-1)}\ , i = 1,2, \dots, n$$

Then, we will transform the return on the logarithmic scale, as follows:

$$x(i) = \ln s(i) - \ln s(i-1)\ , i = 1,2, \dots, n$$

By applying the BS approach to Brownian geometric motion, the stock price will satisfy the following stochastic differential equation:

$$dS(t) = \mu * S(t) * dt + \sigma * S(t) * dB(t)$$

In the previous equation, *dB(t)* is a standard Brownian motion and μ and σ are real constants. The previous equation is valid where the hypothesis that *s (i) - s (i - 1)* is small, and this happens when the stock prices undergo slight variations. This is because ln *(1 + z)* is roughly equal to *z* if *z* is small. The analytical solution of the previous equation is the following equation:

$$S(t) = S(0) * e^{(\alpha(t)+\sigma*B(t))}$$

By passing the previous equation on a logarithmic scale, we obtain the following equation:

$$\ln\frac{S(t)}{S(0)} = \alpha(t) + \sigma * B(t)$$

In the previous equation, we can observe the following:

- α is the drift.
- B(t) is a standard Brownian motion.
- σ is the standard deviation.

We introduced the concept of drift, which represents the trend of a long-term asset in the stock market. To understand drift, we will use the concept of river currents. If we pour liquid color into a river, it will spread by following the direction imposed by the river's current. Similarly, drift represents the tendency of a stock to follow the trend of a long-term asset.

Applying Monte Carlo simulation

Using the BS model discussed in the previous section, we can evaluate the daily price of an asset starting from that of the previous day multiplied by an exponential contribution based on a coefficient, *r*. This coefficient is a periodic rate of return. It translates into the following equation:

$$StockPrice(t) = StockPrice(t-1) * e^{r}$$

The second term in the previous equation, e^r, is called the **daily return**, and, according to the BS model, it is given by the following formula:

$$e^{(\alpha(t)+\sigma*B(t))}$$

There is no way to predict the rate of return of an asset. The only way to represent it is to consider it as a random number. So, to predict the price trend of an asset, we can use a model based on random movement such as that represented by BS equations.

The BS model assumes that changes in the stock price depend on the expected return over time. The daily return has two terms: the fixed drift rate and the random stochastic variable. The two terms provide for the certainty of movement and uncertainty caused by volatility.

To calculate the drift, we will use the expected rate of return, which is the most likely rate to occur, using the historical average of the log returns and variance, as follows:

$$drift = mean(\log(returns)) - 0.5 * variance(\log\,(returns))$$

According to the previous equation, the daily change rate of the asset is the mean of the returns, which are less than half of the variance over time. We continue our work, calculating the drift for the return of the Amazon security calculated in the *Handling the stock price trend as time series* section:

1. To evaluate the drift, we need the mean and variance of the returns. Since we also calculate the standard deviation, we will need the calculation of the daily return:

```
MeanLogReturns = np.array(AmznLogReturns.mean())
VarLogReturns = np.array(AmznLogReturns.var())
StdevLogReturns = np.array(AmznLogReturns.std())
```

   Three `numpy` functions were used:

   `mean()`: This computes the arithmetic mean along the specified axis and returns the average of the array elements.

   `var()`: This computes the variance along the specified axis. It returns the variance of the array elements, which is a measure of the spread of a distribution.

   `std()`: This computes the standard deviation along the specified axis.

   Now we can calculate the drift as follows:

```
Drift = MeanLogReturns - (0.5 * VarLogReturns)
print("Drift = ",Drift)
```

The following result is returned:

```
Drift =  [0.00086132]
```

This is the fixed part of the Brownian motion. The drift returns the annualized change in the expected value and compensates for the asymmetry in the results compared to the straight Brownian motion.

2. To evaluate the second component of the Brownian motion, we will use the random stochastic variable. This corresponds to the distance between the mean and the events, expressed as the number of standard deviations. Before doing this, we need to set the number of intervals and iterations. The number of intervals will be equal to the number of observations, which is 2,518, while the number of iterations, which represents the number of simulation models that we intend to develop, is 20:

```
NumIntervals = 2518
Iterations = 20
```

3. Before generating random values, it is recommended that you set the seed to make the experiment reproducible:

```
np.random.seed(7)
```

Now, we can generate the random distribution:

```
SBMotion = norm.ppf(np.random.rand(NumIntervals,
Iterations))
```

A 2518 x 20 matrix is returned, containing the random contribution for the 20 simulations that we want to perform and for the 2,518 time intervals that we want to consider. Recall that these intervals correspond to the daily prices of the last 10 years.

Two functions were used:

`norm.ppf()`: This SciPy function gives the value of the variate for which the cumulative probability has the given value.

`np.random.rand()`: This NumPy function computes random values in a given shape. It creates an array of the given shape and populates it with random samples from a uniform distribution over [0, 1].

We will calculate the daily return as follows:

```
DailyReturns = np.exp(Drift + StdevLogReturns * SBMotion)
```

The daily return is a measure of the change that occurred in a stock's price. It is expressed as a percentage of the previous day's closing price. A positive return means the stock has grown in value, while a negative return means it has lost value. The `np.exp()` function was used to calculate the exponential of all elements in the input array.

4. After long preparation, we have arrived at a crucial moment. We will be able to carry out predictions based on the Monte Carlo method. The first thing to do is to recover the starting point of our simulation. Since we want to predict the trend of Amazon stock prices, we recover the first value present in the `AMZN.csv` file:

```
StartStockPrices = AmznData.iloc[0]
```

The pandas `iloc()` function is used to return a pure integer using location-based indexing for selection. Then, we will initialize the array that will contain the predictions:

```
StockPrice = np.zeros_like(DailyReturns)
```

The `numpy zeros_like()` function is used to return an array of zeros with the same shape and type as a given array. Now, we will set the starting value of the `StockPrice` array, as follows:

```
StockPrice[0] = StartStockPrices
```

5. To update the predictions of the Amazon stock prices, we will use a `for` loop that iterates for a number that is equal to the time intervals we are considering:

```
for t in range(1, NumIntervals):
    StockPrice[t] = StockPrice[t - 1] * DailyReturns[t]
```

For the update, we will use the BS model according to the following equation:

$$StockPrice(t) = StockPrice(t-1) * e^{r}$$

$$= StockPrice(t-1) * e^{(\alpha(t)+\sigma * B(t))}$$

Finally, we can view the results:

```
plt.figure(figsize=(10,5))
plt.plot(StockPrice)
AMZNTrend = np.array(AmznData.iloc[:, 0:1])
plt.plot(AMZNTrend,'k*')
plt.show()
```

The following graph is printed:

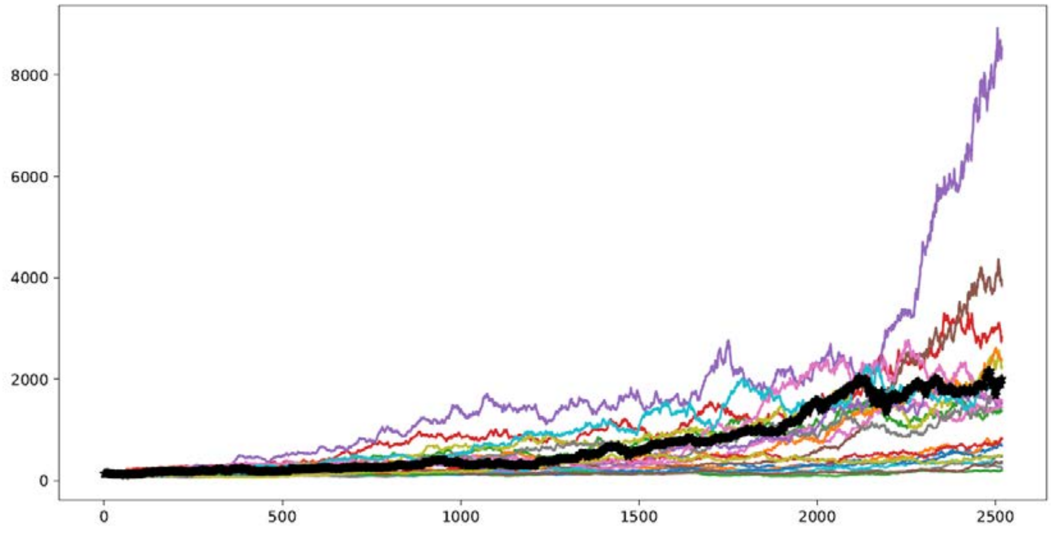


Figure 8.5 – Amazon trend graph

In the previous graph, the curve highlighted in black represents the trend of the Amazon stock prices in the last 10 years. The other curves are our simulations. We can see that some curves move away from the expected curve, while others appear much closer to the actual trend.

Studying risk models for portfolio management

Having a good risk measure is of fundamental importance in finance, as it is one of the main tools for evaluating financial assets. This is because it allows you to monitor securities and provides a criterion for the construction of portfolios. One measure, more than any other, that has been widely used over the years is **variance**.

Using variance as a risk measure

The advantage of a diversified portfolio in terms of risk and the expected value allows us to find the right allocation for the securities. Our aim is to obtain the highest expected value at the same risk or to minimize the risk of obtaining the same expected value. To achieve this, it is necessary to trace the concept of risk back to a measurable quantity, which is generally referred to as the variance. Therefore, by maximizing the expected value of the portfolio returns for each level of variance, it is possible to reconstruct a curve called the efficient frontier, which determines the maximum expected value that can be obtained with the securities available for the construction of the portfolio for each level of risk.

The minimum variance portfolio represents the portfolio with the lowest possible variance value regardless of the expected value. This parameter has the purpose of optimizing the risk represented by the variance of the portfolio. Tracing the risk exclusively to the measure of variance is optimal only if the distribution of returns is normal. In fact, the normal distribution enjoys some properties that make the variance a measure that is enough to represent the risk. It is completely determinable through only two parameters (mean and variance). It is, therefore, enough to know the mean and the variance to determine any other point of the distribution.

Introducing the value-at-risk metric

Consider the variance as the only risk measure in the case of non-normal and limiting values. A risk measure that has been widely used for over two decades is **Value at Risk** (**VaR**). The birth of VaR was linked to the growing need for financial institutions to manage risk and, therefore, be able to measure it. This is due to the increasingly complex structure of financial markets.

Actually, this measure was not introduced to stem the limits of variance as a risk measure since an approach to calculate the VaR value starts precisely from the assumptions of normality. However, to make it easier to understand, let's enclose the overall risk of a security into a single number or a portfolio of financial assets by adopting a single metric for different types of risk.

In the financial context, the VaR is an estimate, given a confidence interval, of how high the losses of a security or portfolio may be in each time horizon. The VaR, therefore, focuses on the left tail of the distribution of returns, where events with a low probability of realization are located. Indicating the losses and not the dispersion of the returns around their expected value makes it a measure closer to the common idea of risk than variance.

Important note

J.P. Morgan is credited as the bank that made VaR a widespread measure. In 1990, the president of J.P. Morgan, Dennis Weatherstone, was dissatisfied with the lengthy risk analysis reports he received every day. He wanted a simple report that summarized the bank's total exposure across its entire trading portfolio.

After calculating the VaR, we can say that, with a probability given by the confidence interval, we will not lose more than the VaR of the portfolio in the next N days. VaR is the level of loss that will not be exceeded with a probability given by the confidence interval.

For example, a VaR of €1 million over a year with a 95% confidence level means that the maximum loss for the portfolio for the next year will be €1 million in 95% of cases. Nothing tells us what happens to the remaining 5% of cases.

The following graph shows the probability distribution of portfolio returns with the indication of the value of the VaR:

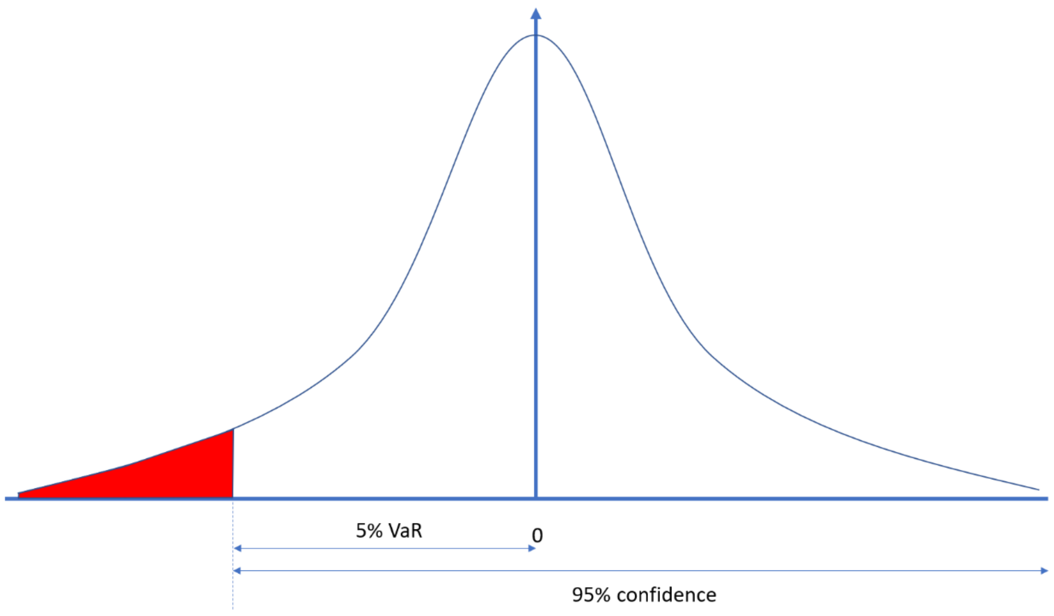


Figure 8.6 – Probability distribution of the portfolio returns

VaR is a function of the following two parameters:

- Time horizon
- Level of confidence

Some characteristics of VaR must be specified:

- VaR does not describe the worst loss.
- VaR says nothing about the distribution of losses in the left tail.
- VaR is subject to sampling errors.

Important note

The sampling error tells us how much the sampled value deviates from the real population value. This deviation is because the sample is not representative of the population or has distortions.

VaR is a widely used risk measure that summarizes, in a single number, important aspects of the risk of a portfolio of financial instruments. It has the same unit of measurement as the returns of the portfolio on which it is calculated, and it is simple to understand, answering the simple question: *How bad can financial investments go?*

Let's now examine a practical case of calculating the VaR.

Estimating the VaR for some NASDAQ assets

NASDAQ is one of the most famous stock market indices in the world. Its name is an acronym for the National Association of Securities Dealers Quotation. This is the index that represents the stocks of the technology sector in the US. Thinking of NASDAQ in the investor's mind, the brands of the main technological and social houses of the US can easily emerge. Just think of companies such as Google, Amazon, Facebook, and many others; they are all covered by the NASDAQ listing.

Here, we will learn how to recover the data of the quotes of six companies listed by NASDAQ, and then we will demonstrate how to estimate the risk associated with the purchase of a portfolio of sharess of these securities:

1. As always, we start by importing the libraries:

```
import datetime as dt
import numpy as np
import pandas_datareader.data as wb
import matplotlib.pyplot as plt
from scipy.stats import norm
```

   The following libraries were imported:

   The `datetime` library contains classes for manipulating dates and times. The functions it contains allow us to easily extract attributes for formatting and manipulating dates.

   The `pandas_datareader.data` module of the pandas library contains functions that allow us to extract financial information, not just from a series of websites that provide these types of data. The collected data is returned in the pandas DataFrame format. The `pandas` library is an open source BSD-licensed library that contains data structures and operations to manipulate high-performance numeric values for the Python programming language.

2. We will set the stocks we want to analyze by defining them with tickers. We also decide the time horizon:

```
StockList = ['ADBE','CSCO','IBM','NVDA','MSFT','HPQ']
StartDay = dt.datetime(2019, 1, 1)
EndDay = dt.datetime(2019, 12, 31)
```

Six tickers have been included in a DataFrame. A ticker is an abbreviation used to uniquely identify the sharess listed on the stock exchange of a particular security on a specific stock market. It is made up of letters, numbers, or a combination of both. The tickers are used to refer to six leading companies in the global technology sector:

`ADBE`: Adobe Systems Inc. – one of the largest and most differentiated software companies in the world.

`CSCO`: Cisco Systems Inc. – the production of **Internet Protocol** (**IP**)-based networking and other products for communications and information technology.

`IBM`: International Business Machines – the production and consultancy of information technology-related products.

`NVDA`: Nvidia Corp. – visual computing technologies. This is the company that invented the GPU.

`MSFT`: Microsoft Corp. – this is one of the most important companies in the sector, as well as one of the largest software producers in the world by turnover.

`HPQ`: HP Inc. – the leading global provider of products, technologies, software, solutions, and services to individual consumers and large enterprises.

After deciding the tickers, we set the time horizon of our analysis. We simply set the start date and end date of our analysis by defining the whole year of 2019.

3. Now we can recover the data:

```
StockData =  wb.DataReader(StockList, 'yahoo',
                                    StartDay,EndDay)
StockClose = StockData["Adj Close"]
print(StockClose.describe())
```

To retrieve the data, we used the `DataReader()` function of the `pandas_datareader.data` module. This function extracts data from various internet sources into a pandas DataFrame. The following topics have been passed:

`StockList`: The list of stocks to be recovered

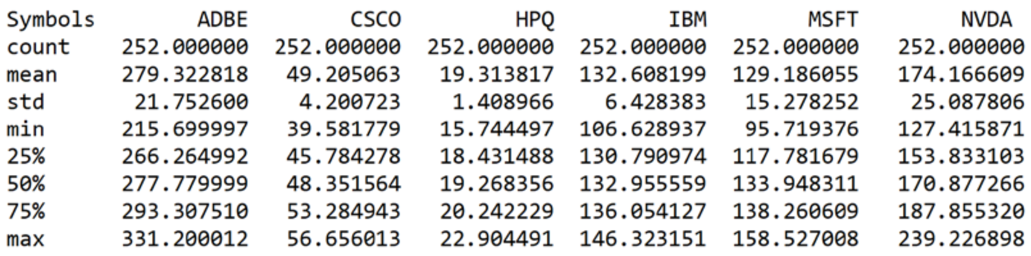
`'yahoo'`: The website from which to collect data

`StartDay`: Start date of monitoring

`EndDay`: End date of monitoring

The recovered data is entered in a pandas DataFrame that will contain 36 columns corresponding to 6 pieces of information for each of the 6 stocks. Each record will contain the following information for each day: the high value, the low value, the open value, the close value, the volume, and the adjusted close.

For the risk assessment of a portfolio, only one value will suffice: the adjusted close. This column was extracted from the starting DataFrame and stored in the `StockData` variable. We then developed basic statistics for each stock using the `describe()` function. The following statistics have been returned:



```
Symbols        ADBE        CSCO         HPQ         IBM        MSFT        NVDA
count    252.000000  252.000000  252.000000  252.000000  252.000000  252.000000
mean     279.322818   49.205063   19.313817  132.608199  129.186055  174.166609
std       21.752600    4.200723    1.408966    6.428383   15.278252   25.087806
min      215.699997   39.581779   15.744497  106.628937   95.719376  127.415871
25%      266.264992   45.784278   18.431488  130.790974  117.781679  153.833103
50%      277.779999   48.351564   19.268356  132.955559  133.948311  170.877266
75%      293.307510   53.284943   20.242229  136.054127  138.260609  187.855320
max      331.200012   56.656013   22.904491  146.323151  158.527008  239.226898
```

Figure 8.7- Statistics of the portfolios

Analyzing the previous table, we can note that there are 252 records. These are the days when the stock exchange was opened in 2019. Let's take note of it as this data will be useful later on. We also note that the values in the columns have very different ranges due to the different values of the stocks. More easily understand the trend of stocks, it is better to draw graphs. Let's do this next:

```
fig, axs = plt.subplots(3, 2)
axs[0, 0].plot(StockClose['ADBE'])
axs[0, 0].set_title('ADBE')
axs[0, 1].plot(StockClose['CSCO'])
axs[0, 1].set_title('CSCO')
axs[1, 0].plot(StockClose['IBM'])
axs[1, 0].set_title('IBM')
```

```
axs[1, 1].plot(StockClose['NVDA'])
axs[1, 1].set_title('NVDA')
axs[2, 0].plot(StockClose['MSFT'])
axs[2, 0].set_title('MSFT')
axs[2, 1].plot(StockClose['HPQ'])
axs[2, 1].set_title('HPQ')
```

In order to make an easy comparison between the trends of the 6 stocks, we have traced 6 subplots that are ordered in 3 rows and 2 columns. We used the `subplots()` function of the `matplotlib` library. This function returns a tuple containing a figure object and axes. So, when you use `fig, axs = plt.subplots()`, you unpack this tuple into the variables of `fig` and `axs`. Having `fig` is useful if you want to change the attributes at the figure level or save the figure as an image file later. The variable, `axs`, allows us to set the attributes of the axes of each subplot. In fact, we called this variable to define what to draw in each subplot by calling it with the row-column indices of the chart matrix. In addition, for each chart, we also printed the title, which allows us to understand which ticker it refers to.

After doing this, we plot the graph:

```
plt.figure(figsize=(10,5))
plt.plot(StockClose)
plt.show()
```

The following `matplotlib` functions were used:

`figure()`: This function creates a new figure, which is empty for now, and we set the size of the frame using the `figsize` parameter, which sets the width and height in inches.

`plot()`: This function plots the `AmznData` dataset.

`show()`: This function, when running in IPython in PyLab mode, displays all the figures and returns to the IPython prompt.

The following graphs are printed:

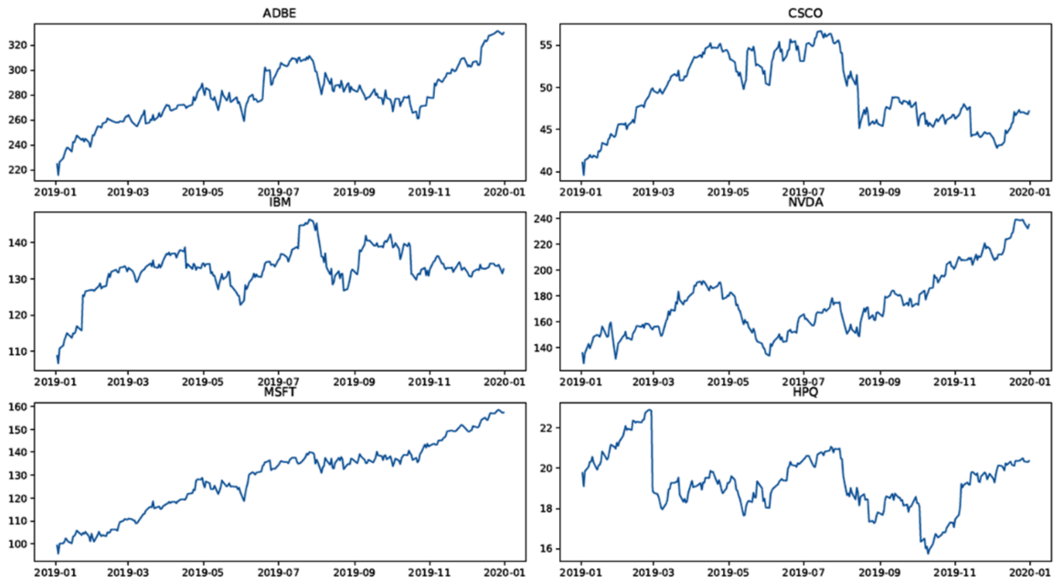


Figure 8.8 – Graphs of the statistics

Analyzing the previous figure, everything is clearer. The trends of stocks are evident. Leaving aside the absolute value, which varies considerably from one stock to another, we can note that only the Microsoft stock has recorded an almost increasing trend throughout the monitoring period. On the contrary, the other stocks have shown fluctuating trends. We also note that the HPQ stock has recorded three sudden falls.

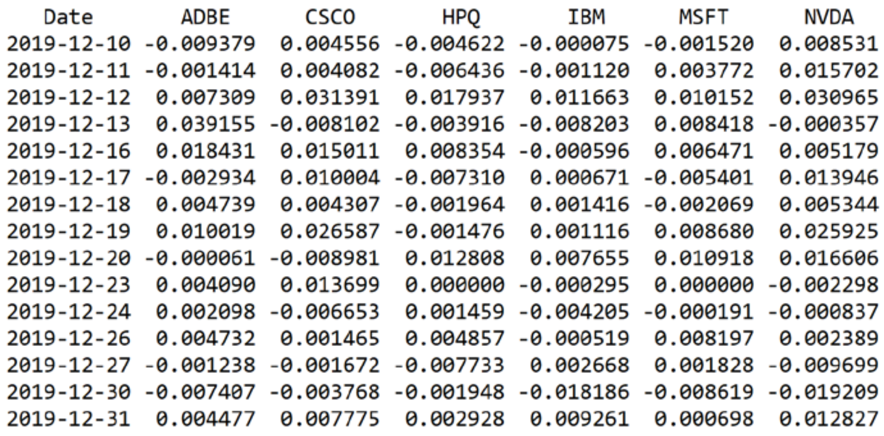
4. After taking a quick look at the trend of stocks, the time has come to evaluate the returns:

```
StockReturns = StockClose.pct_change()
print(StockReturns.tail())
```

The `pct.change()` function returns the percentage change between the current close price and the previous value. By default, the function calculates the percentage change from the immediately preceding row.

The concept of the percentage variation of a time series is linked to the concept of the return of a stock price. The returns-based approach provides for the normalization of data, which is an operation of fundamental importance to evaluate the relationships between variables characterized by different metrics. These concepts have been explored in the *Using Monte Carlo methods for stock price prediction* section of this chapter. Note that we have only referred to some of them.

We then printed the queue of the returned DataFrame to analyze its contents. The following results are returned:



```
    Date        ADBE      CSCO       HPQ       IBM      MSFT      NVDA
2019-12-10 -0.009379  0.004556 -0.004622 -0.000075 -0.001520  0.008531
2019-12-11 -0.001414  0.004082 -0.006436 -0.001120  0.003772  0.015702
2019-12-12  0.007309  0.031391  0.017937  0.011663  0.010152  0.030965
2019-12-13  0.039155 -0.008102 -0.003916 -0.008203  0.008418 -0.000357
2019-12-16  0.018431  0.015011  0.008354 -0.000596  0.006471  0.005179
2019-12-17 -0.002934  0.010004 -0.007310  0.000671 -0.005401  0.013946
2019-12-18  0.004739  0.004307 -0.001964  0.001416 -0.002069  0.005344
2019-12-19  0.010019  0.026587 -0.001476  0.001116  0.008680  0.025925
2019-12-20 -0.000061 -0.008981  0.012808  0.007655  0.010918  0.016606
2019-12-23  0.004090  0.013699  0.000000 -0.000295  0.000000 -0.002298
2019-12-24  0.002098 -0.006653  0.001459 -0.004205 -0.000191 -0.000837
2019-12-26  0.004732  0.001465  0.004857 -0.000519  0.008197  0.002389
2019-12-27 -0.001238 -0.001672 -0.007733  0.002668  0.001828 -0.009699
2019-12-30 -0.007407 -0.003768 -0.001948 -0.018186 -0.008619 -0.019209
2019-12-31  0.004477  0.007775  0.002928  0.009261  0.000698  0.012827
```

Figure 8.9 – The stock returns DataFrame

In the previous table, the minus sign indicates a negative return or a loss.

5. Now we are ready to assess the investment risk of a substantial portfolio of stocks of these prestigious companies. To do this, we need to set some variables and calculate others:

```
PortfolioValue = 1000000000.00
ConfidenceValue = 0.95
Mu = np.mean(StockReturns)
Sigma = np.std(StockReturns)
```

To start, we set the value of our portfolio; it is a billion dollars. These figures should not frighten you. For a bank that manages numerous investors, achieving this investment value is not difficult. So, we set the confidence interval. Previously, we said that VaR is based on this value. Subsequently, we started to calculate some fundamental quantities for the VaR calculation. I am referring to the mean and standard deviation of returns. To do this, we used the related `numpy` functions: `np.mean ()` and `np.std`.

We continue to set the parameters necessary for calculating the VaR:

```
WorkingDays2019 = 252.
AnnualizedMeanStockRet = MeanStockRet/WorkingDays2019
AnnualizedStdStockRet =
                StdStockRet/np.sqrt(WorkingDays2019)
```

Previously, we saw that the data extracted from the finance section of the Yahoo website contained 252 records. This is the number of working days of the stock exchange in 2019, so we set this value. So, let's move on to annualizing the mean and the standard deviation just calculated. This is because we want to calculate the annual risk index of the stocks. For the annualization of the average, it is enough to divide by the number of working days, while for the standard deviation, we must divide by the square root of the number of working days.

6. Now we have all the data we need to calculate the VaR:

```
INPD = norm.ppf(1-ConfidenceValue,AnnualizedMeanStockRet,
                   AnnualizedStdStockRet)
VaR = PortfolioValue*INPD
```

To start, we calculate the inverse normal probability distribution with a risk level of 1 for the confidence, mean, and standard deviation. This technique involves the construction of a probability distribution starting from the three parameters we have mentioned. In this case, we work backward, starting from some distribution statistics, and try to reconstruct the starting distribution. To do this, we use the `norm.ppf()` function of the SciPy library.

The `norm()` function returns a normal continuous random variable. The acronym, **ppf**, stands for **percentage point function**, which is another name for the quantile function. The quantile function, associated with a probability distribution of a random variable, specifies the value of the random variable so that the probability that the variable is less than or equal to that value is equal to the given probability.

At this point, the VaR is calculated by multiplying the inverse normal probability distribution obtained by the value of the portfolio. To make the value obtained more readable, it was rounded to the first two decimal places:

```
RoundVaR=np.round_(VaR,2)
```

Finally, the results obtained were printed, one for each row, to make the comparison simple:

```
for i in range(len(StockList)):
    print("Value-at-Risk for", StockList[i],
                   "is equal to ",RoundVaR[i])
```

The following results are returned:

```
Value-at-Risk for ADBE is equal to  -1547.29
Value-at-Risk for CSCO is equal to  -1590.31
Value-at-Risk for IBM is equal to  -2047.22
Value-at-Risk for NVDA is equal to  -1333.65
Value-at-Risk for MSFT is equal to  -1286.01
Value-at-Risk for HPQ is equal to  -2637.71
```

The stocks that returned the highest risk were HP and IBM, while the one that returned the lowest risk was the Microsoft stock.

Summary

In this chapter, we applied the concepts of simulation based on Monte Carlo methods and, more generally, on the generation of random numbers to real cases related to the world of financial engineering. We started by defining the model based on Brownian motion, which describes the uninterrupted and irregular movement of small particles when immersed in a fluid. We learned how to describe the mathematical model, and then we derived a practical application that simulates a random walk as a Wiener process.

Afterward, we dealt with another practical case of considerable interest, that is, how to use Monte Carlo methods to predict the stock prices of the famous Amazon company. We started to explore the trend of Amazon sharess in the last 10 years, and we performed simple statistics to extract preliminary information on any trends that we confirmed through visual analysis. Subsequently, we learned to treat the trend of stock prices as a time series, calculating the daily return. We then addressed the problem with the BS model, defining the concepts of drift and standard Brownian motion. Finally, we applied the Monte Carlo method to predict possible scenarios relating to the trend of stock prices.

As a final practical application, we assessed the risk associated with a portfolio of sharess of some of the most famous technology companies listed on the NASDAQ market. We first defined the concept of referrals connected to a financial asset, and then we introduced the concept of VaR. Subsequently, we implemented an algorithm that, given a confidence interval and a time horizon, calculates the VaR on the basis of the daily returns returned by the historical data of the stock prices.

In the next chapter, we will learn about the basic concepts of artificial neural networks, how to apply feedforward neural network methods to our data, and how the neural network algorithm works. Then, we will understand the basic concepts of a deep neural network and how to use neural networks to simulate a physical phenomenon.

9
Simulating Physical Phenomena Using Neural Networks

Neural networks are exceptionally effective at getting good characteristics for highly structured data. Physical phenomena are conditioned by numerous variables that can be easily measured through modern sensors. In this way, big data is produced that is difficult to deal with using classic techniques. Neural networks lend themselves to simulating complex environments.

In this chapter, we will learn how to develop models based on **artificial neural networks** (**ANNs**) to simulate physical phenomena. We will start by exploring the basic concepts of neural networks, and then we will examine their architecture and main elements. We will demonstrate how to train a network to update its weights. Then, we will apply these concepts to a practical use case to solve a regression problem. In the last part of the chapter, we will analyze deep neural networks.

In this chapter, we're going to cover the following topics:

- Introducing the basics of neural networks
- Understanding feedforward neural networks
- Simulating airfoil self-noise using ANNs
- Exploring deep neural networks

Technical requirements

In this chapter, we will learn how to use ANNs to simulate complex environments. To understand the topics, basic knowledge of algebra and mathematical modeling is needed.

To work with the Python code in this chapter, you need the following file (available on GitHub at `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `Airfoil Self-Noise.py`

Introducing the basics of neural networks

ANNs are numerical models developed with the aim of reproducing simple neural activities of the human brain, such as object identification and voice recognition. The structure of an ANN is composed of nodes that, similar to the neurons present in a human brain, are interconnected with each other through weighted connections, which reproduce the synapses between neurons.

The system output is updated until it iteratively converges via the connection weights. The information derived from experimental activities is used as input data and the result processed by the network is returned as an output. The input nodes represent the predictive variables, and the output neurons are represented by the dependent variables. We use the predictive variables to process the dependent variables.

ANNs are very versatile in simulating regression and classification problems. They can learn the process of working out the solution to a problem by analyzing a series of examples. In this way, the researcher is released from the difficult task of building a mathematical model of the physical system, which, in some cases, is impossible to represent.

Understanding biological neural networks

ANNs are based on a model that draws inspiration from the functioning principles of the human brain and how the human brain processes the information that comes to it from the peripheral organs. In fact, ANNs consist of a series of neurons that can be thought of as individual processors, since their basic task is to process the information that is provided to them at the input. This processing is similar to the functioning of a biological neuron, which receives electrical signals, processes them, and then transmits the results to the next neuron. The essential elements of a biological neuron include the following:

- Dendrites
- Synapses
- Body cells
- Axon

The information is processed by the biological neuron according to the following steps:

1. Dendrites get information from other neurons in the form of electrical signals.
2. The flow of information occurs through the synapses.
3. The dendrites transmit this information to the cell body.
4. In the cell body, the information is added together.
5. If the result exceeds a threshold limit, the cell reacts by passing the signal to another cell. The passage of information takes place through the axon.

The following diagram shows the essential elements of the structure of a biological neuron:

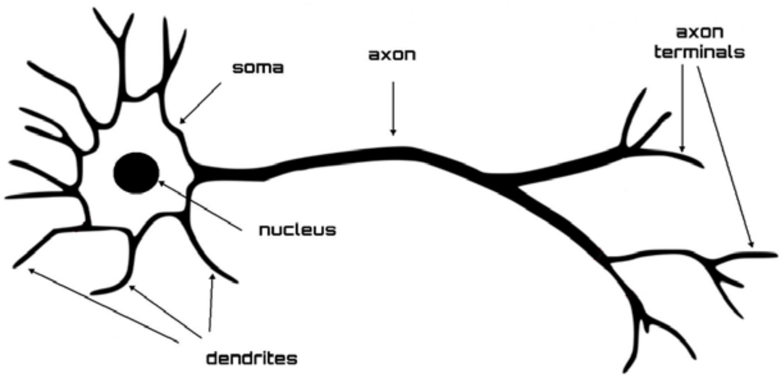


Figure 9.1 – Structure of a neuron

Synapses assume the role of neurotransmitters; in fact, they can exert an excitatory or inhibitory action against the neuron that is immediately after it. This effect is regulated by the synapses through the weight that is associated with them. In this way, each neuron can perform a weighted sum of the inputs, and if this sum exceeds a certain threshold, it activates the next neuron.

Important note

The processing performed by the neuron lasts for a few milliseconds. From a computational point of view, it represents a relatively long time. So, we could say that this processing system, taken individually, is relatively slow. However, as we know, it is a model based on quantity; it is made up of a very high number of neurons and synapses that work simultaneously and in parallel.

In this way, the processing operations that are performed are very effective, and they allow us to obtain results in a relatively short period of time. We can say that the strength of neural networks lies in the teamwork of neurons. Taken individually, they do not represent a particularly effective processing system; however, taken together, they represent an extremely high-performing simulation model.

The functioning of a brain is regulated by neurons and represents an optimized machine that can solve even complex problems. It is a simple structure, improved over time through the evolution of the species. It has no central control; the areas of the brain are all active in carrying out a task, which is aimed at solving a problem. The workings of all parts of the brain take place in a contributory way, and each part contributes to the result. In addition to this, the human brain is equipped with a very effective error regulation system. In fact, if a part of the brain stops working, the operations of the entire system continue to be performed, even if with a lower performance.

Exploring ANNs

As we have stated, a model based on ANNs draws inspiration from the functioning of the human brain. In fact, an artificial neuron is similar to a biological neuron in that it receives information as input derived from another neuron. A neuron's input represents the output of the neuron that is found immediately before in the architecture of a model based on neural networks.

Each input signal to the neuron is then multiplied by the corresponding weight. It is then added to the results obtained by the other neurons to process the activation level of the next neuron. The essential elements of the architecture of a model based on ANNs include the neurons that are distinguished from the input neurons, and the output neurons by the number of layers of synapses and by the connections between these neurons. The following diagram shows the typical architecture of an ANN:

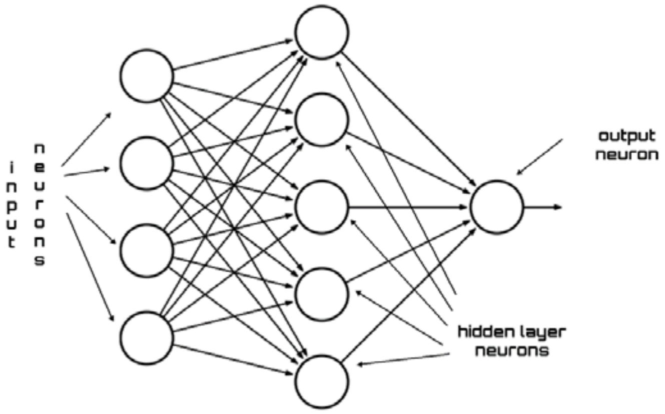


Figure 9.2 – Architecture of an ANN

The input signals, which represent the information detected by the environment, are sent to the input layer of the ANN. In this way, they travel, in parallel, along with the connections through the internal nodes of the system and up to the output. The architecture of the network, therefore, returns a response from the system. Put simply, in a neural network, each node is able to process only local information with no knowledge of the final goal of the processing and not keeping any memory of the latter. The result obtained depends on the architecture of the network and on the values assumed by the artificial synapses.

There are cases in which a single synapse layer is unable to return an adequate network response to the signal supplied at the input. In these cases, multiple layers of synapses are required because a single layer is not enough. These networks are called deep neural networks. The network response is obtained by treating the activation of one layer of neurons at a time and then proceeding from the input to the output, passing through the intermediate layers.

Important note

The ANN target is the result of the calculation of the outputs performed for all the neurons, so an ANN is presented as a set of mathematical function approximations.

The following elements are essential in an ANN architecture:

- Weights
- Bias
- Layers
- Activation functions

In the following sections, we will deepen our understanding of these concepts.

Describing the structure of the layers

In the architecture of an ANN, it is possible to identify the nodes representing the neurons distributed in a form that provides a succession of layers. In a simple structure of an ANN, it is possible to identify an input layer, an intermediate layer (hidden layer), and an output layer, as shown in the following diagram:

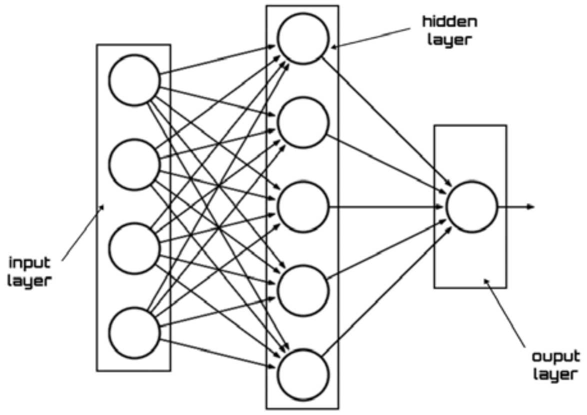


Figure 9.3 – The various layers of an ANN

Each layer has its own task, which it performs through the action of the neurons it contains. The input layer is intended to introduce the initial data into the system for further processing by the subsequent layers. From the input level, the workflow of the ANN begins.

Important note

In the input layer, artificial neurons have a different role to play in some passive way because they do not receive information from the previous levels. In general, they receive a series of inputs and introduce information into the system for the first time. This level then sends the data to the next levels where the neurons receive weighted inputs.

The hidden layer in an ANN is interposed between input levels and output levels. The neurons of the hidden layer receive a set of weighted inputs and produce an output according to the indications received from an activation function. It represents the essential part of the entire network, since it is here that the magic of transforming the input data into output responses takes place.

Hidden levels can operate in many ways. In some cases, the inputs are weighted randomly, while in others, they are calibrated through an iterative process. In general, the neuron of the hidden layer functions as a biological neuron in the brain. That is, it takes its probabilistic input signals, processes them, and converts them into an output corresponding to the axon of the biological neuron.

Finally, the output layer produces certain outputs for the model. Although they are very similar to other artificial neurons in the neural network, the type and number of neurons in the output layer depend on the type of response the system must provide. For example, if we are designing a neural network for the classification of an object, the output layer will consist of a single node that will provide us with this value. In fact, the output of this node must simply provide a positive or negative indication of the presence or absence of the target in the input data. For example, if our network must perform an object classification task, then this layer will contain only one neuron destined to return this value. This is because this neuron must return a binary signal, that is, a positive or negative response that signals the presence or absence of the object among the data provided as input.

Analyzing weights and biases

In a neural network, weights represent a crucial factor in converting an input signal into the system response. They represent a factor such as the slope of a linear regression line. In fact, the weight is multiplied by the inputs and the result is added to the other contributions. These are numerical parameters that determine the contribution of a single neuron in the formation of the output.

If the inputs are $x_1, x_2, \dots x_n$, and the synaptic weights to be applied to them are denoted as $w_1, w_2, \dots w_n$, the output returned by the neuron is expressed through the following formula:

$$y = f(x) = \sum_{i=1}^{n} x_i * w_i$$

The previous formula is a matrix multiplication to reach the weighted sum. The neuron elaboration can be denoted as follows:

$$Output = \sum_{i=1}^{n} x_i * w_i + bias$$

In the previous formula, the bias assumes the role of the intercept in a linear equation. The bias represents an additional parameter that is used to regulate the output along with the weighted sum of the inputs to the neuron.

The input of the next layer is the output of the neurons in the previous layer, as shown in the following diagram:

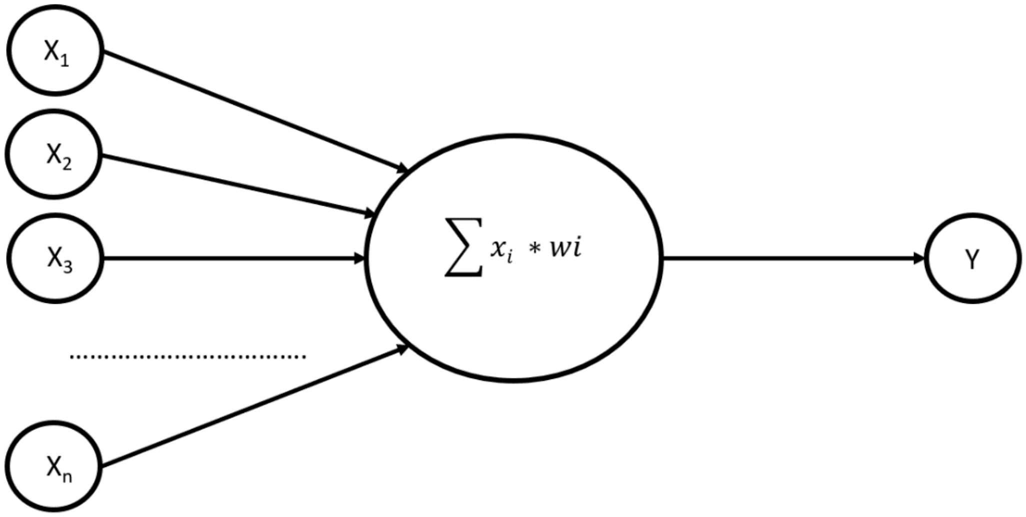


Figure 9.4 – Output of the neurons

The schema presented in the previous diagram explains the role played by the weight in the formation of a neuron. Note that the input provided to the neuron is weighed with a real number that reproduces the activity of the synapse of a biological neuron. When the weight value is positive, the signal has an excitatory activity. If, on the other hand, the value is negative, the signal is inhibitory. The absolute value of the weight indicates the strength of the contribution to the formation of the system response.

Explaining the role of activation functions

The abstraction of neural network processing is primarily obtained via the activation functions. This is a mathematical function that transforms the input into output and controls a neural network process. Without the contribution of activation functions, a neural network can be assimilated to a linear function. A linear function occurs when the output is directly proportional to the input. For example, let's analyze the following equation:

$$y = 5 * x + 3$$

In the previous equation, the exponent of x is equal to 1. This is the condition for the function to be linear: it must be a first-degree polynomial. It is a straight line with no curves. Unfortunately, most real-life problems are nonlinear and complex in nature. To treat nonlinearity, activation functions are introduced in neural networks. Recall that, for a function to be nonlinear, it is sufficient that it is a polynomial function of a degree higher than the first. For example, the following equation defines a nonlinear function of the third degree:

$$y = 5 * x^3 + 3$$

The graph of a nonlinear function is curved and adds to the complexity factor. Activation functions give the nonlinearity property to neural networks and make them true universal function approximators.

There are many activation functions available for a neural network to use. The most used activation functions are listed here:

- **Sigmoid**: The sigmoid function is a mathematical function that produces a sigmoidal curve, which is a characteristic curve for its S shape. This is one of the earliest and most used activation functions. This squashes the input to any value between 0 and 1 and makes the model logistic in nature.
- **Unit step**: A unit step activation function is a much-used feature in neural networks. The output assumes a value of 0 for a negative argument and a value of 1 for a positive argument. The range is between (0,1) and the output is binary in nature. These types of activation functions are useful for binary schemes.
- **Hyperbolic tangent**: This is a nonlinear function, defined in the range of values (-1, 1), so you do not need to worry about activations blowing up. One thing to clarify is that the gradient is stronger for tanh than sigmoid. Deciding between sigmoid and tanh will depend on your gradient strength requirement. Like sigmoid, tanh also has the missing slope problem.

- **Rectified Linear Unit** (**ReLU**): This is a function with linear characteristics for parts of the existence domain that will output the input directly if it is positive; otherwise, it will output zero. The range of output is between 0 and infinity. ReLU finds applications in computer vision and speech recognition using deep neural networks.

The previously listed activation functions are shown in the following diagram:

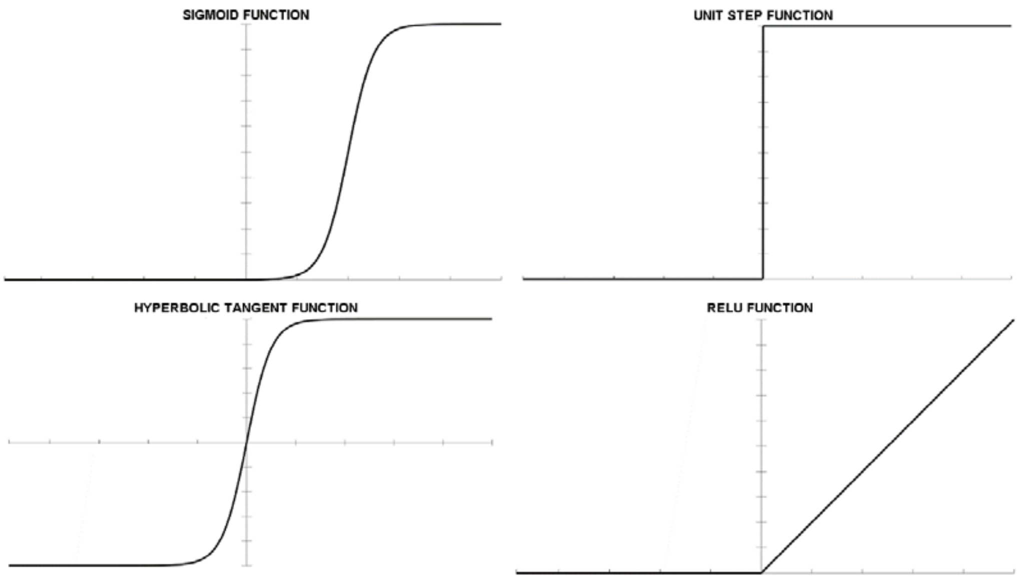


Figure 9.5 – Representation of activation functions

Now we will look at the simple architecture of a neural network, which shows us how the flow of information proceeds.

Understanding feedforward neural networks

The processing from the input layer to the hidden layer(s) and then to the output layer is called feedforward propagation. The transfer function is applied at each hidden layer, and then the activation function value is propagated to the next layer. The next layer can be another hidden layer or the output layer.

Important note

The term "feedforward" is used to indicate the networks in which each node receives connections only from the lower layers. These networks emit a response for each input pattern but fail to capture the possible temporal structure of the input information or to exhibit endogenous temporal dynamics.

Exploring neural network training

The learning ability of an ANN is manifested in the training procedure. This represents the crucial phase of the whole algorithm as it is through the characteristics extracted during training that the network acquires the ability to generalize. The training takes place through a comparison between a series of inputs corresponding to known outputs and those supplied by the model. At least, this is what happens in the case of supervised algorithms in which the labeled data is compared with that provided by the model.

The results achieved by the model are influenced by the data used in the training phase—to obtain good performance, the data used must be sufficiently representative of the phenomenon. From this, we can understand the importance of the dataset used in this phase, which is called the training set.

To understand the procedure through which a neural network learns from data, let's consider a simple example. We will analyze the training of a neural network with a single hidden level. Let's say that the input level has a neuron and the network will have to face a binary classification problem—only two output values of 0 or 1.

The training of the network will take place according to the following steps:

1. Enter the input in the form of a data matrix.
2. Initialize the weights and biases with random values. This step will be performed once, at the beginning of the procedure only. Later weights and biases will be updated through the error propagation process.
3. Apply inputs to the network.
4. Calculate the output for each neuron from the input level, to the hidden levels, to the output level.
5. Calculate the error on the outputs.
6. Use the output error to calculate the error signals for the previous layers. The partial derivative of the activation function is used to calculate the error signals.
7. Use the error signals to calculate the weight adjustments.
8. Apply the weight adjustments.
9. Repeat steps *4* to *9* until the error is minimized.

Steps *3* and *4* represent the direct propagation phase, while steps *5* and *8* represent the backpropagation phase.

The most used method to train a network through the adjustment of neuron weights is the delta rule, which compares the network output with the desired values. Subtract the two values and the difference is used to update all of the input weights, which have different values of zero. The process is repeated until convergence is achieved.

The following diagram shows the weight adjustment procedure:

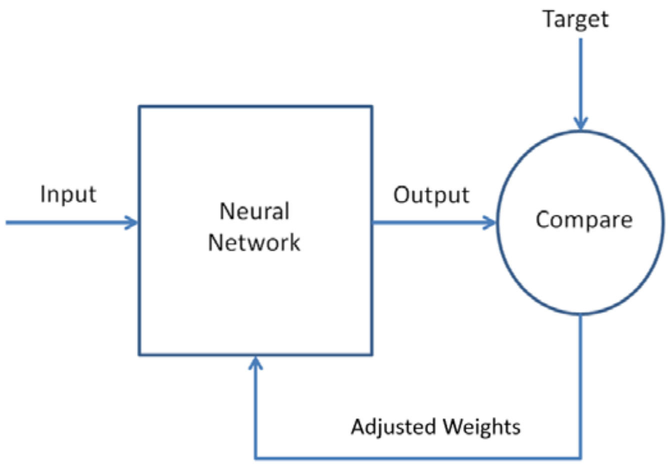


Figure 9.6 – The weight adjustment procedure

The training procedure is extremely simple: it is a simple comparison between the calculated values and the labeled values. The difference between the weighted input values and the expected output values is calculated from the comparison—the difference, which represents the evaluation error between the calculated and expected values, is used to recalculate all of the input weights. It is an iterative procedure that is repeated until the error between the expected and calculated values approaches zero.

Simulating airfoil self-noise using ANNs

The noise generated by an airfoil is due to the interaction between a turbulent airflow and the aircraft's airfoil blades. Predicting the acoustic field in these situations requires an aeroacoustics methodology that can operate in complex environments. Additionally, the method that is used must avoid the formulation of coarse hypotheses regarding geometry, compactness, and the content of the frequency of sound sources. The prediction of the sound generated by a turbulent flow must, therefore, correctly model both the physical phenomena of sound propagation and the turbulence of the flow. Since these two phenomena manifest energy and scales of very different lengths, the correct prediction of the sound generated by a turbulent flow is not easy to model.

Aircraft noise is a crucial topic for the aerospace industry. The NASA Langley Research Center has funded several strands of research to effectively study the various mechanisms of self-noise airfoil. Interest was motivated by its importance for broadband helicopter rotors, wind turbines, and cell noises. The goal of these studies then focused on the challenge of reducing external noises generated by the entire cell of an aircraft by 10 decibels.

In this example, we will elaborate on a model based on ANNs to predict self-noise airfoil from a series of airfoil data measured in a wind tunnel.

Important note

The dataset we will use was developed by NASA in 1989 and is available on the UCI Machine Learning Repository site. The UCI Machine Learning Repository is available at `https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise`.

The dataset was built using the results of a series of aerodynamic and acoustic tests on sections of aerodynamic blades performed in an anechoic wind tunnel.

The following list shows the features of the dataset:

- Number of instances: 1,503
- Number of attributes: 6
- Dataset characteristics: Multivariate
- Attribute characteristics: Real
- Dataset date: 2014-03-04

The following list presents a brief description of the attributes:

- `Frequency`: Frequency in Hertz (Hz)
- `AngleAttack`: Angle of attack in degrees
- `ChordLength`: Chord length in meters
- `FSVelox`: Free-stream velocity in meters per second
- `SSDT`: **Suction-side displacement thickness (SSDT)** in meters
- `SSP`: Scaled sound pressure level in decibels

In the six attributes we have listed, the first five represent the predictors, and the last one represents the response of the system that we want to simulate. It is, therefore, a regression problem because the answer has continuous values. In fact, it represents the self-noise airfoil, in decibels, measured in the wind tunnel.

Importing data using pandas

The first operation we will perform is the importing of data, which, as we have already mentioned, is available on the UCI website. As always, we will analyze the code line by line:

1. We start by importing the libraries. In this case, we will operate differently from what has been done so far. We will not import all of the necessary libraries at the beginning of the code, but we will introduce them in correspondence with their use and we will illustrate their purposes in detail:

   ```
   import pandas as pd
   ```

   The `pandas` library is an open source, BSD-licensed library that provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language. It offers data structures and operations for manipulating numerical data in a simple way. We will use this library to import the data contained in the dataset retrieved from the UCI website.

   The UCI dataset does not contain a header, so it is necessary to insert the names of the variables in another variable. Now, let's put these variable names in the following list:

   ```
   ASNNames=
   ['Frequency','AngleAttack','ChordLength','FSVelox',
   'SSDT','SSP']
   ```

2. Now we can import the dataset. This is available in `.dat` format, and, to make your job easier, it has already been downloaded and is available in this book's GitHub repository:

   ```
   ASNData = pd.read_csv('airfoil_self_noise.dat', delim_
   whitespace=True, names=ASNNames)
   ```

   To import the `.dat` dataset, we used the `read_csv` module of the `pandas` library. In this function, we passed the filename and two other attributes, namely `delim_whitespace` and `names`. The first specifies whether or not whitespace will be used as `sep`, and the second specifies a list of column names to use.

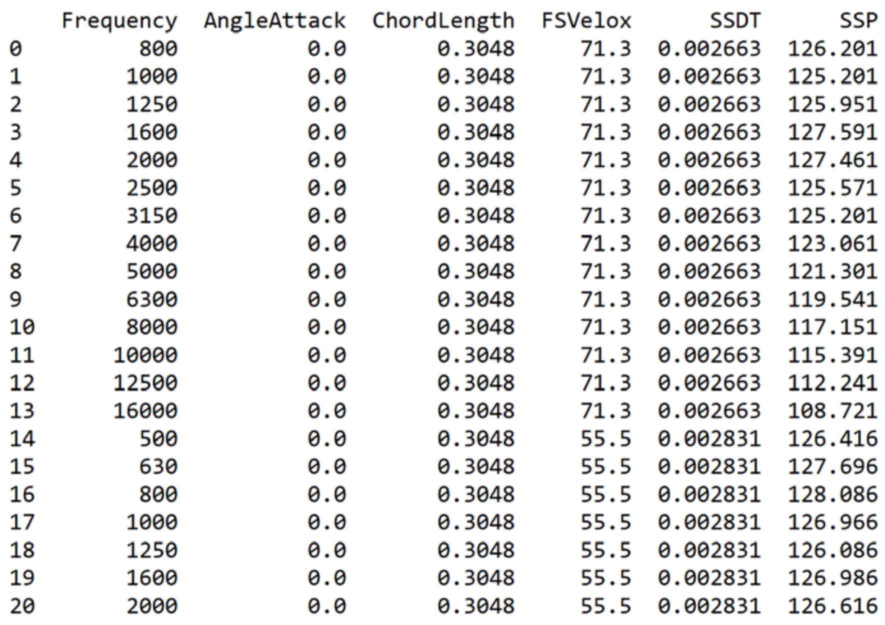
> **Important note**
>
> Remember to set the path so that Python can find the `.dat` file to open.

Before beginning with data analysis through ANN regression, we perform an exploratory analysis to identify how data is distributed and extract preliminary knowledge. To display the first 20 rows of the imported DataFrame, we can use the `head()` function, as follows:

```
print(ASNData.head(20))
```

The pandas `head()` function gets the first *n* rows of a pandas DataFrame. In this case, it returns the first 20 rows for the `ASNData` object based on position. It is used for quickly testing whether our dataset has the right type of data in it. This function, with no arguments, gets the first five rows of data from the DataFrame.

The following data is printed:



```
    Frequency  AngleAttack  ChordLength  FSVelox      SSDT      SSP
0         800          0.0       0.3048     71.3  0.002663  126.201
1        1000          0.0       0.3048     71.3  0.002663  125.201
2        1250          0.0       0.3048     71.3  0.002663  125.951
3        1600          0.0       0.3048     71.3  0.002663  127.591
4        2000          0.0       0.3048     71.3  0.002663  127.461
5        2500          0.0       0.3048     71.3  0.002663  125.571
6        3150          0.0       0.3048     71.3  0.002663  125.201
7        4000          0.0       0.3048     71.3  0.002663  123.061
8        5000          0.0       0.3048     71.3  0.002663  121.301
9        6300          0.0       0.3048     71.3  0.002663  119.541
10       8000          0.0       0.3048     71.3  0.002663  117.151
11      10000          0.0       0.3048     71.3  0.002663  115.391
12      12500          0.0       0.3048     71.3  0.002663  112.241
13      16000          0.0       0.3048     71.3  0.002663  108.721
14        500          0.0       0.3048     55.5  0.002831  126.416
15        630          0.0       0.3048     55.5  0.002831  127.696
16        800          0.0       0.3048     55.5  0.002831  128.086
17       1000          0.0       0.3048     55.5  0.002831  126.966
18       1250          0.0       0.3048     55.5  0.002831  126.086
19       1600          0.0       0.3048     55.5  0.002831  126.986
20       2000          0.0       0.3048     55.5  0.002831  126.616
```

Figure 9.7 – DataFrame output

To extract further information, we can use the `info()` function, as follows:

```
print(ASNData.info())
```

The `info()` method returns a concise summary of the `ASNData` DataFrame, including the `dtypes` index and the `dtypes` column, non-null values, and memory usage.

The following results are returned:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1503 entries, 0 to 1502
Data columns (total 6 columns):
Frequency      1503 non-null int64
AngleAttack    1503 non-null float64
ChordLength    1503 non-null float64
FSVelox        1503 non-null float64
SSDT           1503 non-null float64
SSP            1503 non-null float64
dtypes: float64(5), int64(1)
memory usage: 70.5 KB
None
```

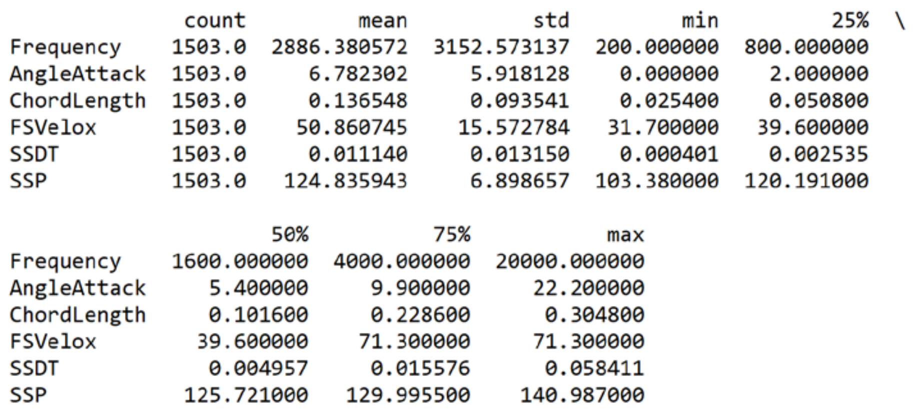
By reading the information returned by the `info` method, we can confirm that it is 1,503 instances and 6 variables. In addition to this, the types of variables returned to us are 5 `float64` variables and 1 `int64` variables.

3. To obtain a first screening of the data contained in the `ASNData` DataFrame, we can compute a series of basic statistics. We can use the `describe()` function in the following way:

```
BasicStats = ASNData.describe()
BasicStats = BasicStats.transpose()
print(BasicStats)
```

The `describe()` function produces descriptive statistics that return the central tendency, dispersion, and shape of a dataset's distribution, excluding **Not-a-Number** (**NaN**) values. It is used for both numeric and object series, as well as the DataFrame columns containing mixed data types. The output will vary depending on what is provided. In addition to this, we have transposed the statistics to appear better on the screen and to make it easier to read the data.

The following statistics are printed:



```
              count         mean          std         min         25%  \
Frequency    1503.0  2886.380572  3152.573137  200.000000  800.000000
AngleAttack  1503.0     6.782302     5.918128    0.000000    2.000000
ChordLength  1503.0     0.136548     0.093541    0.025400    0.050800
FSVelox      1503.0    50.860745    15.572784   31.700000   39.600000
SSDT         1503.0     0.011140     0.013150    0.000401    0.002535
SSP          1503.0   124.835943     6.898657  103.380000  120.191000

                     50%          75%           max
Frequency    1600.000000  4000.000000  20000.000000
AngleAttack     5.400000     9.900000     22.200000
ChordLength     0.101600     0.228600      0.304800
FSVelox        39.600000    71.300000     71.300000
SSDT            0.004957     0.015576      0.058411
SSP           125.721000   129.995500    140.987000
```

Figure 9.8 – Basic statistics of the DataFrame

From the analysis of the previous table, we can extract useful information. First of all, we can note that the data shows great variability. The average of the values ranges from approximately 0.14 to 2,886. Not only that, but some variables have a very large standard deviation. For each variable, we can easily recover the minimum and maximum. In this way, we can note that the interval of the analyzed frequencies goes from 200 to 20,000 Hz. These are just some considerations; we can recover many others.

Scaling the data using sklearn

In the statistics extracted using the `describe()` function, we have seen that the predictor variables (frequency, angle of attack, chord length, free-stream velocity, and SSDT) have an important variability. In the case of predictors with different and varied ranges, the influence on the system response by variables with a larger numerical range could be greater than those with a lower numeric range. This different impact could affect the accuracy of the prediction. Actually, we want to do exactly the opposite, that is, improve the predictive accuracy and reduce the sensitivity of the model from features that can affect prediction due to a wide range of numerical values.

To avoid this phenomenon, we can reduce the values so that they fall within a common range, guaranteeing the same characteristics of variability possessed by the initial dataset. In this way, we will be able to compare variables belonging to different distributions and variables expressed in different units of measurement.

Important note

Recall how we rescale the data before training a regression algorithm. This is a good practice. Using a rescaling technique, data units are removed, allowing us to compare data from different locations easily.

We then proceed with data rescaling. In this example, we will use the min-max method (usually called **feature scaling**) to get all of the scaled data in the range [0, 1]. The formula to achieve this is as follows:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

To do feature scaling, we can apply a preprocessing package offered by the `sklearn` library. This library is a free software machine learning library for the Python programming language. The `sklearn` library offers support for various machine learning techniques, such as classification, regression, and clustering algorithms, including **support vector machines** (**SVMs**), random forests, gradient boosting, k-means, and DBSCAN. `sklearn` is created to work with various Python numerical and scientific libraries such as NumPy and SciPy:

Important note

To import a library that is not part of the initial distribution of Python, you can use the `pip install` command, followed by the name of the library. This command should be used only once and not every time you run the code.

1. The `sklearn.preprocessing` package contains numerous common utility functions and transformer classes to transform the features in a way that works with our requirements. Let's start by importing the package:

   ```
   from sklearn.preprocessing import MinMaxScaler
   ```

   To scale features between the minimum and maximum values, the `MinMaxScaler` function can be used. In this example, we want to rescale the date between zero and one so that the maximum absolute value of each feature is scaled to unit size.

2. Let's start by setting the scaler object:

   ```
   ScalerObject = MinMaxScaler()
   ```

3. To get validation of what we are doing, we will print the object just created in order to check the set parameters:

   ```
   print(ScalerObject.fit(ASNData))
   ```

   The following result is returned:

   ```
   MinMaxScaler(copy=True, feature_range=(0, 1))
   ```

4. Now, we can apply the `MinMaxScaler()` function, as follows:

```
ASNDataScaled = ScalerObject.fit_transform(ASNData)
```

   The `fit_transform()` method fits to the data and then transforms it. Before applying the method, the minimum and maximum values that are to be used for later scaling are calculated. This method returns a NumPy array object.

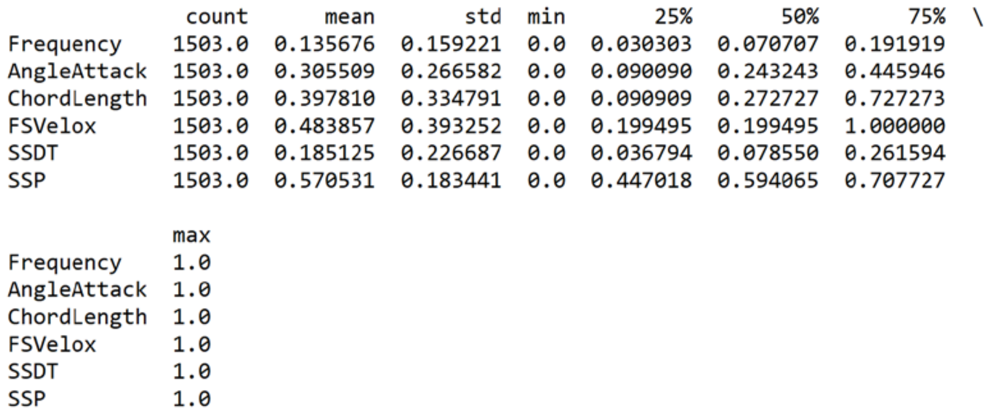
5. Recall that the initial data had been exported in the pandas `DataFrame` format. Scaled data should also be transformed into the same format in mdoo to be able to apply the functions available for pandas DataFrames. The transformation procedure is easy to perform; just apply the pandas `DataFrame()` function as follows:

```
ASNDataScaled = pd.DataFrame(ASNDataScaled,
columns=ASNNames)
```

6. At this point, we can verify the results obtained with data scaling. Let's compute the statistics using the `describe()` function once again:

```
summary = ASNDataScaled.describe()
summary = summary.transpose()
print(summary)
```

   The following statistics are printed:



```
              count      mean       std  min       25%       50%       75%  \
Frequency    1503.0  0.135676  0.159221  0.0  0.030303  0.070707  0.191919
AngleAttack  1503.0  0.305509  0.266582  0.0  0.090090  0.243243  0.445946
ChordLength  1503.0  0.397810  0.334791  0.0  0.090909  0.272727  0.727273
FSVelox      1503.0  0.483857  0.393252  0.0  0.199495  0.199495  1.000000
SSDT         1503.0  0.185125  0.226687  0.0  0.036794  0.078550  0.261594
SSP          1503.0  0.570531  0.183441  0.0  0.447018  0.594065  0.707727

             max
Frequency    1.0
AngleAttack  1.0
ChordLength  1.0
FSVelox      1.0
SSDT         1.0
SSP          1.0
```

Figure 9.9 – Output with scaled data

From the analysis of the previous table, the result of the data scaling appears evident. Now all six variables have values between 0 and 1.

Viewing the data using matplotlib

Now, we will try to have a confirmation of the distribution of the data through a visual approach:

1. To start, we will draw a boxplot, as follows:

   ```
   import matplotlib.pyplot as plt
   boxplot = ASNDataScaled.boxplot(column=ASNNames)
   plt.show()
   ```

   A boxplot, also referred to as a whisker chart, is a graphical description used to illustrate the distribution of data using dispersion and position indices. The rectangle (box) is delimited by the first quartile (25th percentile) and the third quartile (75th percentile) and divided by the median (50th percentile). In addition, there are two whiskers, one upper and one lower, indicating the maximum and minimum distribution values excluding any anomalous values.

   `matplotlib` is a Python library for printing high-quality graphics. With `matplotlib`, it is possible to generate graphs, histograms, bar graphs, power spectra, error graphs, scatter graphs, and more with just a few commands. This is a collection of command-line functions similar to those provided by the MATLAB software.

   As we mentioned earlier, the scaled data is in pandas `DataFrame` format. So, it is advisable that you use the `pandas.DataFrame.boxplot` function. This function makes a boxplot of the DataFrame columns, which are optionally grouped by some other columns.

   The following diagram is printed:

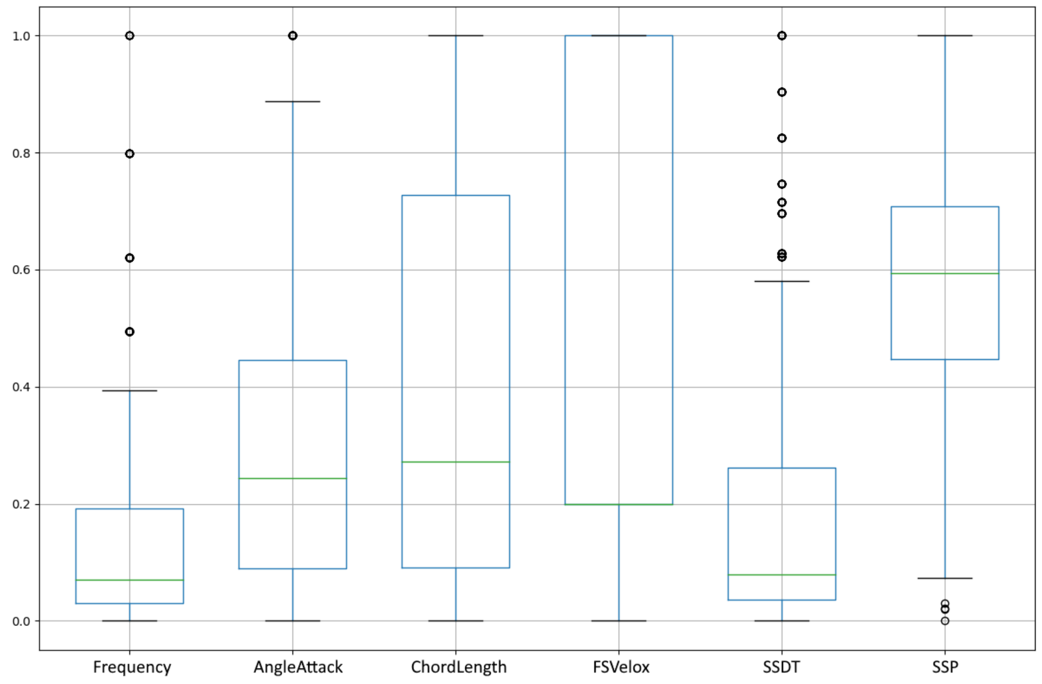


Figure 9.10 – Boxplot of the DataFrame

In the previous diagram, you can see that there are some anomalous values indicated by small circles at the bottom and side of the extreme whiskers of each box. Three variables have these values, called outliers, and their presence can create problems in the construction of the model. Furthermore, we can verify that all the variables are contained in the extreme values that are equal to 0 and 1; this is the result of data scaling. Finally, some variables such as `FSVelox` show great variability of values compared to others, for example, SSDT.

We now measure the correlation between the predictors and the response variable. A technique for measuring the relationship between two variables is offered by correlation, which can be obtained using covariance. To calculate the correlation coefficients in Python, we can use the `pandas.DataFrame.corr()` function. This function computes the pairwise correlation of columns, excluding NA/null values. Three procedures are offered, as follows:

`pearson` (standard correlation coefficient)

`kendall` (Kendall Tau correlation coefficient)

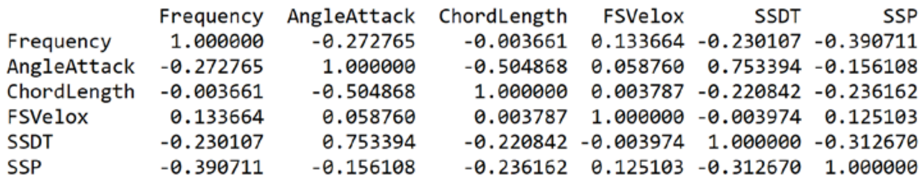
`spearman` (Spearman rank correlation)

> **Important note**
>
> Remember that the correlation coefficient of two random variables is a measure of their linear dependence.

2. Let's calculate the correlation coefficients for the data scaled:

```
CorASNData = ASNDataScaled.corr(method='pearson')
with pd.option_context('display.max_rows', None,
    'display.max_columns', CorASNData.shape[1]):
print(CorASNData)
```

To show all data columns in a screenshot, we used the `option_context` function. The following results are returned:



```
             Frequency  AngleAttack  ChordLength   FSVelox      SSDT       SSP
Frequency     1.000000    -0.272765    -0.003661  0.133664 -0.230107 -0.390711
AngleAttack  -0.272765     1.000000    -0.504868  0.058760  0.753394 -0.156108
ChordLength  -0.003661    -0.504868     1.000000  0.003787 -0.220842 -0.236162
FSVelox       0.133664     0.058760     0.003787  1.000000 -0.003974  0.125103
SSDT         -0.230107     0.753394    -0.220842 -0.003974  1.000000 -0.312670
SSP          -0.390711    -0.156108    -0.236162  0.125103 -0.312670  1.000000
```

Figure 9.11 – Data columns in the DataFrame

We are interested in studying the possible correlation between the predictors and the system response. So, to do this, it will be sufficient to analyze the last row of the previous table. Recall that the values of the various correlation indices vary between -1 and +1; both extreme values represent perfect relationships between the variables, while 0 represents the absence of a relationship. This is if we consider linear relationships. Based on this, we can say that the predictors that show a greater correlation with the response (**SSP**) are **Frequency** and **SSDT**. Both show a negative correlation.

To see visual evidence of the correlation between the variables, we can plot a correlogram. A correlogram graphically presents a correlation matrix. It is used to focus on the most correlated variables in a data table. In a correlogram, correlation coefficients are shown with as nuances that depend on our values. Next to the graph, a colored bar will be proposed in which the corresponding nuanced values of the correlation coefficient can be read. To plot a correlogram, we can use the `matplotlib.pyplot.matshow()` function, which shows a DataFrame as a matrix in a new figure window.

3. Let's plot a correlogram, as follows:

```
plt.matshow(CorASNData)
plt.xticks(range(len(CorASNData.columns)), CorASNData.
columns)
plt.yticks(range(len(CorASNData.columns)), CorASNData.
columns)
plt.colorbar()
plt.show()
```

The following diagram is returned:

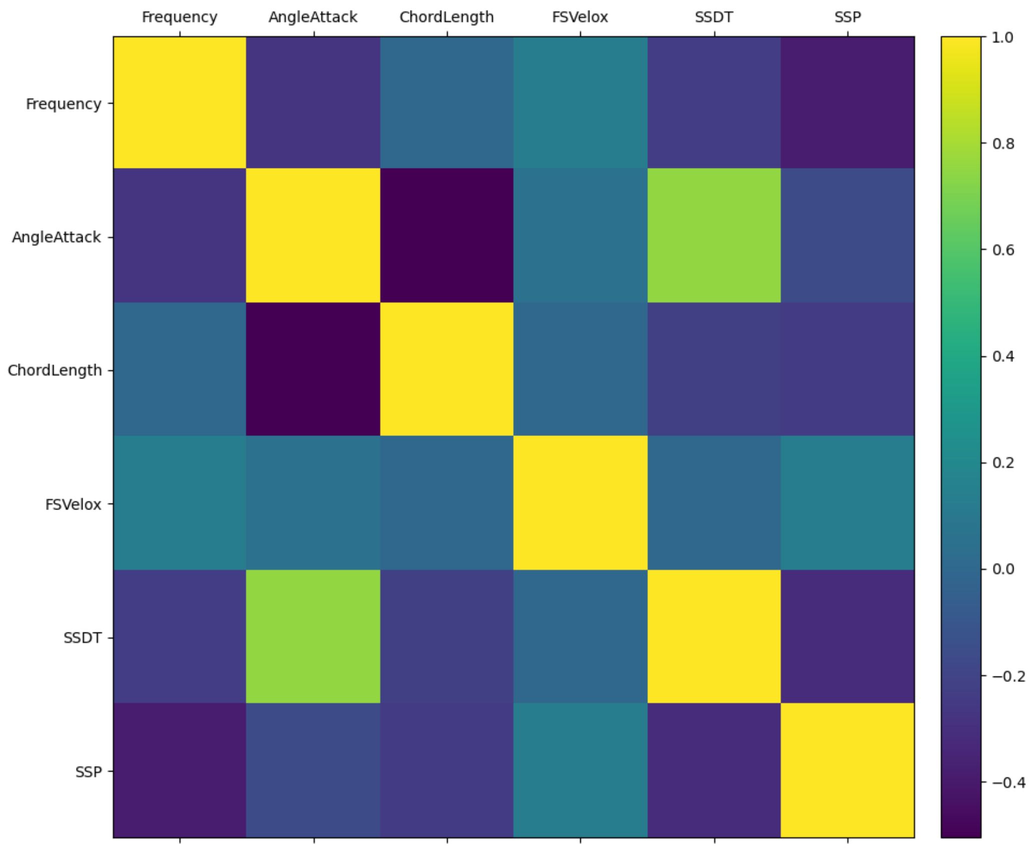


Figure 9.12 – Correlogram of the DataFrame

As we already did in the case of the correlation matrix, in this case too, to analyze the correlation between predictors and the system response, it will be sufficient to consider the bottom row of the graph. The trends already obtained from the correlation matrix are confirmed.

Splitting the data

The training of an algorithm, based on machine learning, represents a crucial phase of the whole process of elaboration of the model. Performing the training of an ANN on the same dataset, which will subsequently be used to test the network, represents a methodological error. This is because the model will be able to perfectly predict the data used for testing, having already seen them in the training phase. However, when it will then be used to predict new cases that have never been seen before, it will inexorably commit evaluation errors. This problem is called data overfitting. To avoid this error, it is good practice to train a neural network to use a different set of data from the one used in the test phase. Therefore, before proceeding with the training phase, it is recommended that you perform a correct division of the data.

Data splitting is used to split the original data into two sets: one is used to train the model and the other to test the model's performance. The training and testing procedures represent the starting point for the model setting in predictive analytics. In a dataset that has 100 observations for predictor and response variables, a data splitting example occurs that divides this data into 70 rows for training and 30 rows for testing. To perform good data splitting, the observations must be selected randomly. When the training data is extracted, the data will be used to upload the weights and biases until an appropriate convergence is achieved.

The next step is to test the model. To do this, the remaining observations contained in the test set will be used to check whether the actual output matches the predicted output. To perform this check, several metrics can be adopted to validate the model:

1. We will use the `sklearn` library to split the `ASNDataScaled` DataFrame. To start, we will import the `train_test_split` function:

   ```
   from sklearn.model_selection import train_test_split
   ```

2. Now, we can divide the starting data into two sets: the `X` set containing the predictors and the `Y` set containing the target. We will use the `pandas.DataFrame.drop()` function, as follows:

   ```
   X = ASNDataScaled.drop('SSP', axis = 1)
   print('X shape = ',X.shape)
   Y = ASNDataScaled['SSP']
   print('Y shape = ',Y.shape)
   ```

3. Using the `pandas.DataFrame.drop()` function, we can remove rows or columns indicating label names and the corresponding axis or the index or column names. In this example, we have removed the target column (`SSP`) from the starting DataFrame.

   The following shapes are printed:

   ```
   X shape = (1503, 5)
   Y shape = (1503,)
   ```

   As is our intention, the two datasets, `X` and `Y`, now contain the 5 predictors and the system response, respectively.

4. Now we can divide the two datasets, `X` and `Y`, into two further datasets that will be used for the training phase and the test phase, respectively:

   ```
   X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
   test_size = 0.30, random_state = 5)
   print('X train shape = ',X_train.shape)
   print('X test shape = ', X_test.shape)
   print('Y train shape = ', Y_train.shape)
   print('Y test shape = ',Y_test.shape)
   ```

   The `train_test_split()` function was used by passing the following four parameters:

   `X`: The predictors.

   `Y`: The target.

   `test_size`: This parameter represents the proportion of the dataset to include in the test split. The following types are available: float, integer or none, and optional (default = 0.25).

   `random_state`: This parameter sets the seed used by the random number generator. In this way, the repetitive splitting of the operation is guaranteed.

   The following results are returned:

   ```
   X train shape =  (1052, 5)
   X test shape =  (451, 5)
   Y train shape =  (1052,)
   Y test shape =  (451,)
   ```

As expected, we finally divided the initial dataset into four subsets. The first two, `X_train` and `Y_train`, will be used in the training phase. The remaining two, `X_test` and `Y_test`, will be used in the testing phase.

Explaining multiple linear regression

In this section, we will deal with a regression problem using ANNs. To evaluate the results effectively, we will compare them with a model based on a different technology. Here, we will make a comparison between the model based on multiple linear regression and a model based on ANNs.

In multiple linear regression, the dependent variable (response) is related to two or more independent variables (predictors). The following equation is the general form of this model:

$$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \cdots + \beta_n * x_n$$

In the previous equation, x1, x2, ... xn are the predictors, and y is the response variable. The βi coefficients define the change in the response of the model related to the changes that occurred in xi, when the other variables remain constant. In the simple linear regression model, we are looking for a straight line that best fits the data. In the multiple linear regression model, we are looking for the plane that best fits the data. So, in the latter our aim is to minimize the overall squared distance between this plane and the response variable.

To estimate the coefficients β, we want to minimize the following term:

$$\sum_i [y_i - (\beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \cdots + \beta_n * x_n)]^2$$

To execute a multiple linear regression study, we can easily use the sklearn library. The `sklearn.linear_model` module is a module that contains several functions to resolve linear problems as a `LinearRegression` class that achieves an ordinary least squares linear regression:

1. To start, we will import the function as follows:

```
from sklearn.linear_model import LinearRegression
```

   Then, we set the model using the `LinearRegression()` function with the following command:

```
Lmodel = LinearRegression()
```

2. Now, we can use the `fit()` function to fit the model:

   ```
   Lmodel.fit(X_train, Y_train)
   ```

   The following parameters are passed:

   `X_train`: The training data.

   `Y_train`: The target data.

   Eventually, a third parameter can be passed; this is the `sample_weight` parameter, which contains the individual weights for each sample.

   This function fits a linear model using a series of coefficients to minimize the residual sum of squares between the expected targets and the predicted targets.

3. Finally, we can use the linear model to predict the new values using the predictors contained in the test dataset:

   ```
   Y_predLM = Lmodel.predict(X_test)
   ```

   At this point, we have the predictions.

   Now, we must carry out a first evaluation of the model to verify how much the prediction approached the expected value.

   There are several descriptors for evaluating a prediction model. In this example, we will use the **mean squared error** (**MSE**).

Important note

MSE returns the average of the squares of the errors. This is the average squared difference between the expected values and the value that is predicted. MSE returns a measure of the quality of an estimator; this is a non-negative value and, the closer the values are to zero, the better the prediction.

4. To calculate the MSE, we will use the `mean_squared_error()` function contained in the `sklearn.metrics` module. This module contains score functions, performance metrics and pairwise metrics, and distance computations. We start by importing the function, as follows:

   ```
   from sklearn.metrics import mean_squared_error
   ```

5. Then, we can apply the function to the data:

```
MseLM = mean_squared_error(Y_test, Y_predLM)
print('MSE of Linear Regression Model')
print(MseLM)
```

6. Two parameters were passed: the expected values (`Y_test`) and the values that were predicted (`Y_predLM`). Then, we print the results, as follows:

```
MSE of Linear Regression Model
0.015826467113949756
```

   The value obtained is low and very close to zero. However, for now, we cannot add anything. We will use this value, later on, to compare it with the value that we calculate for the model based on neural networks.

Understanding a multilayer perceptron regressor model

A multilayer perceptron contains at least three layers of nodes: input nodes, hidden nodes, and output nodes. Apart from the input nodes, each node is a neuron that uses a nonlinear activation function. A multilayer perceptron works with a supervised learning technique and backpropagation method for training the network. The presence of multiple layers and nonlinearity distinguishes a multilayer perceptron from a simple perceptron. A multilayer perceptron is applied when data cannot be separated linearly:

1. To build a multilayer perceptron-based model, we will use the sklearn `MLPRegressor` function. This regressor proceeds iteratively in the data training. At each step, it calculates the partial derivatives of the loss function with respect to the model parameters and uses the results obtained to update the parameters. There is a regularization term added to the loss function to reduce the model parameters to avoid data overfitting.

   First, we will import the function:

```
from sklearn.neural_network import MLPRegressor
```

   The `MLPRegressor()` function implements a multilayer perceptron regressor. This model optimizes the squared loss by using a limited-memory version of the **Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) algorithm or stochastic gradient descent algorithm.

2. Now, we can set the model using the `MLPRegressor` function, as follows:

   ```
   MLPRegModel = MLPRegressor(hidden_layer_sizes=(50),
                activation='relu', solver='lbfgs',
                tol=1e-4, max_iter=10000, random_state=0)
   ```

   The following parameters are passed:

   `hidden_layer_sizes=(50)`: This parameter sets the number of neurons in the hidden layer; the default value is 100.

   `activation='relu'`: This parameter sets the activation function. The following activation functions are available: identity, logistic, tanh, and ReLU. The last one is set by default.

   `solver='lbfgs'`: This parameter sets the solver algorithm for weight optimization. The following solver algorithms are available: LBFGS, **stochastic gradient descent** (**SGD**), and the SGD optimizer (`adam`).

   `tol=1e-4`: This parameter sets the tolerance for optimization. By default, `tol` is equal to `1e-4`.

   `max_iter=10000`: This parameter sets the maximum number of iterations. The solver algorithm iterates until convergence is imposed by tolerance or by this number of iterations.

   `random_state=1`: This parameter sets the seed used by the random number generator. In this way, it will be possible to reproduce the same model and obtain the same results.

3. After setting the parameters, we can use the data to train our model:

   ```
   MLPRegModel.fit(X_train, Y_train)
   ```

   The `fit()` function fits the model using the training data for predictors (`X_train`) and the response (`Y_train`). Finally, we can use the model trained to predict new values:

   ```
   Y_predMLPReg = MLPRegModel.predict(X_test)
   ```

   In this case, the test dataset (`X_test`) was used.

4. Now, we will evaluate the performance of the MLP model using the MSE metric, as follows:

   ```
   MseMLP = mean_squared_error(Y_test, Y_predMLPReg)
   print(' MSE of the SKLearn Neural Network Model')
   print(MseMLP)
   ```

   The following result is returned:

   ```
   MSE of the SKLearn Neural Network Model
   0.003315706807624097
   ```

   At this point, we can make an initial comparison between the two models that we have set up: the multiple linear regression-based model and the ANN-based model. We will do this by comparing the results obtained by evaluating the MSE for the two models.

5. We obtained an MSE of 0.0158 for the multiple linear regression-based model, and an MSE of 0.0033 for the ANN-based model. The last one returns a smaller MSE than the first by an order of magnitude, confirming the prediction that we made where neural networks return values much closer to the expected values.

   Finally, we make the same comparison between the two models; however, this time, we adopt a visual approach. We will draw two scatter plots in which we will report on the two axes the actual values (expected) and the predicted values, respectively:

   ```
   # SKLearn Neural Network diagram
   plt.figure(1)
   plt.subplot(121)
   plt.scatter(Y_test, Y_predMLPReg)
   plt.plot((0, 1), "r--")
   plt.xlabel("Actual values")
   plt.ylabel("Predicted values")
   plt.title("SKLearn Neural Network Model")

   # SKLearn Linear Regression diagram
   plt.subplot(122)
   plt.scatter(Y_test, Y_predLM)
   plt.plot((0, 1), "r--")
   plt.xlabel("Actual values")
   plt.ylabel("Predicted values")
   ```

```
plt.title("SKLearn Linear Regression Model")
plt.show()
```

By reporting the actual and expected values on the two axes, it is possible to check how this data is arranged. To help with the analysis, it is possible to trace the bisector of the quadrant, that is, the line of the equation, Y = X.

The following diagrams are printed:

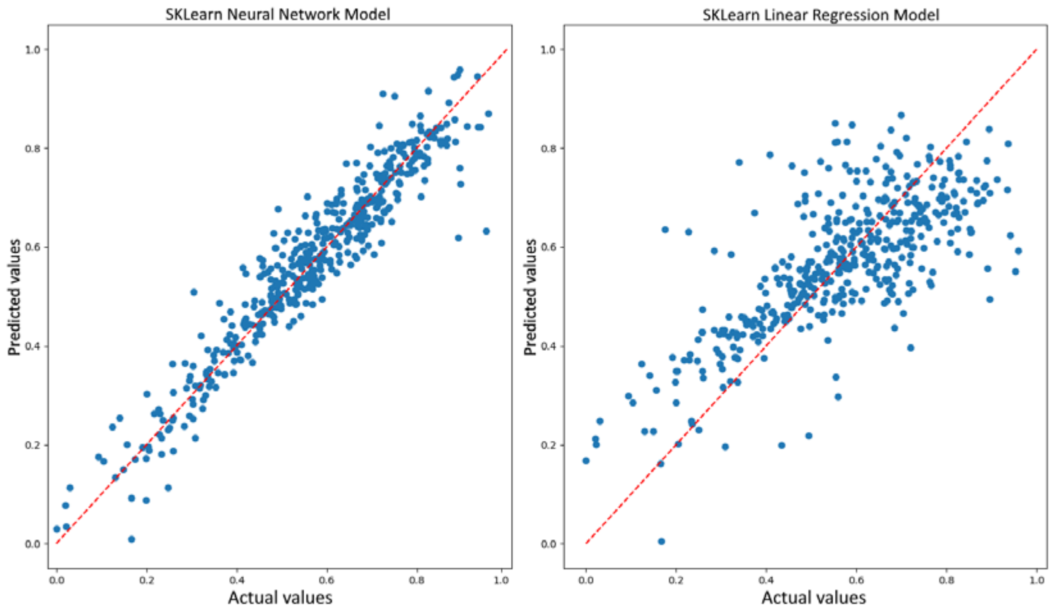


Figure 9.13 – Scatterplots of the neural network models

Hypothetically, all observations should be positioned exactly on the bisector line (the dotted line in the diagram), but we can be satisfied when the data is close to this line. About half of the data points must be below the line and the other half must be above the line. Points that move significantly away from this line represent possible outliers.

Analyzing the results reported in the previous diagram, we can see that, in the graph related to the ANN-based model, the points are much closer to the dotted line. This confirms the idea that this model returns better predictions than the multiple linear regression-based model.

Exploring deep neural networks

Deep learning is defined as a class of machine learning algorithms with certain characteristics. These models use multiple, hidden, nonlinear cascade layers to perform feature extraction and transformation jobs. Each level takes in the outputs from the previous level. These algorithms can be supervised, to deal with classification problems, or unsupervised, to deal with pattern analysis. The latter is based on multiple hierarchical layers of data characteristics and representations. In this way, the features of the higher layers are obtained from those of the lower layers, thus forming a hierarchy. Moreover, they learn multiple levels of representation corresponding to various levels of abstraction until they form a hierarchy of concepts.

The composition of each layer depends on the problem that needs to be solved. Deep learning techniques mainly adopt multiple hidden levels of an ANN but also sets of propositional formulas. The ANNs adopted have at least 2 hidden layers, but the applications of deep learning contain many more layers, for example, 10 or 20 hidden levels.

The development of deep learning in this period certainly depended on the exponential increase in data, with the consequent introduction of big data. In fact, with this exponential increase in data, there has been an increase in performance due to the increase in the level of learning, especially with respect to algorithms that already exist. In addition to this, the increase in computer performance also contributed to the improvement of obtainable results and to the considerable reduction in calculation times. There are several models based on deep learning. In the following sections, we will analyze the most popular ones.

Getting familiar with convolutional neural networks

A consequence of the application of the deep learning algorithms to ANNs is the development of a new model that is much more complex but with amazing results, that is, the **convolutional neural network** (**CNN**).

A CNN is a particular type of artificial feedforward neural network in which the connectivity pattern between neurons is inspired by the organization of the visual cortex of the human eye. Here, individual neurons are arranged in such a way as to devote themselves to the various regions that make up the visual field as a whole.

The hidden layers of this network are classified into various types: convolutional, pooling, ReLU, fully connected, and loss layers, depending on the role played. In the following sections, we will analyze them in detail.

Convolutional layers

This is the main layer of this model. It consists of a set of learning filters with a limited field of vision but extended along the entire surface of the input. Here, there are convolutions of each filter along the surface dimensions, making the scalar products between the filter inputs and the input image. Therefore, this generates a two-dimensional activation function that is activated if the network recognizes a certain pattern.

Pooling layers

In this layer, there is a nonlinear decimation that partitions the input image into a set of non-overlapping rectangles whose values are determined according to the nonlinear function. For example, with max pooling, the maximum of a certain value is identified for each region. The idea behind this layer is that the exact position of a feature is less important than its position compared to the others; therefore, superfluous information is omitted, also avoiding overfitting.

ReLU layers

The ReLU layer allows you to linearize the two-dimensional activation function and to set all negative values to zero.

Fully connected layers

This layer is generally placed at the end of the structure. It allows you to carry out the high-level reasoning of the neural network. It is called "fully connected" because the neurons in this layer have all been completely connected to the previous level.

Loss layers

This layer specifies how much the training penalizes the deviation between the predictions and the true values in output; therefore, it is always found at the end of the structure.

Examining recurrent neural networks

One of the tasks considered standard for a human, but of great difficulty for a machine, is the understanding of a piece of text. Given an ordered set of words, how can a machine be taught to understand its meaning? It is evident that, in this task, there is a more subtle relationship between the input data than in other cases. In the case of the process of classifying the content of an image, the whole image is processed by the machine simultaneously. This does not make sense in the elaboration of a piece of text, since the meaning of the words does not depend only on the words themselves, but also on their context.

Therefore, to understand a piece of text, it is not enough to know the set of words that are needed in it, but it is necessary to relate them to respect the order in which they are read. It is necessary to consider, and subsequently remember, the temporal context.

Recurrent neural networks essentially allow us to remember data that could be significant during the process we want to study.

This depends on the propagation rule that is used. To understand the functioning of this type of propagation, and of memory, we can consider a case of a recurrent neural network: the Elman network.

An Elman network is very similar to a hidden single-layer feedforward neural network, but a set of neurons called context units is added to the hidden layer. For each neuron present in the hidden layer, a context unit is added, which receives, as input, the output of the corresponding hidden neuron and returns its output to the same hidden neuron.

A type of network very similar to that of Elman is that of Jordan, in which the context units save the states of the output neurons instead of those of the hidden neurons. The idea behind it, however, is the same as Elman, and it is the same as many other recurrent neural networks that are based on the following principle: receiving a sequence of data as input and processing a new sequence of output data, which is obtained by subsequently recalculating the data of the same neurons.

The recurrent neural networks that are based on this principle are manifold, and the individual topologies are chosen to face different problems. For example, if it is not enough to remember the previous state of the network, but information processed many steps before may be necessary, **Long Short-Term Memory** (**LSTM**) neural networks can be used.

Analyzing LSTM networks

LSTM is a special architecture of recurrent neural networks. These models are particularly suited to the context of deep learning because they offer excellent results and performance.

LSTM-based models are perfect for prediction and classification in the time series field, and they are replacing several traditional machine learning approaches. This is because LSTM networks can account for long-term dependencies between data. For example, this allows us to keep track of the context within a sentence to improve speech recognition.

An LSTM-based model contains cells, named LSTM blocks, that are linked together. Each cell provides three types of ports: the input gate, the output gate, and the forget gate. These ports execute the write, read, and reset functions, respectively, on the cell memory. These ports are analogical and are controlled by a sigmoid activation function in the range of [0, 1]. Here, 0 means total inhibition, and 1 means total activation. The ports allow the LSTM cells to remember information for an unspecified amount of time.

Therefore, if the input port reads a value below the activation threshold, the cell will maintain the previous state, whereas if the value is above the activation threshold, the current state will be combined with the input value. The forget gate restores the current state of the cell when its value is zero, while the exit gate decides whether the value inside the cell should be removed or not.

Summary

In this chapter, we learned how to develop models based on ANNs to simulate physical phenomena. We started by analyzing the basic concepts of neural networks and the principles they are based on that are derived from biological neurons. We examined, in detail, the architecture of an ANN, understanding the concepts of weights, bias, layers, and the activation function.

Subsequently, we analyzed the architecture of a feedforward neural network. We saw how the training of the network with data takes place, and we understood the weight adjustment procedure that leads the network to correctly recognize new observations.

Next, we applied the concepts learned by tackling a practical case. We developed a model based on neural networks to solve a regression problem. We learned how to scale data and then how to subset the data for training and testing. We learned how to develop a model based on linear and MLP regression and how to evaluate the performance of these models to make a comparison.

Finally, we explored deep neural networks. We defined them by analyzing their basic concepts. We analyzed the basics of CNNs, recurrent neural networks, and LSTM networks.

In the next chapter, we will explore other practical model simulation applications. We will focus on simulation models in the field of project management.

10 Modeling and Simulation for Project Management

Sometimes, monitoring resources, budgets, and milestones for various projects and divisions can present a challenge. Simulation tools help us improve planning and coordination in the various phases of the project so that we always keep control of it. In addition, the preventive simulation of a project can highlight the critical issues related to a specific task. This helps us evaluate the cost of any actions to be taken. Through the preventive evaluation of the development of a project, errors that increase the costs of a project can be avoided.

In this chapter, we will deal with practical cases of project management using the tools we learned about in the previous chapters. We will learn how to evaluate the results of the actions we take when managing a forest using Markov processes, and then move on and learn how to evaluate a project using the Monte Carlo simulation.

In this chapter, we're going to cover the following main topics:

- Introducing project management
- Managing a tiny forest problem
- Scheduling project time using the Monte Carlo simulation

Technical requirements

In this chapter, we will address modeling examples of project management. To deal with these topics, it is necessary that you have a basic knowledge of algebra and mathematical modeling.

To work with the Python code in this chapter, you'll need the following files (available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Simulation-Modeling-with-Python`):

- `TinyForestManagement.py`
- `TinyForestManagementModified.py`
- `MonteCarloTasksScheduling.py`

Introducing project management

To assess the consequences of a strategic or tactical move in advance, companies need reliable predictive systems. Predictive analysis systems are based on data collection and the projection of reliable scenarios in the medium- and long-term. In this way, we can provide indications and guidelines for complex strategies, especially those that must consider numerous factors from different entities.

This allows us to examine the results of the evaluation in a more complete and coordinated way since we can simultaneously consider a range of values and, consequently, a range of possible scenarios. Finally, when managing complex projects, the use of artificial intelligence to interpret data has increased, thus giving these projects meaning. This is because we can perform a sophisticated analysis of the information in order to improve the strategic decision-making process we will undertake. This methodology allows us to search and analyze data from different sources so that we can identify patterns and relationships that may be relevant.

Understanding what-if analysis

What-if analysis is a type of analysis that can contribute significantly to making managerial decisions more effective, safe, and informed. It is also the basic level of predictive analysis based on data. What-if analysis is a tool capable of elaborating different scenarios to offer different possible outcomes. Unlike advanced predictive analysis, what-if analysis has the advantage of only requiring basic data to be processed.

This type of activity falls into the category of predictive analytics, that is, those that produce forecasts for the future, starting from a historical basis or trends. By varying some parameters, it is possible to simulate different scenarios and, therefore, understand what impact a given choice would have on costs, revenues, profits, and so on.

It is therefore a structured method to determine which predictions related to strategy changes can go wrong, thereby judging the probability and consequences of the studies carried out before they happen. Through the analysis of historical data, it is possible to create such predictive systems capable of estimating future results following the assumptions that were made about a group of variables of independent inputs, thus allowing us to formulate some forecasting scenarios with the aim of evaluating the behavior of a real system.

Analyzing the scenario at hand allows us to determine the expected values related to a management project. These analysis scenarios can be applied in different ways, the most typical of which is to perform multi-factor analysis, that is, analyze models containing multiple variables:

- Realization of a fixed number of scenarios by determining the maximum and minimum difference and creating intermediate scenarios through risk analysis. Risk analysis aims to determine the probability that a future result will be different from the average expected result. To show this possible variation, an estimate of the less likely positive and negative results is performed.

Random factorial analysis through the use of Monte Carlo methods, thus solving a problem by generating appropriate random numbers and observing that fraction of the numbers that obeys one or more properties. These methods are useful for obtaining numerical solutions for problems that are too complicated to solve analytically.

Managing a tiny forest problem

As we mentioned in *Chapter 5, Simulation-Based Markov Decision Processes*, a stochastic process is called **Markovian** if it starts from an instant *t* in which an observation of the system is made. The evolution of this process will depend only on *t*, so it will not be influenced by the previous instants. So, a process is called Markovian when the future evolution of the process depends only on the instant of observing the system and does not depend in any way on the past. MDP is characterized by five elements: decision epochs, states, actions, transition probability, and reward.

Summarizing the Markov decision process

The crucial elements of a Markovian process are the states in which the system finds itself, and the available actions that the decision maker can carry out on that state. These elements identify two sets: the set of states in which the system can be found, and the set of actions available for each specific state. The action chosen by the decision maker determines a random response from the system, which ultimately brings it into a new state. This transition returns a reward that the decision maker can use to evaluate the goodness of their choice.

Important Note

In a Markovian process, the decision maker has the option of choosing which action to perform in each system state. The action chosen takes the system to the next state and the reward for that choice is returned. The transition from one state to another enjoys the property of Markov: the current state depends only on the previous one.

A Markov process is defined by four elements, as follows:

- *S*: System states.
- *A*: Actions available for each state.
- *P*: Transition matrix. This contains the probabilities that an action *a* takes the system from *s* state to *s′* state.
- *R*: Rewards obtained in the transition from *s* state to *s′* state with an action *a*.

In an MDP problem, it becomes crucial to take actions to obtain the maximum reward from the system. Therefore, this is an optimization problem in which the sequence of choices that the decision maker will have to make is called an optimal policy.

A policy maps both the states of the environment and the actions to be chosen to those states, representing a set of rules or associations that respond to a stimulus. The policy's goal is to maximize the total reward received through the entire sequence of actions performed by the system. The total reward that's obtained by adopting a policy is calculated as follows:

$$R_T = \sum_{i=0}^{T} r_{t+1} = r_t + r_{t+1} + \cdots + r_T$$

In the previous equation, r_T is the reward of the action that brings the environment into the terminal state s_T. To get the maximum total reward, we can select the action that provides the highest reward to each individual state. This leads to choosing the optimal policy that maximizes the total reward.

Exploring the optimization process

As we mentioned in *Chapter 5, Simulation-Based Markov Decision Processes*, an MDP problem can be addressed using **dynamic programming** (**DP**). DP is a programming technique that aims to calculate an optimal policy based on a knowing model of the environment. The core of DP is to utilize the state-value and action-value in order to identify good policies.

In DP methods, two processes called policy evaluation and policy improvement are used. These processes interact with each other, as follows:

- Policy evaluation is done through an iterative process that seeks to solve Bellman's equation. The convergence of the process for k → ∞ imposes approximation rules, thus introducing a stop condition.
- Policy improvement improves the current policy based on the current values.

In the DP technique, the previous phases alternate and end before the other begins via an iteration procedure. This procedure requires a policy evaluation at each step, which it done through an iterative method whose convergence is not known a priori and depends on the starting policy; that is, we can stop evaluating the policy at some point, while still ensuring convergence to an optimal value.

Important Note

The iterative procedure we have described uses two vectors that preserve the results obtained from the policy evaluation and policy improvement processes. We indicate the vector that will contain the value function with *V*; that is, the discounted sum of the rewards obtained. We indicate the carrier that will contain the actions chosen to obtain those rewards with *Policy*.

The algorithm then, through a recursive procedure, updates these two vectors. In the policy evaluation, the value function is updated as follows:

$$V(s) = \sum_{s'} P_{Policy(s)}(s, s') \left(R_{Policy(s)}(s, s') + \gamma * V(s') \right)$$

In the previous equation, we have the following:

- $V(s)$ is the function value at the state *s*.
- $R(s, s')$ is the reward returned in the transition from state *s* to state *s'*.
- γ is the discount factor.
- $V(s')$ is the function value at the next state.

In the policy improvement process, the policy is updated as follows:

$$V(s)$$

In the previous equation, we have the following:

- $V(s')$ is the function value at state *s'*.
- $R_a(s, s')$ is the reward returned in the transition from state *s* to state *s'* with action *a*.
- γ is the discount factor.
- $P_a(s, s')$ is the probability that an action *a* in the *s* state is carried out in the *s'* state.

Now, let's see what tools we have available to deal with MDP problems in Python.

Introducing MDPtoolbox

The MDPtoolbox package contains several functions connected to the resolution of discrete-time Markov decision processes, that is, value iteration, finite horizon, policy iteration, linear programming algorithms with some variants, and several functions we can use to perform reinforcement learning analysis.

This toolbox was created by researchers from the Applied Mathematics and Computer Science Unit of INRA Toulouse (France), in the Matlab environment. The toolbox was presented by the authors in the following article: Chadès, I., Chapron, G., Cros, M. J., Garcia, F., & Sabbadin, R. (2014). *MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems. Ecography*, 37 (9), 916-920.

Important Note

The MDPtoolbox package was subsequently made available in other programming platforms, including GNU Octave, Scilab, and R. It was later made available for Python programmers by S. Cordwell. You can find out more at the following URL: https://github.com/sawcordwell/pymdptoolbox.

To use the MDPtoolbox package, we need to install it. The different installation procedures are indicated on the project's GitHub website. As recommended by the author, you can use the default Python pip package manager. **Pip** stands for **Python Package Index** and is the largest and most official Python package repository. Anyone who develops a Python package, in 99% of cases, makes it available on this repository.

To install the MDPtoolbox package using pip, just write the following command:

```
pip install pymdptoolbox
```

Once installed, just load the library to be able to use it immediately.

Defining the tiny forest management example

To analyze in detail how to deal with a management problem using Markovian processes, we will use an example already available in the MDPtoolbox package. It deals with managing a small forest in which there are two types of resources: wild fauna and trees. The trees of the forest can be cut, and the wood that's obtained can be sold. The decision maker has two actions: wait and cut. The first action is to wait for the tree to grow fully before cutting it to obtain more wood. The second action involves cutting the tree to get money immediately. The decision maker has the task of making their decision every 20 years.

The tiny forest environment can be in one of the following three states:

- **State 1**: Forest age 0-20 years
- **State 2**: Forest age 21-40 years
- **State 3**: Forest age over 40 years

We might think that the best action is to wait until we have the maximum amount of wood to come and thus obtain the greatest gain. Waiting can lead to the loss of all the wood available. This is because as the trees grow, there is also the danger that a fire could develop, which could cause the wood to be lost completely. In this case, the tiny forest would be returned to its initial state (state 1), so we would lose what we would have gained.

In the case a fire does not occur, at the end of each period *t* (20 years), if the state is *s* and the wait action is chosen, the forest will move to the next state, which will be the minimum of the following pair (s + 1, 3). If there are no fires, the age of the forest will never assume a state higher than 3 since state 3 matches with the oldest age class. Conversely, if a fire occurs after the action is applied, the forest returns the system to its initial state (state 1), as shown in the following image:

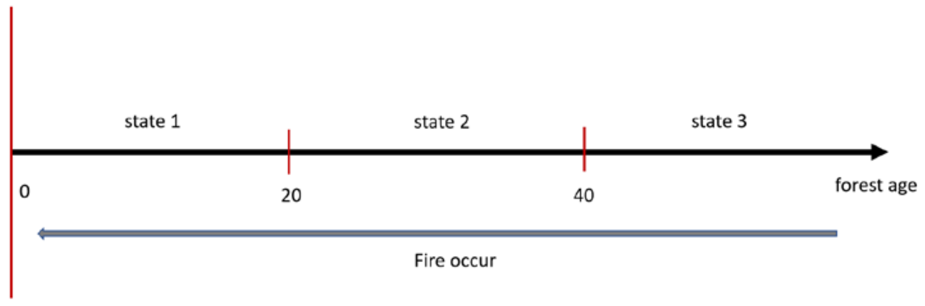


Figure 10.1 – States of the age of a forest

Set $p = 0.1$ as the probability that a fire occurs during a period t. The problem is how to manage this in the long-term to maximize the reward. This problem can be treated as a MDP.

Now, let's move on and define the problem as an MDP. We have said that the elements of an MDP are state, action, transition matrix *P*, and reward *R*. We must then define these elements. We have defined the states already – there are three. We also defined the actions: wait or cut. We pass these to define the transition matrix P (s, s', a). It contains the chances of the system going from one state to another. We have two actions available (Wait, Cut), so we will define two transition matrices. If we indicate with p the probability that a fire occurs, then in the case of the wait action, we will have the following transition matrix:

$$P(,,1) = \begin{bmatrix} p & 1-p & 0 \\ p & 0 & 1-p \\ p & 0 & 1-p \end{bmatrix}$$

Now, let's analyze the content of the transition matrix. Each row is relative to a state, in the sense that row 1 returns the probabilities that, starting from state 1, it will remain in state 1 or pass to state 2 or 3. In fact, if we are in state 1, we will have a probability p that we remain in that state, which happens if a fire occurs. Always starting from state 1, if no fire occurs, we have the remaining *1-p* probability of moving to the next state, which is state 2. From this, it is clear that when starting from state 1, the probability of passing to state 3 is equal to 0 – it's impossible to do so.

Row 2 of the transition matrix contains the transition probabilities starting from state 2. In fact, starting from state 2, if a fire occurs, there will be an equal probability p to pass into state 1. Always starting from state 2, if no fire occurs, we have the remaining *1-p* probability of moving to the next state, which is state 3. In this case, once again, the probability of remaining in state 2 is equal to 0.

Finally, if we are in state 3, if a fire occurs, we will have a probability equal to p of going to state 1, and the remaining *1-p* probability of remaining in state 3, which happens if no fire occurs. The probability of going to state 2 is equal to 0.

Now, let's define the transition matrix in the case of choosing the cut action:

$$P(,,2) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

In this case, the analysis of the previous transition matrix is much more immediate. In fact, the cut action brings the state of the system to 1 in each instance. Therefore, the probability is always 1. Then, that 1 goes to state 1 and 0 for all the other transitions as they are not possible.

Now, let's define the vectors that contain the rewards; that is, the vector R (s, s', as we have defined it), starting from the rewards returned by the wait action:

$$R(,1) = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix}$$

The action of waiting for the growth of the forest will bring a reward of 0 for the first two states, while the reward will be the maximum for state 3. The value of the reward in state 3 is equal to 4, which represents the value provided by the system by default. Let's see how the vector of rewards is modified if you choose the cut action:

$$R(,2) = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

In this case, cutting in state 1 does not bring any reward since the trees are not able to supply wood yet. The cut in state 2 brings a reward, but this is lower than the maximum reward, which we said is obtainable if we wait for the end of the three periods *t* before cutting. A similar situation arises if the cut is made at the beginning of the third period. In this case, the reward is greater than that of the previous state but still less than the maximum.

Addressing management problems using MDPtoolbox

Our goal is to develop a policy that allows us to manage the tiny forest in order to obtain the maximum prize. We will do this using the `MDPtoolbox` package, which we introduced in the previous section, and analyzing the code line by line:

1. Let's start as always by importing the necessary library:

   ```
   import mdptoolbox.example
   ```

   By doing this, we imported the `MDPtoolbox` module, which contains the data for this example.

2. To begin, we will extract the transition matrix and the reward vectors:

   ```
   P, R = mdptoolbox.example.forest()
   ```

   This command retrieves the data stored in the example. To confirm the data is correct, we print the content of these variables, starting from the transition matrix:

   ```
   print(P[0])
   ```

   The following matrix is printed:

   ```
   [[0.1 0.9 0. ]
    [0.1 0.  0.9]
    [0.1 0.  0.9]]
   ```

This is the transition matrix for the wait action. Consistent with what is indicated in the *Defining the tiny forest management example* section, having set $p = 0.1$, we can confirm the transition matrix.

Now, we print the transition matrix related to the cut action:

```
print(P[1])
```

The following matrix is printed:

```
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]]
```

This matrix is also consistent with what was previously defined. Now, let's check the shape of the reward vectors, starting with the wait action:

```
print(R[:,0])
```

Let's see the content:

```
[0. 0. 4.]
```

If the cut action is chosen, we will have the following rewards:

```
print(R[:,1])
```

The following vector is printed:

```
[0. 1. 2.]
```

Finally, let's fix the discount factor:

```
gamma=0.9
```

All the problem data has now been defined. We can now move on and look at the model in greater detail.

3. The time has come to apply the policy iteration algorithm to the problem we have just defined:

```
PolIterModel = mdptoolbox.mdp.PolicyIteration(P, R, gamma)
PolIterModel.run()
```

The `mdptoolbox.mdp.PolicyIteration()` function performs a discounted MDP that's solved using the policy iteration algorithm. Policy iteration is a dynamic programming method that adopts a value function in order to model the expected return for each pair of action-state. These techniques update the value functions using the immediate reward and the (discounted) value of the next state in a process called bootstrapping. The results are stored in tables or with approximate function techniques.

The function, starting from an initial *P0* policy, updated the function value and the policy through an iterative procedure, alternating the following two phases:

- **Policy evaluation**: Given the current policy *P*, estimate the action-value function.
- **Policy Improvement**: If we calculate a better policy based on the action-value function, then this policy is made the new policy and we return to the previous step.

When the value function can be calculated exactly for each action-state pair, the policy iteration we performed with the greedy policy improvement leads to convergence by returning the optimal policy. Essentially, repeatedly executing these two processes converges the general process toward the optimal solution.

In the `mdptoolbox.mdp.PolicyIteration()` function, we have passed the following arguments:

- **P**: Transition probability
- **R**: Reward
- **gamma**: Discount factor

The following results are returned:

- **V**: Optimal value function. *V* is an *S* length vector.
- **policy**: Optimal policy. The policy is an *S* length vector. Each element is an integer corresponding to an action that maximizes the value function. In this example, only two actions are foreseen: 0 = wait, 1 = cut.
- **iter**: Number of iterations.
- **time**: CPU time used to run the program.

Now that the model is ready, we must evaluate the results by checking the obtained policy.

To begin, we check the updates of the value function:

```
print(PolIterModel.V)
```

The following results are returned:

```
 (26.244000000000014, 29.484000000000016,
33.484000000000016)
```

A value function specifies how good for the system a state is. This value represents the total reward expected for a system from the status *s*. The value function depends on the policy that the agent selects for the actions to be performed on.

Let's move on and extract the policy:

```
print(PolIterModel.policy)
```

A policy suggests the behavior of the system at a given time. It maps the detected states of the environment and the actions to take when they are in those states. This corresponds to what, in psychology, would be called a set of rules or associations of the stimulus response. The policy is the crucial element of an MDP model since it defines the behavior. The following results are returned:

```
 (0, 0, 0)
```

Here, the optimal policy is to not cut the forest in all three states. This is due to the low probability of a fire occurring, which causes the wait action to be the best action to perform. In this way, the forest has time to grow and we can achieve both goals: maintain an old forest for wildlife and earn money by selling the cut wood.

Let's see how many iterations have been made:

```
print(PolIterModel.iter)
```

The following result is returned:

```
2
```

Finally, let's print the CPU time:

```
print(PolIterModel.time)
```

The following result is returned:

```
0.12830829620361328
```

Only 0.13 seconds is required to perform the value iteration procedure.

Changing the probability of fire

The analysis of the previous example has clarified how to derive an optimal policy from a well-posed problem. We can now define a new problem by changing the initial conditions of the system. Under the default conditions provided in our example, the probability of a fire occurring is low. In this case, we have seen that the optimal policy advises us to wait and not cut the forest. But what if we increase the probability of a fire occurring? This is a real-life situation; just think of warm places particularly subject to strong winds. To model this new condition, simply change the problem settings by changing the probability value *p*. The `mdptoolbox.example.forest()` module allows us to modify the basic characteristics of the problem. Let's get started:

1. Let's start by importing the example module:

```
import mdptoolbox.example
P, R = mdptoolbox.example.forest(3,4,2,0.8)
```

In contrast with the example discussed in the previous section, *Addressing management problems*, where we used `MDPtoolbox`, in this case, we have passed some parameters. Let's analyze them in detail. In the `mdptoolbox.example.forest ()` function, we passed the following parameters (3, 4, 2, 0.8). Let's analyze their meaning:

`3`: Number of states. This must be an integer greater than 0.

`4`: The reward when the forest is in the oldest state and the wait action is performed. This must be an integer greater than 0.

`2`: The reward when the forest is in the oldest state and the cut action is performed. This must be an integer greater than 0.

`0.8`: The probability of a fire occurring. This must be in]0, 1[.

By analyzing the past data, we can see that we confirmed the first three parameters, while we only changed the probability of a fire occurring, thus increasing this possibility from 0.1 to 0.8.

Let's see this change to the initial data as it changed the transition matrix:

```
print(P[0])
```

The following matrix is printed:

```
[[0.8 0.2 0. ]
 [0.8 0.  0.2]
 [0.8 0.  0.2]]
```

As we can see, the transition matrix linked to the wait action has changed. Now, the probability that the transition is in a state other than 1 has significantly decreased. This is due to the high probability that a fire will occur. Let's see what happens when the transition matrix is linked to the cut action:

```
print(P[1])
```

The following matrix is printed:

```
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]]
```

This matrix remains unchanged. This is due to the result of the cut action, which returns the system to its initial state. Likewise, reward vectors may be unchanged since the rewards that were passed are the same as the ones provided by the default problem. Let's print these values:

```
print(R[:,0])
```

This is the reward vector connected to the wait action. The following vector is printed:

```
[0. 0. 4.]
```

In the case of the cut action, the following vector is retuned:

```
print(R[:,1])
```

The following vector is printed:

```
[0. 1. 2.]
```

As anticipated, nothing has changed. Finally, let's fix the discount factor:

```
gamma=0.9
```

All the problem data has now been defined. We can now move on and look at the model in greater detail.

2. We will now apply the value iteration algorithm:

```
PolIterModel = mdptoolbox.mdp.PolicyIteration(P, R, 
gamma)
PolIterModel.run()
```

Now, we can extract the results, starting with the value function:

```
print(PolIterModel.V)
```

The following results are printed:

```
(1.5254237288135597, 2.3728813559322037,
6.217445225299711)
```

Let's now analyze the crucial part of the problem: let's see what policy the simulation model suggests to us:

```
print(PolIterModel.policy)
```

The following results are printed:

```
(0, 1, 0)
```

In this case, the changes we've made here, compared to the default example, are substantial. It's suggested that we adopt the wait action if we are in state 1 and 3, while if we are in state 2, it is advisable to try to cut the forest. Since the probability of a fire is high, it is convenient to cut the wood already available and sell it before a fire destroys it in full.

Then, we print the number of iterations of the problem:

```
print(PolIterModel.iter)
```

The following result is printed:

```
1
```

Finally, we print the CPU time:

```
print(PolIterModel.time)
```

The following result is printed:

```
0.14069104194641113
```

These examples have highlighted how simple the modeling procedure of a management problem is through using MDPs.

Scheduling project time using Monte Carlo simulation

Each project requires a time of realization, and the beginning of some activities can be independent or dependent on previous activities ending. Scheduling a project means determining the times of realization of the project itself. A project is a temporary effort undertaken to create a unique product, service, or result. The term **project management** refers to the application of knowledge, skills, tools, and techniques for the purpose of planning, managing, and controlling a project and the activities of which it is composed.

The key figure in this area is the project manager, who has the task and responsibility of coordinating and controlling the various components and actors involved, with the aim of reducing the probability of project failure. The main difficulty in this series of activities is to achieve the objectives set in compliance with constraints such as the scope of the project, time, costs, quality, and resources. In fact, these are limited aspects that are linked to each other and that need effective optimization.

The definition of these activities constitutes one of the key moments of the planning phase. After defining what the project objectives are with respect to time, cost, and resources, it is necessary to proceed with identifying and documenting the activities that must be carried out to successfully complete the project.

For complex projects, it is necessary to create an ordered structure by decomposing the project into simpler tasks. For each task, it will be necessary to define activities and execution times. This starts with the main objective and breaks down the project to the immediately lower level in all those deliverables or main sub-projects that make it up.

These will, in turn, be broken down. This will continue until you are satisfied with the degree of detail of the resulting final items. Each breakdown results in a reduction in the size, complexity, and cost of the interested party.

Defining the scheduling grid

A fundamental part of all project management is constructing the scheduling grid. This is an oriented graph that represents the temporal succession and the logical dependencies between the activities involved in the realization of the project. In addition to constructing the grid, the scheduling process also determines the start and end times of activities based on factors such as duration, resources, and so on.

In the example we are dealing with, we will take care of evaluating the times necessary for the realization of a complex project. Let's start by defining the scheduling grid. Suppose that, by decomposing the project structure, we have defined six tasks. For each task, the activities, the personnel involved, and the time needed to finish the job were defined.

Some tasks must be performed in a series, in the sense that the activities of the previous task must be completed so that they can start those of the next task. Others, however, can be performed in parallel, in the sense that two teams can simultaneously work on two different tasks to reduce project delivery times. This sequence of tasks is defined in the scheduling grid, as follows:

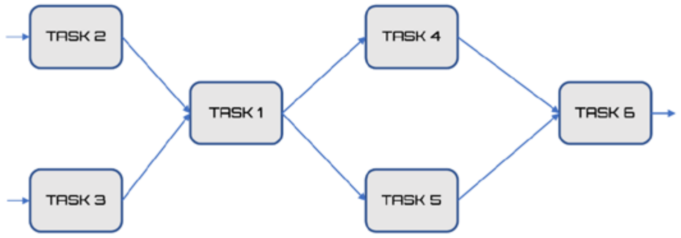


Figure 10.2 – Sequence of tasks in the grid

The preceding diagram shows us that the first two tasks develop in parallel, which means that the time required to finish these two tasks will be provided by the time-consuming task. The third task develops in a series, while the next two are, again, in parallel. The last task is still in the series. This sequence will be necessary when we evaluate the project times.

Estimating the task's time

The duration of these tasks is often difficult to estimate due to the number of factors that can influence it: the availability and/or productivity of the resources, the technical and physical constraints between the activities, and the contractual commitments.

Expert advice, supported by historical information, can be used wherever possible. The members of the project team will also be able to provide information on the duration or the maximum recommended limit for the duration for the task by deriving information from similar projects.

There are several ways we can estimate tasks. In this example, we will use three-point estimation. In three-point estimation, the accuracy of the duration of the activity estimate can be increased in terms of the amount of risk in the original estimate. Three-point estimates are based on determining the following three types of estimates:

- **Optimistic**: The duration of the activity is based on the best scenario in relation to what is described in the most probable estimate. This is the minimum time it will take to complete the task.
- **Pessimistic**: The duration of the activity is based on the worst-case scenario in relation to what is described in the most probable estimate. This is the maximum time that it will take to complete the task.

- **More likely**: The duration of the activity is based on realistic expectations in terms of availability for the planned activity.

A first estimate of the duration of the activity can be constructed using an average of the three estimated durations. This average typically provides a more accurate estimate of the duration of the activity than a more likely single value estimate. But that's not what we want to do.

Suppose that the project team used three-point estimation for each of the six tasks. The following table shows the times proposed by the team:

Task	Optimistic time	Pessimistic time	More likely time
Task 1	3	5	8
Task 2	2	4	7
Task 3	3	5	9
Task 4	4	6	10
Task 5	3	5	9
Task 6	2	6	8

After defining the sequence of tasks and the time it will take to perform each individual task, we can develop an algorithm for estimating the overall time of the project.

Developing an algorithm for project scheduling

In this section, we will analyze an algorithm for scheduling a project based on the Monte Carlo simulation. We will look at all the commands in detail, line by line:

1. Let's start by importing the libraries that we will be using in the algorithm:

```
import numpy as np
import random
import pandas as pd
```

   `numpy` is a Python library that contains numerous functions that help us manage multidimensional matrices. Furthermore, it contains a large collection of high-level mathematical functions we can use on these matrices.

   The `random` library implements pseudo-random number generators for various distributions. The `random` module is based on the Mersenne Twister algorithm.

   The `pandas` library is an open source BSD licensed library that contains data structures and operations that can be used to manipulate high-performance numeric values for the Python programming language.

2. Let's move on and initialize the parameters and the variables:

   ```
   N = 10000
   TotalTime=[]
   T =  np.empty(shape=(N,6))
   ```

   `N` represents the number of points that we generate. These are the random numbers that will help us define the time of that task. The `TotalTime` variable is a list that will contain the `N` assessments of the overall time needed to complete the project. Finally, the `T` variable is a matrix with `N` rows and six columns and will contain the `N` assessments of the time needed to complete each individual task.

3. Now, let's set the three-point estimation matrix, as defined in the table in the *Estimating the task's time* section:

   ```
   TaskTimes=[[3,5,8],
              [2,4,7],
              [3,5,9],
              [4,6,10],
              [3,5,9],
              [2,6,8]]
   ```

   This matrix contains the three times for each representative row of the six tasks: optimistic, more likely, and pessimistic.

At this point, we must establish the form of the distribution of times that we intend to adopt.

Exploring triangular distribution

When developing a simulation model, it is necessary to introduce probabilistic events. Often, the simulation process starts before you have enough information about the behavior of the input data. This forces us to decide on a distribution. Among those that apply to incomplete data is the triangular distribution. The triangular distribution is used when assumptions can be made on the minimum and maximum values and on the modal values.

Important Note

A probability distribution is a mathematical model that links the values of a variable to the probabilities that these values can be observed. Probability distributions are used for modeling the behavior of a phenomenon of interest in relation to the reference population, or to all the cases in which the investigator observes a given sample.

In *Chapter 3, Probability and Data Generating Processes*, we analyzed the most widely used probability distributions. When the random variable is defined in a certain range, but there are reasons to believe that the degrees of confidence decrease linearly from the center to the extremes, there is the so-called triangular distribution. This distribution is very useful for calculating measurement uncertainties since, in many circumstances, this type of model can be more realistic than the uniform one.

Let's consider the first task of the project we are analyzing. For this, we have defined the three times: optimistic (3), more likely (5), and pessimistic (8). We draw a graph in which we report these three times on the abscissa and the probability of their occurrence on the ordinate. Using the triangular probability distribution, the probabilities that the event occurs is between the limit values, which in our case are optimistic and pessimistic. We do this while assuming the maximum value in correspondence with the value more likely to occur. For the intermediate values, where we know nothing, suppose that the probability increases linearly from optimistic to more likely, and then always decreases linearly from more likely to pessimistic, as shown in the following graph:

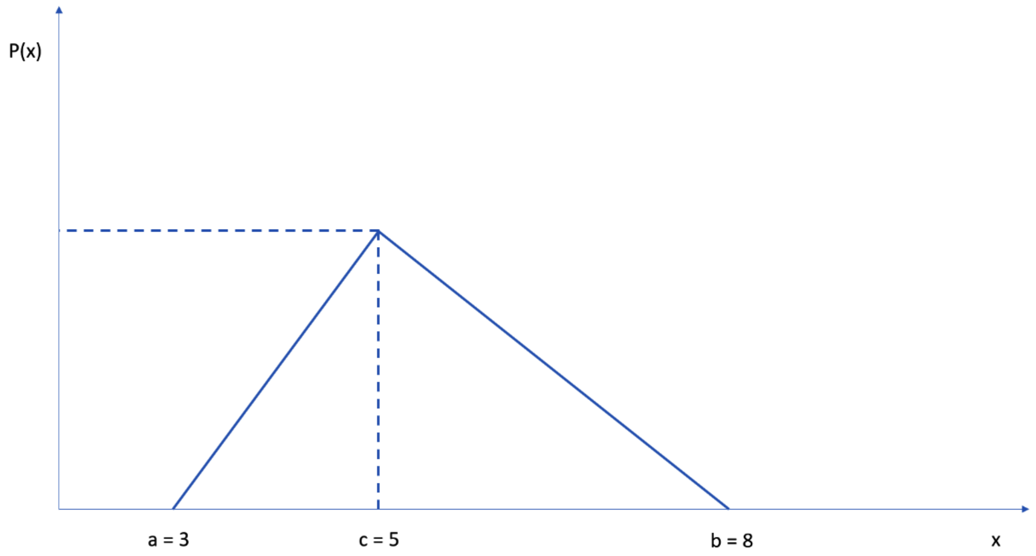


Figure 10.3 – Probability graph

Our goal is to model the times of each individual task using a random variable with uniform distribution in the interval (0, 1). If we indicate this random variable with `trand`, then the triangular distribution allows us to evaluate the probability that the task ends in that time is distributed. In the triangular distribution, we have identified two triangles that have the abscissa in common for x = c. This value acts as a separator between the two values that the distribution assumes. Let's denote it with `Lh`. It is given by the following formula:

$$Lh = \frac{(c - a)}{(b - a)}$$

In the previous equation, we have the following:

- *a*: Optimistic time
- *b*: Pessimistic time
- *c*: More likely time

That being said, we can generate variations according to the triangular distribution with the following equations:

$$T = \begin{cases} a + \sqrt{trand * (b - a) * (c - a)} & \forall\, a < trand < Lh \\ b - \sqrt{(1 - trand) * (b - a) * (b - c)} & \forall\, Lh \leq trand < 1 \end{cases}$$

The previous equations allow us to perform the Monte Carlo simulation. Let's see how:

1. First, we generate the separation value of the triangular distribution:

```
Lh=[]
for i in range(6):
    Lh.append((TaskTimes[i][1]-TaskTimes[i][0])
                 /(TaskTimes[i][2]-TaskTimes[i][0]))
```

   Here, we initialized a list and then populated it with a `for` loop that iterates over the six tasks, evaluating a value of `Lh` for each one.

   Now, we use two `for` loops and an `if` conditional structure to develop the Monte Carlo simulation:

```
for p in range(N):
    for i in range(6):
        trand=random.random()
```

```
          if (trand < Lh[i]):
              T[p][i] = TaskTimes[i][0] +
                     np.sqrt(trand*(TaskTimes[i][1]-
                     TaskTimes[i][0])*
                     (TaskTimes[i][2]-TaskTimes[i][0]))
          else:
              T[p][i] = TaskTimes[i][2] -
                     np.sqrt((1-trand)*(TaskTimes[i][2]-
                     TaskTimes[i][1])*
                    (TaskTimes[i][2]-TaskTimes[i][0]))
```

The first `for` loop continues generating the random values *N* times, while the second loop is used to perform the evaluation for six tasks. The conditional structure `if`, on the other hand, is used to discriminate between the two values distinct from the value of *Lh* so that it can use the two equations that we defined previously.

Finally, for each of the *N* iterations, we calculate an estimate of the total time for the execution of the project:

```
  TotalTime.append( T[p][0]+
                 np.maximum(T[p][1],T[p][2]) +
                 np.maximum(T[p][3],T[p][4]) + T[p][5])
```

For the calculation of the total time, we referred to the scheduling grid defined in the *Defining the scheduling grid* section. The procedure is simple: if the tasks develop in a series, then you add the times up, while if they develop in parallel, you choose the maximum value among the times of the tasks.

2. Now, let's take a look at the values we have attained:

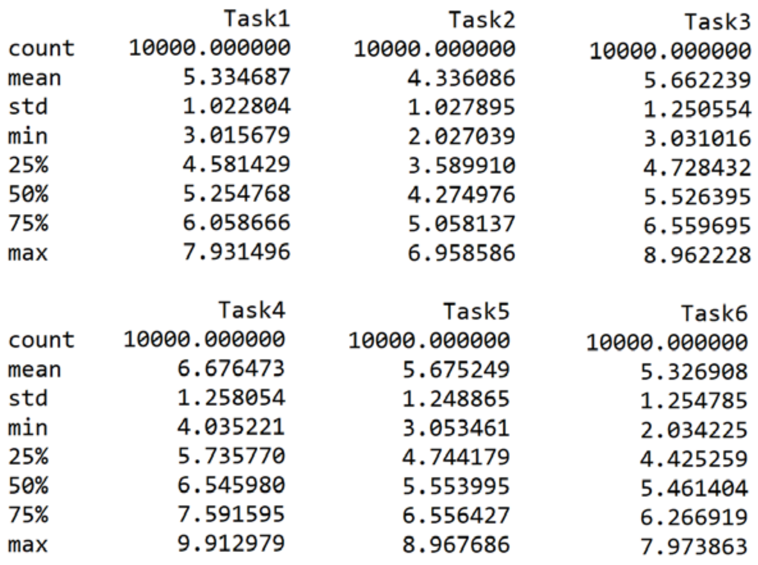
```
  Data = pd.DataFrame(T,columns=['Task1', 'Task2', 'Task3',
                              'Task4', 'Task5', 'Task6'])
  pd.set_option('display.max_columns', None)
  print(Data.describe())
```

For detailed statistics of the times estimated with the Monte Carlo method, we have transformed the matrix (*Nx6*) containing the times into a pandas DataFrame. The reason for this is that the `pandas` library has useful functions that allow us to extract detailed statistics from a dataset immediately. In fact, we can do this with just a line of code by using the `describe()` function.

The `describe()` function generates a series of descriptive statistics that return useful information on the dispersion and the form of the distribution of a dataset.

The pandas `set.option()` function was used to display all the statistics of the matrix and not just a part of it, as expected by default.

The following results are returned:



```
              Task1         Task2         Task3
count  10000.000000  10000.000000  10000.000000
mean       5.334687      4.336086      5.662239
std        1.022804      1.027895      1.250554
min        3.015679      2.027039      3.031016
25%        4.581429      3.589910      4.728432
50%        5.254768      4.274976      5.526395
75%        6.058666      5.058137      6.559695
max        7.931496      6.958586      8.962228

              Task4         Task5         Task6
count  10000.000000  10000.000000  10000.000000
mean       6.676473      5.675249      5.326908
std        1.258054      1.248865      1.254785
min        4.035221      3.053461      2.034225
25%        5.735770      4.744179      4.425259
50%        6.545980      5.553995      5.461404
75%        7.591595      6.556427      6.266919
max        9.912979      8.967686      7.973863
```

Figure 10.4 – Values of the DataFrame

By analyzing these statistics, we have confirmed that the estimated times are between the limit values imposed by the problem: optimistic and pessimistic. In fact, the minimum and maximum times are very close to these values. Furthermore, we can see that the standard deviation is very close to the unit. Finally, we can confirm that we have generated 10,000 values.

We can now trace the histograms of the distribution of the values of the times to analyze their form:

```
hist = Data.hist(bins=10)
```

The following diagram is printed:

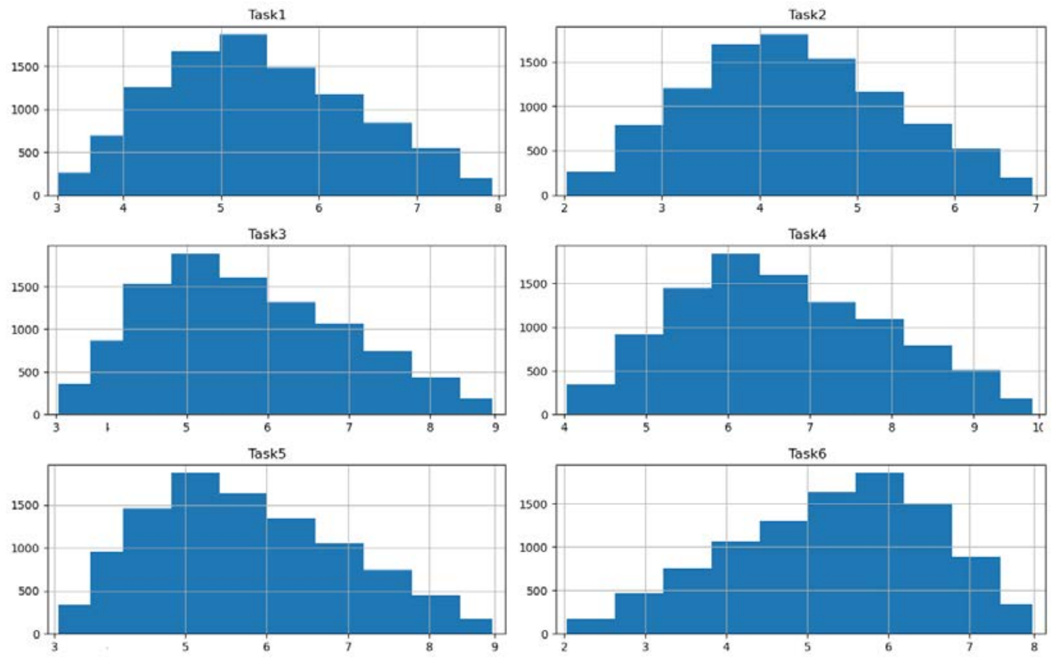


Figure 10.5 – Histograms of the values

By analyzing the previous diagram, we can confirm the triangular distribution of the time estimates, as we had imposed at the beginning of the calculations.

3. At this point, we only need to print the statistics of the total times. Let's start with the minimum value:

```
print("Minimum project completion time = ",
                                np.amin(TotalTime))
```

The following result is returned:

```
Minimum project completion time =  14.966486785163458
```

Let's analyze the average value:

```
print("Mean project completion time = ",np.
mean(TotalTime))
```

The following result is returned:

```
Mean project completion time =  23.503585938922157
```

Finally, we will print the maximum value:

```
print("Maximum project completion time = ",np.
amax(TotalTime))
```

The following result is printed:

```
Maximum project completion time =  31.90064194829465
```

In this way, we obtained an estimate of the time needed to complete the project based on the Monte Carlo simulation.

Summary

In this chapter, we addressed several practical model simulation applications based on project management-related models. To start, we looked at the essential elements of project management and how these factors can be simulated to retrieve useful information.

Next, we tackled the problem of running a tiny forest for the wood trade. We treated the problem as an MDP, summarizing the basic characteristics of these processes and then moved on to a practical discussion of them. We defined the elements of the problem and then we saw how to use the policy evaluation and policy improvement algorithms to obtain the optimal forest management policy. This problem was addressed using the `MDPtoolbox` package, which is available from Python.

Subsequently, we addressed the problem of evaluating the execution times of a project using Monte Carlo simulation. To start, we defined the task execution diagram by specifying which tasks are performed in series and which are performed in parallel. So, we introduced the times of each task through three-point estimation. After this, we saw how to model the execution times of the project with triangular distribution using random evaluations of each phase. Finally, we performed 10,000 assessments of the overall project times.

In the next chapter, we will summarize the simulation modeling processes we looked at in the previous chapters. Then, we will explore the main simulation modeling applications that are used in real life. Finally, we will discover future challenges regarding simulation modeling.

11
What's Next?

In this chapter, we will summarize what has been covered so far in this book and what the next steps are from this point on. You will learn how to apply all the skills that you have gained to other projects, as well as the real-life challenges in building and deploying simulation models and other common technologies that data scientists use. By the end of this chapter, you will have a better understanding of the issues associated with building and deploying simulating models and additional resources and technologies you can learn about to sharpen your machine learning skills.

In this chapter, we're going to cover the following main topics:

- Summarizing simulation modeling concepts
- Applying simulation models to real life
- Next steps for simulation modeling

Summarizing simulation modeling concepts

Useful in cases where it is not possible to develop a mathematical model capable of effectively representing a phenomenon, simulation models imitate the operations performed over time by a real process. The simulation process involves generating an artificial history of the system to be analyzed; subsequently, the observation of this artificial history is used to trace information regarding the operating characteristics of the system itself and make decisions based on it.

The use of simulation models as a tool to aid decision-making processes has ancient roots and is widespread in various fields. Simulation models are used to study the behavior of a system over time, and are built based on a set of assumptions made on the behavior of the system that's expressed using mathematical-logical-symbolic relationships. These relationships are between the various entities that make up the system. The purpose of a model is to simulate changes in the system and predict the effects of these changes on the real system. For example, they can be used in the design phase before the model's actual construction.

Important Note

Simple models are resolved analytically, using mathematical methods. The solution consists of one or more parameters, called **behavior measures**. Complex models are simulated numerically on the computer, where the data is treated as being derived from a real system.

Let's summarize the tools we have available to develop a simulation model.

Generating random numbers

In simulation models, the quality of the final application strictly depends on the possibility of generating good quality random numbers. In several algorithms, decisions are made based on a randomly chosen value. The definition of random numbers includes that of random processes through a connection that specifies its characteristics. A random number appears as such because we do not know how it was generated, but once the law within which it was generated is defined, we can reproduce it whenever we want.

Deterministic algorithms do not allow us to generate random number sequences, but simply make pseudo-random sequence generation possible. Pseudo-random sequences differ from random ones in the strict sense in that they are reproducible and therefore predictable.

Multiple algorithms are available for generating pseudo-random numbers. In *Chapter 2, Understanding Randomness and Random Numbers*, we analyzed the following in detail:

- **Linear Congruential Generator** (**LCG**): This generates a sequence of pseudo-randomized numbers using a piecewise discontinuous linear equation.
- **Lagged Fibonacci Generator** (**LFG**): This is based on a generalization of the Fibonacci sequence.

More specific methods are added to these ones to generate uniform distributions of random numbers. The following graph shows a uniform distribution of 1,000 random numbers in the range 1-10:

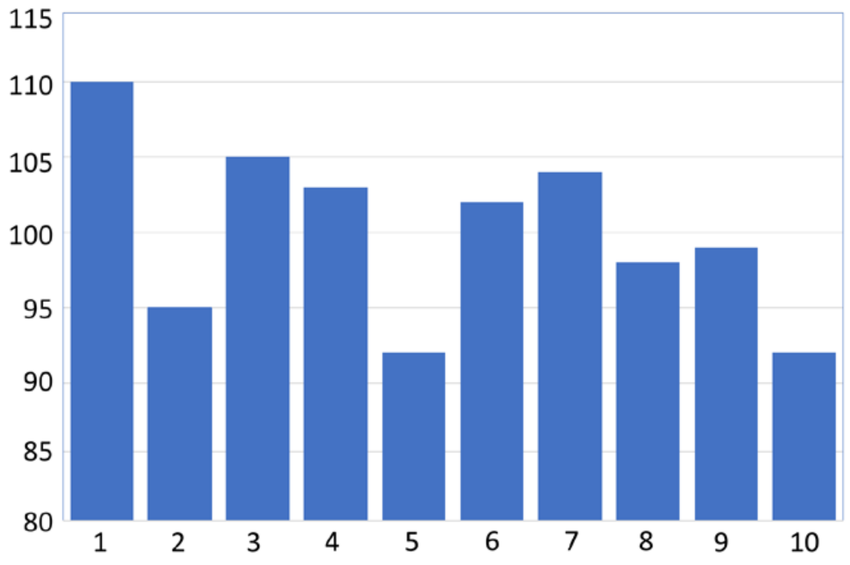


Figure 11.1 – A graph of the distribution of random numbers

We analyzed the following methods, both of which we can use to derive a generic distribution, starting from a uniform distribution of random numbers:

- **Inverse transform sampling method:** This method uses inverse cumulative distribution to generate random numbers.
- **Acceptance-rejection method:** This method uses the samples in the region under the graph of its density function.

A pseudo-random sequence returns integers uniformly distributed in each interval, with a very long repetition period and with a low level of correlation between one element of the sequence and the next.

To self-evaluate the skills that are acquired when generating random numbers, we can try to write some Python code for a bingo card generator. Here, we just limit the numbers from 1 to 90 and make sure that the numbers cannot be repeated and are equally likely.

Applying Monte Carlo methods

Monte Carlo simulation is a numerical method based on probabilistic procedures. Vine is widely used in statistics for the resolution of problems that present analytical difficulties that are not otherwise difficult to overcome. This method is based on the possibility of sampling an assigned probability distribution using random numbers. It generates a sequence of events distributed according to the assigned probability. In practice, instead of using a sample of numbers drawn at random, a sequence of numbers that has been obtained with a well-defined iterative process is used. These numbers are called pseudo-random because, although they're not random, they have statistical properties similar to those of true random numbers. Many simulation methods can be attributed to the Monte Carlo method, which aims to determine the typical parameters of complex random phenomena.

The following diagram describes the procedure leading from a set of distributions of random numbers to a Monte Carlo simulation:

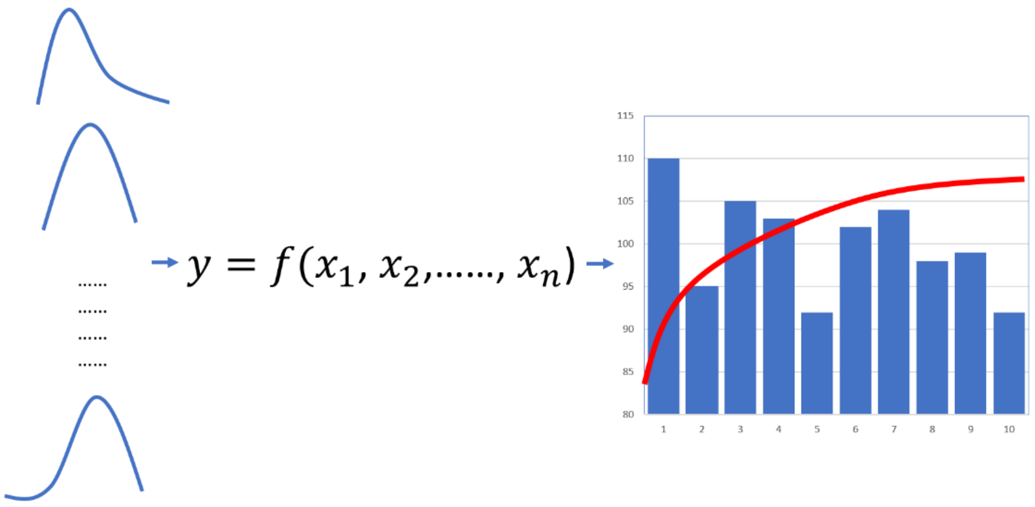


Figure 11.2 – Procedure of a Monte Carlo simulation, starting from a series of distributions of random numbers to one

The Monte Carlo method is essentially a numerical method for calculating the expected value of random variables; that is, an expected value that cannot be easily obtained through direct calculation. To obtain this result, the Monte Carlo method is based on two fundamental theorems of statistics:

- **Law of large numbers**: The simultaneous action of many random factors leads to a substantially deterministic effect.

- **Central limit theorem**: The sum of many independent random variables characterized by the same distribution is approximately normal, regardless of the starting distribution.

Monte Carlo simulation is used to study the response of a model to randomly generated inputs.

Addressing the Markov decision process

Markov processes are discrete stochastic processes where the transition to the next state depends exclusively on the current state. For this reason, they can be called stochastic processes without memory. The typical elements of a Markovian process are the states in which the system finds itself, and the available actions that the decision maker can carry out on that state. These elements identify two sets: the set of states in which the system can be found, and the set of actions available for each specific state. The action chosen by the decision maker determines a random response from the system, which brings it into a new state. This transition returns a reward that the decision maker can use to evaluate their choice, as shown in the following diagram:

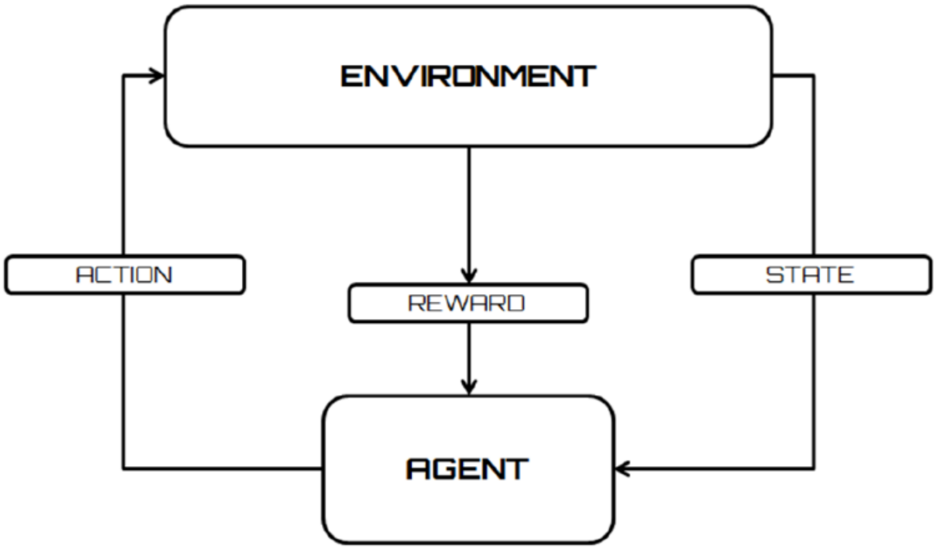


Figure 11.3 – Reward returned from the transition states

Crucial to the system's future choices is the concept of reward, which represents the response of the environment to the action taken. This response is proportional to the weight that the action determines in achieving the objective: it will be positive if it leads to correct behavior, while it will be negative in the case of a wrong action.

Another fundamental concept in Markovian processes is policy: a policy determines the system's behavior in decision making. It maps both the states of the environment and the actions to be chosen in those states, representing a set of rules or associations that respond to a stimulus. In a Markov decision-making model, policy provides a solution that associates a recommended action with each state that can be achieved by the agent. If the policy provides the highest expected utility among the possible actions, it is called an optimal policy. In this way, the system does not have to keep its previous choices in memory. To make a decision, it only needs to execute the policy associated with the current state.

Now, let's consider a practical application of a process that can be treated according to the Markov model. In a small industry, an operating machine works continuously. Occasionally, however, the quality of the products is no longer permissible due to the wear and tear of the spare parts, so the activity must be interrupted and complex maintenance carried out. It is observed that the deterioration occurs after an exponential operating time *Tm* of an average of 40 days, while maintenance requires an exponential random time of an average of 1 day. How is it possible to describe this system with a Markovian model to calculate the probability at a steady state of finding the working machine?

The company cannot bear the downtime of the machine, so it keeps a second one ready to be used as soon as the first one requires maintenance. This second machine is, however, of lower quality, so it breaks after an exponential random work time of 5 days on average and requires an exponential time of 1 day on average to start again. As soon as the main machine is reactivated, the use of the secondary is stopped. If the secondary breaks before the main is reactivated, the repair team insists only on the main one being used, taking care of the secondary only after restarting the main. How is it possible to describe the system with a Markovian model, calculating the probability at a steady state with both machines stopped?

Think about how you might answer those questions using the knowledge you have gained from this book.

Analyzing resampling methods

In resampling methods, a subset of data is extracted randomly or according to a systematic procedure from an original dataset. The aim is to approximate the characteristics of a sample distribution by reducing the use of system resources.

Resampling methods are methods that repeat simple operations many times, generating random numbers to be assigned to random variables or random samples. In these operations, they require more computational time as the number of repeated operations grows. They are very simple methods to implement and once implemented, they are automatic.

These methods generate dummy datasets from the initial data and evaluate the variability of a statistical property from its variability on all dummy datasets. The methods differ from each other in the way dummy datasets are generated. In the following diagram, you can see some datasets that were generated from an initial random distribution:

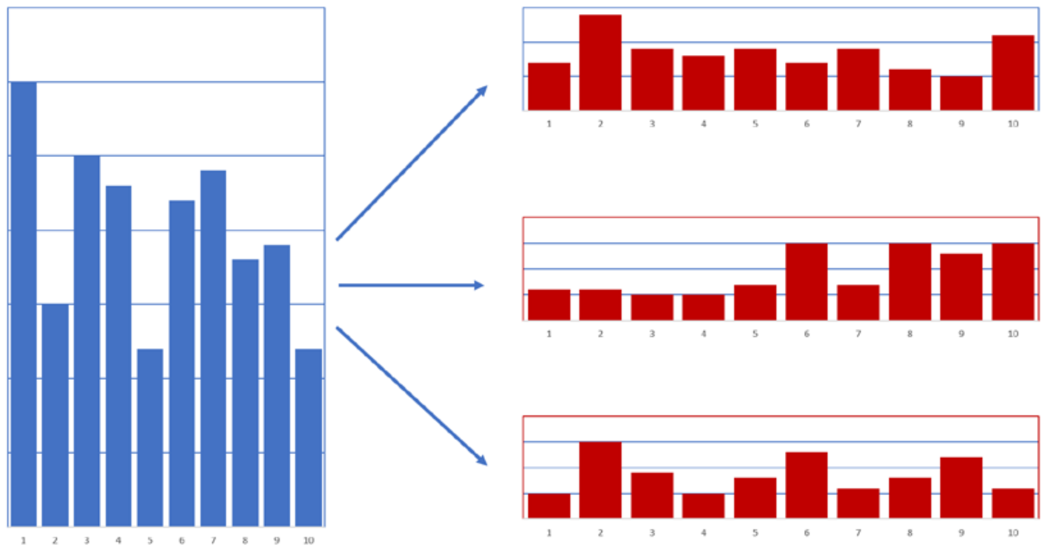


Figure 11.4 – Examples of datasets generated by an initial random distribution

Different resampling methods are available. In this book, we analyzed the following methods:

- **Jackknife technique**: Jackknife is based on calculating the statistics of interest for various sub-samples, leaving out one sample observation at a time. The jackknife estimate is consistent for various sample statistics, such as mean, variance, the correlation coefficient, the maximum likelihood estimator, and others.
- **Bootstrapping**: The logic of the bootstrap method is to build samples that are not observed, but are statistically like those observed. This is achieved by resampling the observed series through an extraction procedure where we reinsert the observations.
- **Permutation test**: Permutation tests are a special case of randomization tests and use series of random numbers formulated from statistical inferences. The computing power of modern computers has made their widespread application possible. These methods do not require assumptions about data distribution to be met.

- **Cross-validation technique**: Cross-validation is a method used in model selection procedures based on the principle of predictive accuracy. A sample is divided into two subsets, of which the first (training set) is used for construction and estimation and the second (validation set) is used to verify the accuracy of the predictions of the estimated model.

Sampling is used if not all the elements of the population are available. For example, investigations into the past can only be done on available historical data, which is often incomplete.

Exploring numerical optimization techniques

Numerous applications, which are widely used to solve practical problems, make use of optimization methods to drastically reduce the use of resources. Minimizing the cost or maximizing the profit of a choice are techniques that allow us to manage numerous decision-making processes. Mathematical optimization models are an example of optimization methods, in which simple equations and inequalities allow us to express the evaluation and avoid the constraints that characterize the alternative methods.

The goal of any simulation algorithm is to reduce the difference between the values predicted by the model and the actual values returned by the data. This is because a lower error between the actual and expected values indicates that the algorithm has done a good simulation job. Reducing this difference simply means minimizing an objective function that the model being built is based on.

In this book, we have addressed the following optimization methods:

- **Gradient descent**: This method is one of the first methods that was proposed for unconstrained minimization and is based on the use of the search direction in the opposite direction to that of the gradient, or anti-gradient. The interest of the direction opposite to the gradient lies precisely in the fact that, if the gradient is continuous, it constitutes a descent direction that is canceled if and only if the point that's reached is a stationary point.
- **Newton-Raphson**: This method is used for solving numerical optimization problems. In this case, the method takes the form of Newton's method for finding the zeros of a function, but applied to the derivative of the function f. This is because determining the minimum point of the function f is equivalent to determining the root of the first derivative.

- **Stochastic gradient descent**: This method solves the problem of evaluating the objective function by introducing an approximation of the gradient function. At each step, instead of the sum of the gradients being evaluated in correspondence with data contained in the dataset, the evaluation of the gradient is used only in a random subset of the dataset.

Using artificial neural networks for simulation

Artificial neural networks (**ANNs**) are numerical models that have been developed with the aim of reproducing some simple neural activities of the human brain, such as object identification and voice recognition. The structure of an ANN is composed of nodes that, analogous with the neurons present in a human brain, are interconnected with each other through weighted connections, which reproduces the synapses between neurons. The system output is updated until it iteratively converges via the connection weights. The information that's derived from experimental activities is used as input data and the result is processed by the network and returned as output. The input nodes represent the predictive variables that we need in order to process the dependent variables that represent the output neurons. The following diagram shows the functionality of an artificial neuron:

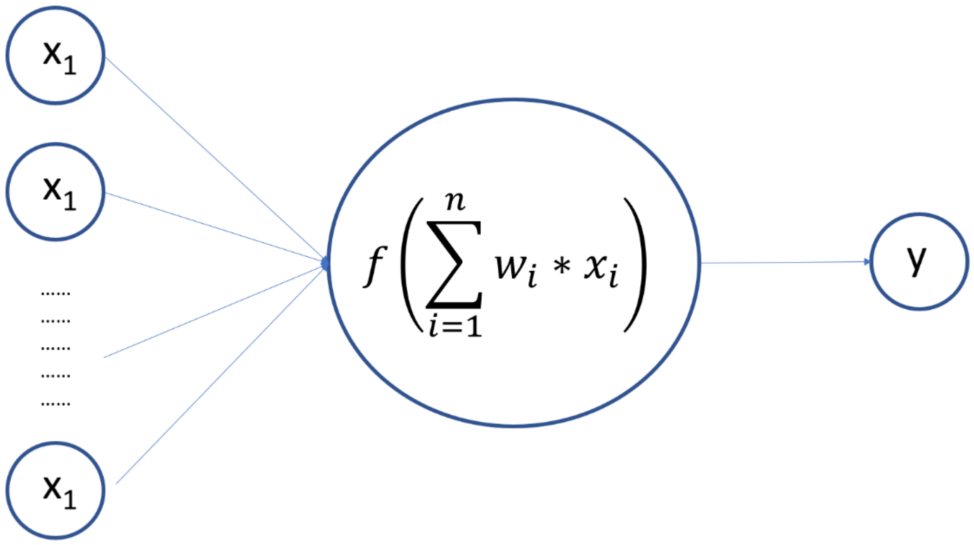


Figure 11.5 – Functionality of an artificial neuron

An ANN's target is the result of calculating the outputs of all the neurons. This means an ANN is a set of mathematical function approximations. A model of this type can simulate the behavior of a real system, such as in pattern recognition. This is the process in which a pattern/signal is assigned to a class. A neural network recognizes patterns by following a training session, in which a set of training patterns are repeatedly presented to the network, with each category that they belong to specified. When a pattern that has never been seen before but belongs to the same category of patterns that it has learned is presented, the network will be able to classify it thanks to the information that was extracted from the training data. Each pattern represents a point in the multidimensional decision space. This space is divided into regions, each of which is associated with a class. The boundaries of these regions are determined by the network through the training process.

Now that we have recapped all the concepts we have learned about throughout this book, let's see how they can be applied to challenges in the real world.

Applying simulation model to real life

The algorithms that we have analyzed in detail throughout this book represent valid tools for simulating real systems. This is why they are widely used in real life to carry out research on the possible evolution of a phenomenon, following a possible choice made on it.

Let's look at some specific examples.

Modeling in healthcare

In the healthcare sector, simulation models have a significant weight and are widely used to simulate the behavior of a system in order to extract knowledge. For example, it is necessary to demonstrate the clinical efficacy of the health intervention under consideration before undertaking an economic analysis. The best available sources are randomized controlled trials. Trials, however, are often designed to leave out the economic aspects, so the key parameters for economic evaluations are generally absent. Therefore, a method is needed to evaluate the effect of disease progression, in order to limit the bias in the cost-effectiveness analysis. This implies the construction of a mathematical model that describes the natural history of the disease, the impact of the interventions applied on the natural history of the disease, and the results in terms of costs and objectives.

The most commonly used techniques are extrapolation, decision analysis, the Markov model, and Monte Carlo simulations:

- In extrapolation, the results of a trial with short follow-up periods are extrapolated beyond the end of the trial itself and consider various possible scenarios, some more optimistic, in which the benefits associated with an intervention are assumed to be constant over time.
- The Markovian model is frequently used in pharmacoeconomic evaluations, especially following the numerous requests for cost-effectiveness evaluations by government organizations.
- An alternative to calculating the costs and benefits of a therapeutic option is the Monte Carlo simulation. As in the Markov model, even in the Monte Carlo simulation, precise states of health and the probability of transition are defined. In this case, however, based on the probability of transition and the result of a random number generator, a path is constructed for each patient until the patient themselves reaches the ultimate state of health envisaged by the model. This process is repeated for each patient in usually very large groups (even 10,000 cases), thus providing a distribution of survival times and related costs. The average values of costs and benefits obtained with this model are very similar to those that we would have calculated by applying the Markov model.
- However, the Monte Carlo simulation also provides a frequency distribution and variance estimates, which allow you to evaluate the level of uncertainty of the results of the model itself. In fact, the Monte Carlo simulation is often used to obtain a sensitivity analysis of the results deriving from the application of the Markov model.

Modeling in financial applications

The Monte Carlo simulation is normally used to predict the future value of various financial instruments. However, as highlighted previously, it is good to underline that this forecasting method is presented exclusively as an estimate and therefore does not provide a precise value as a result. The main financial applications of this method concern pricing options (or derivatives in general) and evaluation security portfolios and financial projects. From this, it is immediately evident that they present an element of analogy.

In fact, options, portfolios, and financial projects have a value that's influenced by many sources of uncertainty. The simulation in question does not lend itself to the evaluation of any financial instrument. Securities such as shares and bonds are not normally valued with the method, precisely because their value is subordinated to a lower number of sources of uncertainty.

The options, on the other hand, are derivative securities, the value of which are influenced by the performance of the underlying functions (which may have the most varied content) and by numerous other factors (interest rates, exchange rates, and so on). The Monte Carlo simulation allows you to generate pseudo-random values for each of these variables and assign a value to the desired option. It should be noted, however, that the Monte Carlo method is only one of those available for pricing options.

Continuing with the category of financial instruments, portfolios are sets of different securities, normally of a varied nature. Portfolios are exposed to a variety of sources of risk. The operational needs of modern financial intermediaries have led to the emergence of calculation methods that aim to monitor the overall risk exposure of their portfolios. The main method in this context is **value at risk (VaR)**, which is often calculated using the Monte Carlo simulation.

Ultimately, when a company must evaluate the profitability of a project, it will have to compare the cost of the same with the revenue generated. The initial cost is normally (but not necessarily) certain. The cash flows that are generated, however, are hardly known a priori. The Monte Carlo method allows us to evaluate the profitability of the project by attributing pseudo-random values to the various cash flows.

Modeling physical phenomenon

The simulation of a physical model allows you to experiment with the model by putting it to the test by changing its parameters. The simulation of a model therefore allows you to experiment with the various possibilities of the model, as well as its limits, in terms of how the model acts as a framework for the experimentation and organization of our ideas. When the model works, it is possible to remove the scaffolding. In this situation, maybe it turns out that it stands up or something new has been discovered. When constructing a model, reference is made to the ideas and knowledge through which the reality of the phenomenon is formally represented.

Just as there is no univocal way to face and solve problems, there is no univocal way to construct the models that describe the behavior of a given phenomenon. The mathematical description of reality struggles to keep considerations of the infinite, complex, and related aspects that represent a physical phenomenon. If the difficulty is already significant for a physical phenomenon, it will be even greater in the case of a biological phenomenon.

The need to select between relevant and non-relevant variables leads to discrimination between these variables. This choice is made thanks to ideas, knowledge, and the school from which those who work on the model come from.

Random phenomena permeate everyday life and characterize various scientific fields, including mathematics and physics. The interpretation of these phenomena experienced a renaissance from the middle of the last century with the formulation of Monte Carlo methods. This milestone was achieved thanks to the intersection of research on the behavior of neurons in a chain reaction and the results achieved in the electronic field with the creation of the first computer. Today, simulation methods based on random number generation are widely used in physics.

One of the key points of quantum mechanics is to determine the energy spectrum of a system. This problem, albeit with some complications, can be resolved analytically in the case of very simple systems. In most cases, however, an analytical solution does not exist. Because of this, there's a need to develop and implement numerical methods capable of solving the differential equations that describe quantum systems. In recent decades, due to technological development and the enormous growth in computing power, we have been able to describe a wide range of phenomena with incredibly high precision.

Modeling public transportation

In recent years, the analysis of issues related to vehicular traffic has taken on an increasingly important role in trying to develop well-functioning transport within cities and on roads in general. Today's transport systems need an optimization process that is coordinated with a development that offers concrete solutions to requests. Through better transportation planning, a process that produces fewer cars in the city and more parking opportunities should lead to a decrease in congestion.

A heavily slowed and congested urban flow, in fact, can not only inconvenience motorists due to the increase in the average travel, but also make road circulation less safe and increase atmospheric and noise pollution.

There are many causes that have led to an increase in traffic, but certainly the most important is the strong increase in overall transport demand; this increase is due to factors of a different nature, such as a large diffusion of cars, a decentralization of industrial and city areas, and an often lacking public transport service.

To try and solve the problem of urban mobility, we need to act with careful infrastructure management and with a traffic network planning program. An important planning tool could be a model of the traffic system, which essentially allows us to evaluate the effects that can be induced on it by interventions in the transport networks, while also allowing us to compare different design solutions. The use of a simulation tool allows us to evaluate some decisional or behavioral problems, quickly and economically, where it is not possible to carry out experiments on the real system. These simulation models represent a valid tool available to technicians and decision makers in the transport sector for evaluating the effects of alternative design choices. These models allow detailed analysis of the solutions being planned at the local level.

There are simulation tools available that allow us to accurately and specifically represent traffic and its evolution instant by instant, all while taking the geometric aspects of the infrastructure and the real behavior of the drivers into consideration, both of which are linked to the characteristics of the vehicle and the driver. Simulation models allow these to be represented on a small scale and, therefore, at a relatively low cost, as well as the effects and consequences related to the development of a new project. Micro-simulations provide a dynamic vision of the phenomenon since the characteristics of the motion of the individual vehicles (flow, density, and speed) are no longer taken into account.

Modeling human behavior

The study of human behavior in the case of a fire, or in cases of general emergency, presents difficulties that cannot be easily overcome since many of the situations whose data it would be important to know about cannot be simulated in laboratory settings. Furthermore, the reliability of the data drawn from exercises in which there are no surprises or anxiety effects such as stress, as well as the possibility of panic that can occur in real situations, can be considered relative. Above all, the complexity of human behavior makes it difficult to predict the data that would be useful for fire safety purposes.

Studies conducted by scientists have shown that the behaviors of people during situations of danger and emergency are very different. In fact, research has shown that during an evacuation, people will often do things that are not related to escaping from fire, and these things can constitute up to two-thirds of the time it takes an individual to leave the building. People often want to know what's happening before evacuating, as the alarm does not necessarily convey much information about the situation.

Having a simulation model capable of reproducing a dangerous situation is extremely useful for analyzing the reactions of people in such situations. In general terms, models that simulate evacuations address this problem in three different ways: optimization, simulation, and risk assessment.

The underlying principles of each of these approaches influence the characteristics of each model. Numerous models assume that occupants evacuate the building as efficiently as possible, ignoring secondary activities and those not strictly related to evacuation. The escape routes chosen during the evacuation are considered optimal, as are the characteristics of the flow of people and exits. The models that consider a large number of people and that treat the occupants as a homogeneous whole, therefore without giving weight to the specific behavior of the individual, tend toward these aspects.

Next steps for simulation modeling

For most of human history, it has been reasonable to expect that when you die, the world will not be significantly different from when you were born. Over the past 300 years, this assumption has become increasingly outdated. This is because technological progress is continuously accelerating. Technological evolution translates into a next-generation product better than the previous one. This product is therefore a more efficient and effective way of developing the next stage of evolutionary progress. It is a positive feedback circuit. In other words, we are using more powerful and faster tools to design and build more powerful and faster tools. Consequently, the rate of progress of an evolutionary process increases exponentially over time, and the benefits such as speed, economy, and overall power also increase exponentially over time. As an evolutionary process becomes more effective and/or efficient, more resources are then used to encourage the progress of this process. This translates into a second level of exponential growth; that is, the exponential growth rate itself grows exponentially.

This technological evolution also affects the field of numerical simulation, which must be compared with the users' need to have more performant and simpler-to-make models. The development of a simulation model requires significant skills in model building, experimentation, and analysis. If we want to progress, we need to make significant improvements in the model building process to meet the demands that come from the decision-making environment.

Increasing the computational power

Numerical simulation is performed by computers, so higher computational powers make the simulation process more effective. The evolution of computational power was governed by **Moore's law**, named after the Intel founder who predicted one of the most important laws regarding the evolution of computational power: every 18 months, the power generated by a chip doubles in computing capacity and halves in price.

When it comes to numerical simulation, computing power is everything. Today's hardware architectures are not so different from those of a few years ago. The only thing that has really changed is the power in processing information. In numerical simulation, information is processed: the more complex a situation becomes, the more the variables involved increase.

The growing processing capacity required for software execution and the increase in the amount of input data have always been satisfied by the evolution of **central processing units** (**CPUs**), according to Moore's law. However, lately, the growth of the computational capacity of CPUs has slowed down and the development of programming platforms has posed new performance requirements that create a strong discontinuity with respect to the hegemony of these CPUs with new hardware architectures in strong diffusion both on the server and on the device's side. In addition, the growing distribution of intelligent applications requires the development of specific architectures and hardware components for the various computing platforms.

Graphical processing units (**GPUs**) were created to perform heavy and complex calculations. They consist of a parallel architecture made up of thousands of small and efficient cores, designed for the simultaneous management of multiple operations. **Field programming gateway array** (**FPGA**) architectures are integrated circuits designed to be configured after production based on specific customer requirements. FPGAs contain a series of programmable logic blocks and a hierarchy of reconfigurable interconnections that allow the blocks to be "wired together."

The advancement of hardware affects not only computing power but also storage capacity. We cannot send information at 1 GBps without having a physical place to contain it. We cannot train a simulation architecture without storing a dataset of several terabytes in size. Innovating means seeing opportunities that were not there previously by making use of components that constantly become more efficient. To innovate means to see what can be done by combining a more performant version of the three accelerators.

Machine learning-based models

Machine learning is a field of computer science that allows computers to learn to perform a task without having to be explicitly programmed for its execution. Evolved from the studies on pattern recognition and theoretical computational learning in the field of artificial intelligence, machine learning explores the study and construction of algorithms that allow computers to learn information from available data and predict new information in light of what has been learned. By building a model that automatically learns to predict new data from observations, these algorithms overcome the classic paradigm of strictly static instructions. Machine learning finds its main use in computing problems where the design and implementation of ad hoc algorithms is not practicable or convenient.

Machine learning has deep links to the field of numerical simulation, which provides methods, theories, and domains of application. In fact, many machine learning problems are formulated as problems regarding minimizing a certain loss function against a specific set of examples (the training set). This function expresses the discrepancy between the values predicted by the model during training and the expected values for each example instance. The ultimate goal is to develop a model capable of correctly predicting the expected values in a set of instances never seen before, thus minimizing the loss function. This leads to a greater generalization of prediction skills.

The different machine learning tasks are typically classified into three broad categories, characterized by the type of feedback that the learning system is based on:

- **Supervised learning**: The sample inputs and the desired outputs are presented to the computer, with the aim of learning a general rule capable of mapping the inputs to the outputs.
- **Unsupervised learning**: The computer only provides input data, without any expected output, with the aim of learning some structure in the input data. Unsupervised learning can represent a goal or aim to extrapolate salient features of the data that are useful for executing another machine learning task.
- **Reinforcement learning**: The computer interacts with a dynamic environment in which it must achieve a certain goal. As the computer explores the domain of the problem, it is given feedback in terms of rewards or punishments in order to direct it toward the best solution.

The following diagram shows the different types of machine learning algorithms:

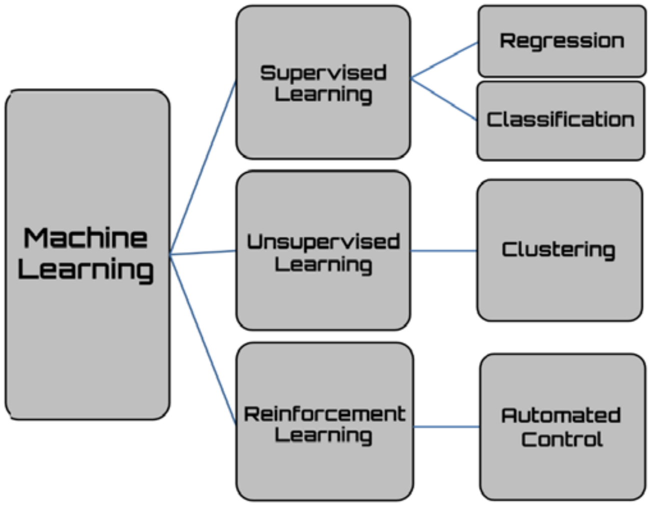


Figure 11.6 – Different types of machine learning algorithms

Automated generation of simulation models

Automated machine learning (**AutoML**) defines applications that can automate the end-to-end process of applying machine learning. Usually, technical experts must process data through a series of preliminary procedures before submitting it to machine learning algorithms. The steps necessary to perform a correct analysis of the data through these algorithms requires specific skills that not everyone has. Although it is easy to create a model based on deep neural networks using different libraries, knowledge of the dynamics of these algorithms is required. In some cases, these skills are beyond those possessed by analysts, who must seek the support of industry experts to solve the problem.

AutoML has been developed with the aim of creating an application that automates the entire machine learning process so that the user can take advantage of these services. Typically, machine learning experts should perform the following activities:

- Preparing the data
- Selecting features
- Selecting an appropriate model class
- Choosing and optimizing model hyperparameters
- Postprocessing machine learning models
- Analyzing the results obtained

AutoML automates all these operations. It offers the advantages of producing simpler and faster-to-create solutions that often outperform hand-designed models. There are several AutoML frameworks available, each of which has characteristics that indicate its preferential use.

Summary

In this chapter, we summarized the technologies that we have exposed throughout this book. We have seen how to generate random numbers and listed the most frequently used algorithms for generating pseudo-random numbers. Then, we saw how to apply Monte Carlo methods for numerical simulation based on the assumptions of two fundamental laws: the law of large numbers and the central limit theorem. We then went on to summarize the concepts that Markovian models are based on and then analyzed the various resampling methods that are available. After that, we explored the most used numerical optimization techniques and learned how to use ANNs for numerical simulation.

Subsequently, we mentioned a series of fields in which numerical simulation is widely used and looked at the next steps that will allow simulation models to evolve.

In this book, we studied various computational statistical simulations using Python. We started with the basics in order to understand various methods and techniques to deepen our knowledge of complex topics. At this point, the developers working with the simulation model would be able to put their knowledge to work, adopting a practical approach to the required implementation and associated methodologies so that they're operational and productive in no time. I hope that I have provided detailed explanations of some essential concepts through the use of practical examples and self-assessment questions, exploring numerical simulation algorithms and providing an overview of the relevant applications in order to help you make the best models for your needs.

Other Books You May Enjoy

Leave a review - let other readers know what you think

Index