

Fundamentals of Machine Learning using Python

Euan Russano and Elaine Ferreira Avelino



FUNDAMENTALS OF MACHINE LEARNING USING PYTHON

FUNDAMENTALS OF MACHINE LEARNING USING PYTHON

Euan Russano and Elaine Ferreira Avelino



www.arclerpress.com

Fundamentals of Machine Learning using Python

Euan Russano and Elaine Ferreira Avelino

Arcler Press

2010 Winston Park Drive,

2nd Floor

Oakville, ON L6H 5R7

Canada

www.arcлерpress.com

Tel: 001-289-291-7705

001-905-616-2116

Fax: 001-289-291-7601

Email: orders@arcлерeducation.com

e-book Edition 2020

ISBN: 978-1-77407-427-5 (e-book)

This book contains information obtained from highly regarded resources. Reprinted material sources are indicated and copyright remains with the original owners. Copyright for images and other graphics remains with the original owners as indicated. A Wide variety of references are listed. Reasonable efforts have been made to publish reliable data. Authors or Editors or Publishers are not responsible for the accuracy of the information in the published chapters or consequences of their use. The publisher assumes no responsibility for any damage or grievance to the persons or property arising out of the use of any materials, instructions, methods or thoughts in the book. The authors or editors and the publisher have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission has not been obtained. If any copyright holder has not been acknowledged, please write to us so we may rectify.

Notice: Registered trademark of products or corporate names are used only for explanation and identification without intent of infringement.

© 2020 Arcler Press

ISBN: 978-1-77407-365-0 (Hardcover)

Arcler Press publishes wide variety of books and eBooks. For more information about Arcler Press and its products, visit our website at www.arcлерpress.com

ABOUT THE AUTHORS



Euan Russano was born in Minas Gerais, Brazil. He is a Chemical Engineer since 2012 by the Rural University of Rio de Janeiro (UFRRJ). He obtained his Msc. in 2014 at UFRRJ, in the area of Chemical Engineering, with specializazion in Process Control. In 2014, Russano began to develop his PhD at the University Duisburg-Essen (UDE) in the field of Water Science. His carrer was initiated in a Petrobras project in the Polymers Laboratory, at UFRRJ. From 2012 to 2014 he worked in the Fluids Flow Laboratory (Petrobras/ UFRRJ) with oil well pressure control. Since 2014 he works as an research assistant at the University of Duisburg-Essen, with water systems identification and control.



Elaine Ferreira Avelino was born in Rio de Janeiro, Brazil. She obtained her Bsc in Forestry Engineering at the Rural University of Rio de Janeiro (UFRRJ) in 2007 and Msc. in 2012 at UFRRJ, in the area of Forestry and Environmental Sciences, with specializazion in Wood Technology. She started her career in the Secretary for Environment of Rio de Janeiro, with Urban and Environmental Planning. Elaine was a professor of Zoology, Enthomology, Forestry Parasithology and Introduction to Research at the Pitagoras University. Since 2013 she works as an international consultant for forest management and environmental licensing.

TABLE OF CONTENTS

<i>List of Figures</i>	<i>xi</i>
<i>List of Tables</i>	<i>xv</i>
<i>List of Abbreviations</i>	<i>xvii</i>
<i>Preface</i>	<i>xix</i>
Chapter 1 Introduction to Python.....	1
1.1. What Is Python	2
1.2. What Makes Python Suitable For Machine Learning?	4
1.3. What Are Other Computational Tools For Machine Learning?	5
1.4. How To Obtain And Configure Python?	9
1.5. Scientific Python Software Set.....	11
1.6. Modules	14
1.7. Notebooks	15
1.8. Variables And Types	16
1.9. Operators And Comparison	18
Chapter 2 Computing Things With Python	21
2.1. Formatting And Printing to Screen.....	22
2.2. Lists, Dictionaries, Tuples, And Sets.....	25
2.3. Handling Files	32
2.4. Exercises.....	34
2.5. Python Statements	36
2.6. For And While Loops	38
2.7. Basic Python Operators	44
2.8. Functions.....	47
Chapter 3 A General Outlook on Machine Learning.....	51

Chapter 4	Elements of Machine Learning	61
4.1.	What Is Machine Learning?.....	62
4.2.	Introduction To Supervised Learning	63
4.3.	Introduction To Unsupervised Learning.....	71
4.4.	A Challenging Problem: The Cocktail Party	73
Chapter 5	Linear Regression With One Variable	77
5.1.	Introduction.....	78
5.2.	Model Structure	78
5.3.	Cost Function	80
5.4.	Linear Regression Using Gradient Descent Method Using Python.....	96
	Exercises.....	111
Chapter 6	A General Review On Linear Algebra	113
6.1.	Introduction.....	114
6.2.	Matrices And Vectors	114
6.3.	Addition	116
6.4.	Multiplication	117
6.5.	Matrix-Vector Multiplication.....	118
6.6.	Matrix-Matrix Multiplication.....	121
6.7.	Inverse And Transpose.....	123
	Exercises.....	127
Chapter 7	Linear Regression With Multiple Inputs/Features.....	131
7.1.	Gradient Descent For Multiple Variables Linear Regression.....	134
7.2.	Normal Equation	137
7.3.	Programming Exercise: Linear Regression With Single Input And Multiple Inputs.....	140
Chapter 8	Classification Using Logistic Regression Model	155
8.1.	Logistic Regression Model Structure.....	156
8.2.	Concept Exercise: Classifying Success In Exam According Hours Of Study	158
8.3.	Programming Exercise: Implementation Of The Exam Results Problem In Python From Scratch	160
8.4.	Bonus: Logistic Regression Using Keras.....	165

Chapter 9	Regularization.....	167
9.1.	Regularized Linear Regression	171
Chapter 10	Introduction To Neural Networks	173
10.1.	The Essential Block: Neurons	174
10.2.	How To Implement A Neuron Using Python And Numpy.....	176
10.3.	Combining Neurons to Build a Neural Network	177
10.4.	Example Of Feedforward Neural Network.....	178
10.5.	How To Implement A Neural Network Using Python and Numpy.....	179
10.6.	How To Train A Neural Network	183
10.7.	Example Of Calculating Partial Derivatives	187
10.8.	Implementation Of A Complete Neural Network With Training Method	189
Chapter 11	Introduction To Decision Trees and Random Forest	195
11.1.	Decision Trees	196
11.2.	Random Forest.....	205
11.3.	Programming Exercise – Decision Tree From Scratch With Python	207
Chapter 12	Principal Component Analysis.....	219
12.1.	Introduction.....	220
12.2.	Mathematical Concepts	220
12.3.	Principal Component Analysis Using Python	227
Chapter 13	Classification Using K-Nearest Neighbor Algorithm.....	233
13.1.	Introduction.....	234
13.2.	Principles and Definition of KNN	234
13.3.	Algorithm	235
13.4.	Example and Python Solution	237
Chapter 14	Introduction To Kmeans Clustering.....	245
14.1.	How Kmeans Works?	246
14.2.	Kmeans Algorithm	247
Chapter 15	Computing With Tensorflow: Introduction And Basics	253
15.1.	Installing Tensorflow Library	254

15.2. Tensors.....	255
15.3. Computational Graph and Session.....	255
15.4. Operating With Matrices.....	256
15.5. Variables.....	260
15.6. Placeholders.....	260
15.7. Ways Of Creating Tensors	262
15.8. Summary	265
Chapter 16 Tensorflow: Activation Functions And Optimization	267
16.1. Activation Functions	268
16.2. Loss Functions	273
16.3. Optimizers.....	274
16.4. Metrics	277
Chapter 17 Introduction To Natural Language Processing.....	279
17.1. Definition Of Natural Language Processing	280
17.2. Usage Of NLP.....	280
17.3. Obstacles In NLP	281
17.4. Techniques Used In NLP	281
17.5. NLP Libraries	282
17.6. Programming Exercise: Subject/Topic Extraction Using NLP.....	282
17.7. Text Tokenize Using NLTK.....	286
17.8. Synonyms From Wordnet.....	288
17.9. Stemming Words With NLTK.....	290
17.10. Lemmatization Using NLTK	291
Chapter 18 Project: Recognize Handwritten Digits Using Neural Networks	293
18.1. Introduction.....	294
18.2. Project Setup	294
18.3. The Data	294
18.4. The Algorithm	294
Appendix A	301
Bibliography.....	319
Index.....	321

LIST OF FIGURES

Figure 4.1: Housing price in Buenos Aires, in USD (hypothetical).

Figure 4.2: Housing price in Buenos Aires, in USD (hypothetical) with initial model.

Figure 4.3: Housing price in Buenos Aires, in USD (hypothetical) with initial model (Model 1) and second model (Model 2).

Figure 4.4: Breast Cancer x Tumor Size – representation in x–y plane.

Figure 4.5: Breast Cancer x Tumor Size – representation in single row.

Figure 5.1: Housing price in Mountain City, in USD (hypothetical).

Figure 5.2: Prediction of Housing Price using “random” parameter values.

Figure 5.3: Prediction of Housing Price using “slightly better” parameter values.

Figure 5.4: Values of J for Housing Price using $a_0 = 0$ as a fixed parameter.

Figure 5.5: Values of J for Housing Price using $a_1 = 0.74$ as a fixed parameter.

Figure 5.6: Surface showing values of J for Housing Price and the approximate minimum, pointing the values of a_0 and a_1 .

Figure 5.7: Values of J for linear regression with fixed $a_0 = -3$ and varying a_1 for the data in Table 5.2.

Figure 5.8: Value of learning rate α , is too small, requiring too much iterations.

Figure 5.9: Value of learning rate α , is reasonable, requiring few iterations and no oscillations.

Figure 5.10: Value of learning rate α , is too high, producing relative divergence in the search for the minimum.

Figure 5.11: Example of local minima and global minima in non-linear cost function.

Figure 5.12: Plot of dataset.

Figure 5.13: Dataset with prediction.

Figure 5.14: Test of plotJ function.

Figure 5.15: Optimum linear regression model using GD.

Figure 5.16: Evolution of optimization process in 2D plot.

Figure 5.17: Evolution of optimization process in cost function.

Figure 7.1: Scatter plot of profit x population data.

Figure 7.2: Plot of observed and predicted profit x population data.

Figure 7.3: Histogram of input feature 1.

Figure 7.4: Histogram of input feature 2.

Figure 7.5: Scatter plot of profit for multiple input example.

Figure 7.6: Plot of observed and predicted profit for multiple inputs example.

Figure 8.1: Dataset showing hours of study and exam results.

Figure 8.2: Dataset (points) and prediction (curve) showing hours of study and exam results.

Figure 9.1: Stock price dataset.

Figure 9.2: Stock price linear model $y = ax + b$. (overfit2.pdf)

Figure 9.3: Stock price quadratic model $y = ax^2 + bx + c$. (overfit3.pdf)

Figure 9.4: Stock price 6th order polynomial model $y = ax^6 + bx^5 + \dots + c$. (overfit4.pdf)

Figure 10.1: Simple neural network.

Figure 11.1: Dataset example.

Figure 11.2: Decision tree for dataset in Figure 11.1.

Figure 11.3: Decision Tree Boundary at $x = 2$.

Figure 11.4: Dataset Example with three labels.

Figure 11.5: Decision tree with three labels.

Figure 11.6: All possible splits in x feature.

Figure 11.7: All possible splits in the _____ feature.

Figure 11.8: Heatmap with cross correlation matrix between features.

Figure 12.1: Plot of original dataset.

Figure 12.2: Reconstructed dataset.

Figure 13.1: Example dataset containing data with three labels.

Figure 13.2: Point with unknown class plotted with the kNN dataset.

Figure 13.3: Point and distance to nearest neighbor ($k = 1$).

Figure 13.4: Point and distance to nearest neighbors ($k = 3$).

Figure 14.1: Dataset to be clustered.

Figure 14.2: Clustering dataset with $k = 2$.

Figure 14.3: Clustering dataset with $k = 3$.

Figure 14.4: Clustering dataset with $k = 4$.

Figure 16.1: Hyperbolic tangent activation function.

Figure 16.2: Sigmoid activation function.

Figure 16.3: ReLU activation function.

Figure 16.4: Softplus activation function.

Figure 18.1: General neural network structure for MNIST dataset problem.

LIST OF TABLES

Table 3.1: E-mail List

Table 4.1: Housing Price (USD) x Size (m^2) in Buenos Aires (Hypothetical Data)

Table 4.2: Data Used to Generate Model 1

Table 4.3: Data Used to Generate Model 2

Table 4.4: Breast Cancer x Tumor Size

Table 4.5: Breast Cancer x Tumor Size – In Terms of 0 and 1

Table 5.1: Housing Dataset Sample

Table 5.2: Data Example for GD Intuition

Table 5.3: Company Sales and Advertising Investment from 2009 to 2018

Table 6.1: Dataset of Poverty Level and Teen Birth Rate for Four US States

Table 7.1: Example of Multiple Feature House Pricing Table

Table 7.2: Example 2 of Multiple Feature House Pricing Table

Table 7.3: Comparison Between Gradient Descent and Normal Equation

Table 8.1: Hours of Study and Exam Result for 20 Students

Table 8.2: Logistic Regression Results for Hours of Study and Exam

Table 8.3: Summary of Probabilities

Table 10.1: Dataset of (Hypothetical) Weight, Height, and Gender

Table 11.1: Probability in Binary Classification

Table 11.2: Example Splitting

Table 11.3: Summary of All Possible Splits and Their Gini Impurity

Table 11.4: Example of Bagging (Cross Repeats So It Is Considered the Major Vote)

Table 13.1: Dataset for kNN Example – Animal Classification

Table 13.2: Unknown Animal To Be Classified

Table 15.1: Constant Tensors

Table 15.2: Tensorflow Operators

Table 15.3: Tensorflow Variables

Table 15.4: Operations on a Session

LIST OF ABBREVIATIONS

ANN	Artificial Neural Networks
API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
GD	Gradient Descent
GPU	Graphics Processing Unit
IDLE	Integrated Development and Learning Environment (Python IDE)
kNN	k-Nearest Neighbors
LLS	Linear Least Squares Method
ML	Machine Learning
MSE	Mean Squared Error
PCA	Principal Component Analysis
PDF	Portable Document Format
SGD	Stochastic Gradient Descent
SQ	Sum of Squares
SVD	Single-Value Decomposition
USD	United States Dollar

PREFACE

Machine learning is a field of study that has drastically grown in the last decades. Whenever one enters the Internet and uses a search engine or an automatic translation service, machine learning is present. Modern surveillance systems are equipped with systems, which can detect and classify objects using machine-learning techniques. When predicting stock prices or general market features, economists use this same technique to have accurate results. These are just to mention a few applications that machine learning has found in different areas.

This book is intended for under-graduate students and enthusiasts who are interested in starting to develop their own machine learning models and analysis. It provides a simple but concise introduction to the main knowledge streams that are necessary to start into this area, even entering a bit beyond the basics with an introduction of TensorFlow, a state-of-the-art framework to develop machine learning models.

Since most applications are currently being developed in Python, the authors chose to focus on this programming language. The first two chapters provide a general introduction to machine learning and Python programming language. Chapter 3 introduces how a machine-learning model can be developed from zero using a story-based description.

On Chapter 4, main concepts that appear when talking about machine learning are described. The following chapters go deeper into each modeling technique. Chapters 5, 6, and 7 deals with the linear regression model and the concepts on linear algebra to understand more complex techniques. Chapter 8 describes the logistic regression model, the equivalent of linear regression for classification. Chapter 9 introduces the concept of regularization to later present neural networks in Chapters 10 and 11. How Decision Trees works with examples, as well as the development of Random Forests are mentioned in Chapter 12.

Chapter 13 shows how Principal Component Analysis (PCA) can be used to simplify a dataset, reducing its dimensions and enabling one to retrieve information of it. Chapter 14 deals with classification problems by showing how k Nearest Neighbors model works with implementation in Python as the previous models.

Chapters 15 and 16 presents to the reader the state-of-the-art framework TensorFlow, a collection of functionalities which highly enhances the development of machine learning models, with the incorporation of efficient computations and intuitive programming.

1

Introduction to Python

CONTENTS

1.1. What Is Python	2
1.2. What Makes Python Suitable For Machine Learning?	4
1.3. What Are Other Computational Tools For Machine Learning?	5
1.4. How To Obtain And Configure Python?	9
1.5. Scientific Python Software Set.....	11
1.6. Modules	14
1.7. Notebooks.....	15
1.8. Variables And Types	16
1.9. Operators And Comparison	18

1.1. WHAT IS PYTHON

Created in 1991 by Guido van Rossum and developed by Python Software Foundation, Python is a high-level, general-purpose programming language.

High-level: Every programming language strongly focused on *abstraction* from the details of the computer is called a high-level language. It may use natural language elements and syntax to make the effort of developing a program simpler and more understandable when compared with low-level programming languages. Therefore, how high-level a programming language depends on the amount of abstraction that it incorporates.

General-purpose: As the term indicates, a general-purpose programming language is designed to be used in a wide variety of fields and applications. This is only possible because the “architecture” of the language does not constrain it limiting to a specific application domain. The opposite of a general-purpose language is a domain-specific one, which is designed to be used within certain application domain.

Because Python incorporates such concepts in its structure, it improves readability and allows developers (programmers) to use fewer lines of code to reach an objective or to express concepts.

Python is a dynamically-type language. This means that a variable name is related to an object unless it is null (in some cases even null is an object). In opposition, statically typed languages use declaration of variables to relate it both to a type (numeric, string) at compile time and to an object (though the latter is optional, if not the variable is null). Being dynamically typed makes Python a programming language much more flexible than static ones, since variables can change types during a program execution through assignment statements.

Python is a strongly typed language, which means that variables cannot be implicitly coerced to unrelated types. Trying to do so raises an error or an exception. A strongly typed language requires an explicit conversion of type before concatenating variables of different types. On the other hand, weakly typed language allows this interchange, but the result may be unexpected if the programmer is not well aware of what he is doing. For instance, the following code will work completely fine in a weakly typed language.

```
a = 1 # a number  
b = "2" # a string  
c = concatenate(a,b) # output is "12"
```

```
d = add(a,b) # output is 3
```

All the commands above are completely legal in a weakly typed programming language. However, in a strongly-typed language, it would raise an error. So the following modification would be necessary.

```
c = concatenate (str(a), b) # first convert a to string then concatenate, output  
is "12"
```

```
d = add(a, int(b)) # first convert b to an integer then add, output is 3
```

Python is an object-oriented programming language. In simple terms, this concept refers to a paradigm where a program is composed by objects containing information (data) or attributes in the form of fields, and actions or procedures, often referred to as methods. Such concept states that an object's procedures are able to read/write information of the objects with which it is associated. Such concept may or may not be explicitly incorporated on a program when using Python. Implicitly it is always used, since any data type in Python is by definition an object.

Another important concept is that Python contains an automatic memory management system. This means that the program “knows” when to allocate or deallocate memory for variables and data arrays. This brings a great advantage since it optimizes memory usage automatically, whereas in languages such as C++ one needs to explicitly allocate and deallocate memory for each variable used in a program. If this process is not properly done in big programs, computer memory can be drained which harms performance.

Python is a free-to-use programming language with a comprehensive standard library and a continuously growing set of libraries for a variety of applications being developed by the international community of Python developers.

Other concepts that Python incorporates, as mentioned by Langtangen (2011):

- **Cleanliness and simplicity:** It was developed to be easy-to-read and intuitive, with a minimalist syntax. This characteristic highly enhances maintainability of projects according to their sizes.
- **Expressive language:** The usage of fewer lines of code leads to less errors (bugs), which helps developers to develop programs faster and such products are easier to maintain.
- **Interpreted language:** Python code does not need to be compiled, because it can be directly read and run by the interpreter.

Naturally, as nothing is perfect, Python has some disadvantages. Run time is comparatively slow when compared with statically-typed, compiled languages as C and Fortran. This disadvantage is inevitably connected with being an interpreted, dynamically typed programming language.

Another disadvantage is connected with the fact of being an open-source, free programming language with packages being developed by the international community. This makes Python decentralized, and packages can be found in different locations, as well as documentations and developing environments. For beginners, it may be challenging to know where to start and how to be systematic on aggregating resources to a personal development system.

1.2. WHAT MAKES PYTHON SUITABLE FOR MACHINE LEARNING?

The answer to such questions can be easily found by searching on the internet. One will find a rich source of answers, and most of them are valid regarding the usage of Python for machine learning.

One simple and direct answer is provided by Patel (2018). Python is easy to understand, and it is capable of encapsulating relatively complex problems in statements which makes the problem look relatively simple, reducing the effort. Also, the huge number of packages available makes it flexible to read different types of data (video, image, numeric, text, etc.) without the need of “reinventing the wheel.” Packages are available to work with images (numpy, scikit), text (nltk, numpy), audio (librosa), solve ML problems (pandas, scikit), data visualization (matplotlib, seaborn), deep learning (TensorFlow, pytorch), scientific computing (scipy), web development (Django, flask) and many other applications and objectives that will go way beyond the scope of the present book.

The huge and integrated worldwide community of Python existent nowadays (2019) also makes it very helpful for Machine Learning, as mentioned by Lefkowitz and Gall (2018). Lefkowitz, who won the PSF’s Community Service Award in 2017, mentions that Python has replaced Lisp programming language, a formerly widely used tool for Artificial Intelligence (according to the author, almost synonyms). The fact that such a thing happened is because, in addition to being high-level as Lisp is, Python has a huge and friendly library ecosystem, with stable integration for operating system. Specifically, Numerical Python (Numpy, a library) and related

tools allow high-level development, with a research-friendly environment and high-performance when dealing with great amount of data, which is essential for Machine Learning. Besides these aspects, Lefkowitz mentions that the support found by the programming community is what has given a foundation for the expressive growth of this language. This community integrates people from a variety of disciplines, like statisticians, engineers, biologists, business analysts, which constructed a social interchange. With exception of JavaScript, Python's community is the one, which has acknowledged this integration and necessity the most. This is essential for Machine Learning, since it is dependent on huge amounts of data from real-world sources for training and system input.

1.3. WHAT ARE OTHER COMPUTATIONAL TOOLS FOR MACHINE LEARNING?

Machine Learning is intimately related with Data Science. Regarding the latter, the most common tools employed by scientists and engineers are Spreadsheet software like Microsoft Excel® and SAS. However, such tools cannot handle huge amounts of data and they cannot be used to directly handle a company's raw data. Also, there is smaller community support for a wider usage such as the requirements for Machine Learning applications (Patel, 2018).

MATLAB is a piece of software currently used for Machine Learning in many universities and companies. It contains the necessary tools for the development of applications for predictive maintenance, sensor analytics, finance, and communication electronics. Some of its advantages when employed in Machine Learning are (The MathWorks Inc., 2019):

- Simple and intuitive user interface for training and comparing models. In some instances, no programming is involved, but only point and click apps;
- Tools for advanced signal processing and feature extraction;
- Model performance optimization through Automatic hyperparameter tuning and feature selection;
- Easy scaling apps from development to production, using big data and clusters;
- Automated generation of C/C++ code for high-performance applications (essential for complex Machine Learning problems);

- Popular algorithms for Machine Learning, involving classification, regression, clustering for supervised and unsupervised learning are already available, the programmer does not need to “reinvent the wheel”;
- Execution is faster than open-source on most statistical and machine learning computations.

However, some disadvantages that MATLAB has in comparison with Python are:

- MATLAB is commercial (paid) while Python is free, open-source;
- MATLAB contains a closed set of packages, only developed by its manufacturer MATHWORKS. On the other hand, Python is a huge community continuously developing packages and libraries which increases its functionality reducing the development time and making it easier to find solutions for bugs and problems.
- Though MATHWORKS states that it is faster, it is well reported that MATLAB is in many situations, slower than other frameworks (e.g., TensorFlow) and it is almost impossible to use at for deployment, except for prototyping.

Another great tool is the R programming language. It is completely focused on statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, etc.) computing, with similarities with Python regarding syntax and data visualization. As Python, it is free and can be used in different operating systems (UNIX, Mac OS, and Windows). Some advantages of its use for Machine Learning (or Data Science in general) are:

- Efficient data management and storage;
- A good set of operators for calculations on vectors and matrices;
- An integrated collection of intermediate tools for data analysis, with large and coherent application;
- Graphical tools for data analysis which can be either printed on screen or saved on hard drive;
- All the basic constructs for programming logic, such as conditionals, loops, user-defined recursive functions as well as input and output facilities.

One disadvantage of R is its lack of flexibility, i.e., it works on a particular form of dataset, whereas Python can work with different formats and it provides the development tools to be used with other systems.

Lastly, another tool which finds its usage in Machine Learning is C++ programming language. C++ is a statically-typed, compiled programming language. These features, besides others, make C++ a highly-efficient language regarding runtime. And as runtime, is essential for complex problems in Machine Learning, C++ may be crucial in such cases. It is important to notice that Python libraries such as TensorFlow and Torch are implemented in C++ under the hood for computing efficiency. And many companies use C++ in their Machine Learning algorithms, though they may use Python or R for experimentation and prototyping. On the other hand, the time required for developing the same algorithm in C++ or in Python is much greater in C++, because of its complexity and difficulty in implementation. To test a code in C++, one needs to compile it first and run it. If an error or a bug is found, the developer needs to rewrite the code, recompile it and run again to see if the problem is solved. In Python, once the code is written it can be run directly, so any bug or error can be corrected much faster which makes the development of Machine Learning algorithms in C++ relatively slower than in Python. So the way to go would be to use Python for development, and later C++ for production, if performance is an issue.

Yegulalp (2019) mentions some other tools popular to be used for Machine Learning (some here are C++ libraries):

- Venerable Shogun (or simply Shogun): Created in 1999 and written in C++, Shogun is a library for Machine Learning with native support for Microsoft Windows and Scala language, and can also be used with Java, Python, C#, Ruby, R, Lua, Octave and Matlab.
- Accord.Net Framework: Machine Learning and signal processing framework for .Net as an extension of a previous project called AForge.net. The tool contains libraries for audio image analysis and processing. The algorithms for vision processing can be used for face detection, stitching together images and tracking moving objects. In addition to these tools, the framework also incorporates more traditional machine learning algorithms such as neural networks and decision-tree systems.
- Apache Mahout: A framework connected to Hadoop (<https://hadoop.apache.org/>), which can also be used for Hadoop projects that could be migrated to stand-alone applications. The last versions reinforced support for Spark framework and support was

- added to ViennaCL library for GPU-accelerated linear algebra.
- Spark MLlib: A framework containing many common algorithms and useful data types designed to run at speed and scale. The primary language of Spark is Java, but still it can be used in connection with Python through NumPy library. Also, Scala users can develop code for it and R users can plug into Spark.
 - H2O: Focused on business applications, H2O is tool which can bind with Python, Scala, R and Java. It can be used, for instance for fraud detection or trend prediction in business-like applications.
 - Cloudera Oryx: A framework to run machine learning models on real-time data through Spark and Kafka stream processing framework. It was developed with the purpose of building projects for anomaly detection or recommendation engines on real-time systems working with historic and real-time data. It is actually (2019) on version 2.0 which its components coupled in a lambda architecture. Hyperparameter selection among other abstractions and algorithms can be added according user necessity.
 - GoLearn: This is the framework for machine learning for Google Go language, created with the objective of simplicity and customizability. The simplicity is clear on the way that data is handled, resembling Scipy and R. The feature of customizability appears on the way data structures can be expanded. With the development of this framework, the programmers also created a Go wrapper to embed in Shogun toolbox, in the Vowpal Wabbit library.
 - Weka: More focused on Data Mining, Weka consists in a collection of machine learning algorithms written in Java. Its license (GNU CPLv3) allows extended functionality, with official and unofficial packages available. Lately it has been made available a set of wrappers, so Weka can be used with Hadoop, but still it doesn't support Spark.
 - Deeplearn.js: Available as an online tool by Google, this framework incorporates neural network models which can be directly trained in any modern browser, without any client-side software. Its performance is boosted by the availability of GPU-accelerated computation through WebGL API, which removes the limitation of system's CPU. The pattern used follows Google's TensorFlow.

- ConvNetJS: A JavaScript library for neural network machine learning directly in a browser. An alternative version is also available for Node.js. One interesting characteristic of this library is that it can make proper use of JavaScript's asynchronicity. The documentation includes plenty of demo examples.

1.4. HOW TO OBTAIN AND CONFIGURE PYTHON?

1.4.1. Installation Using Conda

Conda is an open-source package and environment manager that can be used with command line statements at the Anaconda Prompt for Windows, or in a Terminal window for Mac OS and Linux. It can be used for Python but also to a variety of programming languages: Scala, R, Ruby, Lua, Java, etc.

Because it is a package manager, it can help user to find and install packages. It can be used to generate environments with different version of Python, for instance, without the necessity of switching an environment manager, since Conda is also an environment manager. Because of this, it is very use for a user to create a completely separate environment and run different version of Python without shifting to different software or to interfere in the normal version of Python or in others environments being used.

The most straight forward manner of obtaining Conda is to first install Miniconda, a “simpler” version of Anaconda containing only Conda and its dependencies. If one choose to install directly Anaconda, then it comes with Conda plus over 720 open-source packages. Anaconda includes a graphical installer, which the user only needs to follow the step by step instructions to have a fully-functional version of Anaconda.

To download miniconda, access <http://conda.pydata.org/miniconda.html> and follows the links and instructions to download and install the software.

1.4.2. Installation on Linux

Installation of Python on Linux is relatively simple and direct. In Ubuntu Linux, python can be installed by using the following command on a Terminal:

```
$ sudo apt-get install python
```

1.4.3. Installation on Mac OS X

By default Python is already included in Mac OS X. Still, it can be useful to install a new python environment. For such purpose, there are two most common package managers, Macports or Fink. Macports installation of python makes much easier to install additional packages instead of using the default version of Python on Mac OS X. Installation can be done using Macports through the following command on a Terminal:

```
$ sudo port install py36-ipython +pyside+notebook+parallel+scientific  
$ sudo port install py36-scipy py27-matplotlib py27-sympy  
$ sudo port install py36-spyder
```

Even performing such installation, any .py file or any IDE will still be associated with default Mac OS Python version. To correct that and associate the commands python with the installed versions via Macports, perform the following commands:

```
$ sudo port select python python36  
$ sudo port select ipython ipython36
```

To use the other package manager, Fink, first install it, then use the following set of commands to install packages as well as other packages commonly used for Machine Learning.

```
$ sudo fink install python36 ipython-py36 numpy-py36 matplotlib-py36  
scipy-py36 sympy-py36  
$ sudo fink install spyder-mac-py36
```

1.4.4. Installation on Windows

Differently from Linux or Mac OS, Python does not come by default on Windows, and this operating system also lacks a “natural” package manager system. Therefore, the best way to set up Python is to install a package and environment manager. Some alternatives are:

- Enthought Python Distribution (EPD): a commercial system but also available for free for academic use.
- Anaconda: An open-source package and environment manager not only for Python but for a variety of programming languages. Completely free for personal or commercial use as well as for redistribution. Contains add-on products such Accelerate, IOPRO and MKL Optimizations Johansson (2014).

- WinPython: A portable distribution focused on scientific applications only available for Windows. WinPython Python package incorporates two utilities: WinPython Package Manager (WPPM), a graphical interface to pip which lets you install or uninstall packages; WinPython build toolchain, containing make.py file which is the script used to build WinPython distribution almost from scratch.

EPD and Anaconda are also available for Linux and Max OS X.

1.5. SCIENTIFIC PYTHON SOFTWARE SET

1.5.1. Python Environments

In scientific applications, it is useful to separate projects in so-called environments, with different advantages and with suitability for different workflows. Because Python is not only a programming language but also include the interpreter (CPython), different environments can be constructed in versatile and complementary ways. This concept may be a bit confusing for beginners but with proper attention one can make full use of this advantage of Python framework, specifically when using for scientific computing.

1.5.2. Python Interpreter

Python interpreter is a program used to read and execute python code. The code may be directly input in the interpreter or may be stored in files, in which case the name of the file is passed to the interpreter as an argument. To run the interpreter, simply type “python” (without the double quotes) in the command prompt.

```
$ python
```

To run a file code, type python followed by the name of the file, such as:

```
$ python name_of_file.py
```

The interpreter can be used to solve simple problems, but when complexity grows it becomes quite limited, requiring one to write a program (a .py file).

1.5.3. IDLE

According Python official website, IDLE stands for Integrated Development and Learning Environment. It is integrated because it actually comes together with Python programming language when downloading it from

Python.org. It was developed completely in Python using the GUI toolkit tkinter. IDLE can be used in multi platforms, working mostly the same on Windows, Unix and Mac OS X. It contains an interactive interpreting window (each command can be run real-time by typing it and pressing Enter) with code color, input, output and error messages. It can be used in multi-windows mode with multiple undo possibility, code colorizing, smart indentation, call tips, auto-completion, and other features. IDLE's debugger can be used with persistent breakpoints, stepping, and viewing of global and local namespaces. Additionally, it can be expanded to other functionalities through the installation of add-ons.

1.5.4. Jupyter Notebook

Jupyter, a concatenation of **Julia**, **Python** and **R**, is an open-source web (HTML) application used to create articles or documentation containing live code, equations, visualizations and text. In addition to Machine Learning, it can be used for data cleaning and transformation, numerical simulation, statistical modeling and data visualization, among other uses.

Some of its features are The Jupyter Notebook (2019): + Support for multiple languages: over 40 programming languages (Julia, R, Python, Scala, etc.); + Interactive output: HTML, images, videos, LaTeX and custom MIME types can be created on a live file. + Big data integration: Incorporation with Apache Spark, pandas, scikit-learn, ggplot2, TensorFlow among other frameworks.

The interface of a Jupyter Notebook is similar to Mathematica or Maple, but providing a cell-based environment with high interactivity. Code cells can be intercalated with text and images so the document can be well-structured.

A local server is used to run the notebook from the web browser, so there is no necessity of internet connection. To start a Jupyter notebook session, open the command prompt and run the following command:

```
$ jupyter notebook
```

From the directory where you want to store the notebook. Using this command will open a new browser window (or tab) with a page showing the existing notebooks on the folder, navigable to sublevels of directory and with tools for the creation of new notebooks.

1.5.5. Spyder

According Johansson (2014), Spyder is an IDE for scientific computing with similarities to MATLAB user interface. It incorporates many advantages of traditional IDE environments, such as a single view for code editing, execution and debugging through an arrangement of all windows. Different calculations or problems can be organized as projects in the IDE environment. The author mentions the following advantages of Spyder IDE:

- + Powerful code editor with syntax highlighting, dynamic code introspection and incorporation of python debugger.
- + Console window, variable editor and history window.
- + Documentation is completely integrated with the IDE, as well as help.

1.5.6. Python Program Files

Any code written in Python can be stored in a file with .py extension. A program may be built from one single .py file, or from a collection of files, maybe even stored in different directories depending on the organizational level required. For instance, a file may be named:

myprogram.py

Each line of code is considered a Python statement, or part thereof. The only exception to this are comments, which are lines starting with the # character, preceded or not by spaces, tabs or white-space characters). Comments are for human interpretation of the code and are completely ignored by Python interpreter.

A program can be run at the command prompt by issuing the following command:

```
$ python myprogram.py
```

The standard character encoding for Python files is ASCII. This does not mean that this is the only encoding possible. Conversion to UTF-8 for instance can be done by using the following special line at the beginning (top) of a program file.

```
# -*- coding: UTF-8 -*-
```

The following code snippet illustrates a classical example of a first program written in Python. Let's call it myprogram.py. The code is:

```
# -*- coding: UTF-8 -*-
```

```
print("Hello World")
```

Running this program in a command prompt will simply print the expression “Hello World,” as requested in the command `print()`. Though useless, this illustrates the simplicity and directness of writing a program using Python.

1.6. MODULES

Any Python program file (.py) can be correctly called module as per Python standards. However, in this section we refer to Modules as the set of functionalities available for Python through the program files available as packages. For instance, Python Standard Library contains a great set of modules for cross-platform implementation of functionalities such as operating system commands, file input or output, string management, network communication among others.

To use one module, you need to import it. For instance, to use the `math` module, which contains a consistent collection of standard mathematical functionalities, type the following command on a .py file.

```
import math
```

As an example, one can create a program that calculates the sin of a certain angle by typing the following program:

```
import math  
sin_angle = math.sin(math.pi/2)  
print(sin_angle)
```

Notice that `math.sin` consists in the sine function contained in the `math` module, and `math.pi` contains the value of π stored in the `math` module. Another option would be to explicitly import all functionalities of `math` module and use them without the need of referring to the module at each call of a command contained on the module.

```
from math import *  
sin_angle = sin(pi/2)  
print(sin_angle)
```

Here it is no more explicit that the function `sin` and `pi` are both functionalities contained in the `math` import. Therefore, this syntax though it may make the code more compact it also removes the benefit of knowing from where each functionality came from. In large problems, where many modules may be imported, it can become confusing to know if certain

function was created at the program, or if it was imported through a certain module, thus removing namespace explicit mentioning. Therefore, for relatively big problems it is recommended to keep the namespace syntax to avoid confusion and namespace collisions.

Another approach is to explicitly import only the functionalities being used from a module. This syntax allows the simplification of the code as demonstrated but still it slightly preserves some reference to namespace, which may or may not minimize confusion and namespace collision.

```
from math import sin,pi  
sin_angle = sin(pi/2)  
print(sin_angle)
```

1.7. NOTEBOOKS

According Jupyter Notebook Quick Start Guide, notebooks are documents generated by the Jupyter Notebook APP. The latter is a server-client application that allows the edition and execution of notebooks through a web browser. No internet is required since Jupyter runs locally (local server) or it can be used through the internet if installed in a remote server.

Notebooks may contain code in a variety of languages, regular text (such as this one you are reading), equations, figures, links, tables, etc. Code and rich text elements are separated throughout a notebook by cells, which is the reason why this environment is called cell-based. They are not only pieces of code to be run by a computer, but human-readable documents incorporating both the description of an analysis and the results, discussion and almost any other element that one could find a regular document, but with the peculiarity of containing live code and its output (if the notebook was executed). These features make notebook as very useful tool for data analysis and Machine Learning.

Jupyter notebook app can be started by running the following command in Anaconda prompt or in a Terminal.

```
$ jupyter notebook
```

This command opens a new browser window (or tab depending on the default browser or your computer) and shows a Notebook Dashboard. It is sort of a control panel showing local files and notebooks allowing user to choose one or to shut down their kernels. A kernel is a computational engine responsible for executing the code stored in a notebook. Each kernel

executes a certain programming language, and a language may have different kernels. For instance, according Jupyter notebook app documentation, there are currently (2009) more than 20 kernels for Python, each of them with certain peculiarities. There are also kernels for Java, C++, Lua, Scilab, Perl among others.

Whenever a new notebook is created or a notebook is loaded (if it wasn't already loaded in memory), the associated kernel is also automatically launched. It is the piece responsible for performing the computations at each cell and to produce the result. The amount of CPU and RAM consumed will depend on the type of computations performed. It is important to note that RAM is not released until kernel is shutdown.

1.8. VARIABLES AND TYPES

Variables are used to store information during program runtime. For instance, at one point the age of a person is used to calculate this birthday, and at another point to know if the person is able to retire or not. Instead of always asking for this information, it may be captured once and then stored in a variable with a simple name such as age or to a longer name such as age_of_person_121. This example shows that the name of a variable can be arbitrarily long, and it may contain letters or digits. Variables cannot start with a digit, it must begin with a letter or an underscore. Also, the use of lowercase or uppercase creates different variables. For instance age_of_person_121 is a completely different variable than Age_of_person_121, where the first case is more common than the second. Additionally, variable names must not contain spaces.

Giving a variable an illegal name results in syntax error. In the following example, each of the variable names is illegal.

121_age_of_person = 27

age\$ = 27

class = "Maths"

On the first case, the variable name begins with a digit. On the second, \$ is an illegal character (dollar sign). On the last case, the word class is used as keyword in Python.

Keywords are part of the programming language “grammar,” syntax rules and structure. Therefore, they cannot be used as variable names. These keywords are:

and as assert break class continue def del elif else except exec finally for from global if import in is lambda nonlocal not or pass raise return try while with yield True False None

A variable is always associated with its type. Because Python is a dynamically-typed language, one doesn't need to worry so much about the data type of a certain variable, as it can even change during the program without any problem. But it is useful to have an understanding of the different data types so as to use them more efficiently and effectively. Following are the basic data types in Python

Integer

```
x = 1234
```

```
print(type(x))
```

```
Out: <class 'int'>
```

Floating-point number:

```
x = 1234.0
```

```
print(type(x))
```

```
Out: <class 'float'>
```

Complex number:

```
x = 1234.0 + 1.0j
```

```
print(type(x))
```

```
Out: <class 'complex'>
```

Strings:

```
x = "Machine Learning"
```

```
print(type(x))
```

```
Out: <class 'str'>
```

Boolean Type:

```
x = True
```

```
y = False
```

```
print(type(y))
```

```
Out: <class 'bool'>
```

1.9. OPERATORS AND COMPARISON

Python supports most operators for arithmetic operations and comparisons as one would expect.

Symbol	Operation
+	(sum)
-	(subtraction)
*	(multiplication)
/	(division)
**	(power)
%	(modulus)
//	(integer division)

Examples:

Sum

x = 1.0

y = 2.0

```
print("x + y = , " x + y)
```

Out: x + y = 3.0

Subtraction

x = 1.0

y = 2.0

```
print("x - y = , " x - y)
```

Out: x - y = -1.0

Division

x = 1.0

y = 2.0

```
print("x/y = , " x/y)
```

Out: $x/y = 0.5$

Integer division

$x = 1$

$y = 2$

```
print("x//y = , "x//y)
```

Out: $x//y = 0$

Note that, beginning on Python 3.x, the `/` operator will always perform a floating-point division. However, on Python 2.x, using the `/` operator with two integers will result in an integer division, while if at least one of the operands is a floating-point than the result will be a floating-point division. For instance, $1/2 = 0.5$ (float) in Python 3.x, but $1/2 = 0$ (integer) in Python 2.x.

Boolean operators are:

- And (if both are true then it is true, otherwise is false)
- Not (the opposite, if a variable is true, then “not the variable” will be false)
- Or (if any of the variables are true, is true, otherwise is false).

Examples:

$x = \text{True}$

$y = \text{False}$

```
print(x and y)
```

Out: False

$x = \text{True}$

$y = \text{True}$

```
print(x and y)
```

Out: True

$x = \text{True}$

$y = \text{False}$

```
print(x or y)
```

Out: True

```
#-----
```

```
x = True
```

```
y = False
```

```
print(not x)
```

Out: False

```
#-----
```

2

Computing Things with Python

CONTENTS

2.1. Formatting And Printing to Screen.....	22
2.2. Lists, Dictionaries, Tuples, And Sets.....	25
2.3. Handling Files	32
2.4. Exercises.....	34
2.5. Python Statements	36
2.6. For And While Loops	38
2.7. Basic Python Operators	44
2.8. Functions.....	47

This chapter provides a general overview on the basics of Python programming language. It is assumed that the reader is already somewhat familiar with computers. This is a required knowledge so as to open files and folders, move between them, change names or write a text to a file and save it.

In the previous chapter it was introduced core data structures such numbers (integer, floating-point and complex), strings and Boolean. Here we will see how to print strings to the screen in different forms. Then more advanced structures are explored, such as lists, sets, dictionaries and tuples. We will also see how one can handle files using native Python syntax and all the concepts are brought together at the end of the chapter with a programming exercise.

2.1. FORMATTING AND PRINTING TO SCREEN

Items can be inserted into strings to print their values in the screen. For a proper visualization, string formatting is used as a more elegant approach than using commas or string concatenation. One example of the difference in string formatting and concatenating is shown in the code below.

```
name = 'John'
```

```
age = 4
```

```
# string concatenation
```

```
'Last night, ' + name + ' met a friend after ' + str(age) + ' years.'
```

```
# string formatting
```

```
f'Last night, {name} met a friend after {age} years.'
```

For Python 3.x, there are three ways to format string:

- Using % placeholder (“classical” method);
- Using the .format() string method.
- From Python 3.6, using formatted string literals, also known as f-strings.

2.1.1. Using Placeholder

The character % (called “string formatting operator”) defines a way of directly injecting a variable inside a string, the type of the variable follows the placeholder. For instance, to inject a string in another string, use %s.

```
print("This is an %s of injection." %'example')
```

Multiples items can be given to the same placeholder by gathering them together in a tuple.

```
print("This is the %s text, and %s text here." %('first','another'))
```

Variable names can also be given to the placeholder.

```
mystring = 'example'
```

```
mystring2 = 'Another'
```

```
print("%s of injection." %(mystring2, mystring))
```

There are two conversion methods: str() and repr(). The main difference between them is that repr() returns the *string representation* of the object, explicitly showing quotation marks and special characters, such as escape characters.

```
print('The name of my friend is %s.' %'John')
```

```
print('The name of my friend is %r.' %'John')
```

The name of my friend is John.

The name of my friend is 'John'.

Notice how the tab character \t character adds a tab space in the first string, and is literally printed in the second string.

```
print('This text contains %s.' %'a \tbig space')
```

```
print('This text contains %r.' %'a \tbig space')
```

This text contains a big space.

This text contains 'a \tbig space'.

The d operator is used when dealing with integers (or the integer part of the floating number with no rounding).

```
print('My weight is current %s kg.' %71.95)
```

```
print('My weight is current %d kg.' %71.95)
```

My weight is current 71.95 kg.

My weight is current 71 kg.

To work with floating-points, one should use the f operator. In this case, the operator is preceded by a number, a dot and a second number (e.g., %3.4f). The first number indicates the minimum number of characters the string should contain. If necessary, this space is padded with whitespace if the number does not occupy the complete space. The second part (e.g., .4f)

indicates how many decimal places should be printed (numbers after the decimal point).

```
print('Floating-point numbers: %5.2f' %(312.984))
```

Floating-point numbers: 312.98

```
print('Floating-point numbers: %1.0f' %(312.984))
```

Floating-point numbers: 313

```
print('Floating-point numbers: %10.2f' %(312.984))
```

Floating-point numbers: 312.98

2.1.2. Using the .format() Method

Formatting string can be improved using the .format() method. This one has several advantages over the one previously mentioned. When using multiple variables in a tuple, the objects can be called by an index position, thus enabling to “shuffle” a tuple.

```
print('The {1} {2} {0}'.format('Benz', 'red', 'Mercedes'))
```

The red Mercedes-Benz

A second approach is to associate each object with a keyword to be called.

```
print('Color: {color}, Model: {model}, Power: {type}'.format(color = 'Red', model='BMW', type = 1000))
```

Color: Red, Model: BMW, Power: 1000

The above mentioned features enable one to reuse objects, avoiding repetition and improving code readability.

```
print('{name} has a {size} {animal}. {name} likes it'.format(name = 'Mary', size = 'big', animal = 'dog'))
```

```
print('%s has a %s %s. %s likes it'%(Mary', 'big', 'dog', 'Mary'))
```

Mary has a big dog. Mary likes it

Mary has a big dog. Mary likes it

The selection of padding and precision with .format() method enables easy alignment and thus organization of multiple prints. This is specially useful when performing prints of an iterative process, or of a table.

```
print('{0:8} | {1:9}'.format('Groceries', 'Quantity'))
```

```
print('{0:8} | {1:9.1f}'.format('Milk', 3.))
```

```
print('{0:8} | {1:9.1f}'.format('Apples', 10))
```

Groceries | Quantity

Milk | 3.0

Apples | 10.0

More information on how to use `.format()` method can be found in official Python documentation: <https://docs.python.org/3/library/string.html#formatstrings>

2.1.3. Using f-string Method

An even greater improvement, f-string provides better formatting of strings over `.format()` method. The main difference is that the object can be injected directly into the main string (inside the curly brackets) without the necessity of an outer format as it is done with `.format()` method.

```
age = 45
```

```
print(f'Luke said his age is {age} years old.')
```

Luke said his age is 45 years old.

For proper formatting of padding and precision, use a colon (:) followed by the precision definition as already mentioned previously (e.g., `3.5f`).

```
num = 12.567
```

```
print(f'The temperature outside is {num:10.4f} degrees Celsius.')
```

The temperature outside is 12.5670 degrees Celsius.

Additional information on how to use f-string method can be obtained in the official Python documentation: https://docs.python.org/3/reference/lexical_analysis.html#f-strings

2.2. LISTS, DICTIONARIES, TUPLES, AND SETS

Lists, Sets, Dictionaries, and Tuples are structures that work like *containers*. They store elements inside the object. In some cases the elements may be changeable (lists or dictionaries), while in other they are not mutable (sets and tuples).

2.2.1. Lists

Lists are constructed by using a square bracket [] and putting inside the elements that it should contain. These may be numbers, strings, other lists, in summary any Python object can be contained inside a list.

```
# Assign a list to the variable mylist
```

```
mylist = [0, 1, 2]
```

Notice that each element inside the list is separated by comma. Another example consists in mixing different types of elements in the same list.

```
# Assign a list to the variable mylist
```

```
mylist = ['a string example', 1.2, [1, 2, 3]]
```

The function len() can be used to check the length of a list, i.e., the number of elements contained in the list (note that a list contained inside another list counts as one element).

```
len(mylist)
```

```
3
```

An element of the list can be retrieved using indexing. For instance, the first element of the list is stored at index 0, the second is at index 1, etc.

```
# Reference to the element 0
```

```
mylist[0]
```

```
a string example
```

One can also refer to multiple elements by using the colon (:) operator. For instance, refer to the 2nd element and all the other following using the syntax [1:]

```
# Reference to the element 1 and the ones following
```

```
mylist[1:]
```

```
[1.2, [1, 2, 3]]
```

Similarly, referring from the first element up to index 2 can be done as follows.

```
# Reference up to index 2
```

```
mylist[:2]
```

```
['a string example', 1.2]
```

Lists can be concatenated by using addition symbol (+) and assigning to a new variable (or the variable itself to modify it).

```
# Concatenate mylist with the list [4] (a list of one element) and
```

```
# assign to newlist
```

```
newlist = mylist + [4]
```

```
newlist
```

```
['a string example', 1.2, [1, 2, 3], 4]
```

A list can be duplicated or repeated n times by using the multiplication sign (*).

```
mylist*3
```

```
['a string example', 1.2, [1, 2, 3], 'a string example', 1.2, [1, 2, 3], 'a string example', 1.2, [1, 2, 3]]
```

2.2.1.1. Essential List Methods

Append – Method used to add elements in a list. Each new element is added at the end of the list (right side).

```
mylist = [1, 2, 3]
```

```
mylist.append(5.5)
```

```
mylist
```

```
[1, 2, 3, 5.5]
```

Pop – remove an element of the list according its index and assign it if required, i.e., list.pop(0) removes the 0-th indexed element.

```
mylist = [1, 2, 3]
```

```
popped_item = mylist.pop(0)
```

```
print('popped_item = ', popped_item)
```

```
print('mylist = ', mylist)
```

```
popped_item = 1
```

```
mylist = [2,3]
```

Sort – sort the elements of the list in ascending order. If it is numbers, sorts from the lowest to highest. If strings, from a to z in alphabetical order, with some different behavior if there are capital letters, symbols and letters mixed with numbers.

```
mylist = [3, 5, 1, 2]
```

```
mylist.sort()
```

```
print('mylist = ', mylist)
mylist = [1, 2, 3, 5]
mylist = ['dog', 'zeus', 'corn', 'Height']
mylist.sort()
print('mylist = ', mylist)
mylist = ['Height', 'corn', 'dog', 'zeus']
```

Matrices can be created by nesting lists. For instance, a matrix containing 2 rows and 3 columns is declared by using 2 nested lists in a superior list, each one with 3 elements.

```
mymatrix = [[1, 2, 3], [0, 1, 3]]
```

In this case, a reference to a complete row is done using one index.

```
mymatrix[0]
[1, 2, 3]
```

The selection of one element is done using two indices, the first one to which nested list (row) and the second to the element inside the nested list (column).

```
mymatrix[0][0]
1
```

2.2.2. Dictionaries

Dictionaries are data-structures similar to lists, but with *element mapping*. A mapping associates each element stored in the dictionary to a *key*, which differs it to a list, where each element is associated with an index (number). A mapping doesn't retain order, since they have objects defined by a key. Similar to the list, the element stored in a key can be any Python object, from numbers to strings and other dictionaries, lists, etc.

A dictionary is constructed by using curly brackets {} and defining each key and the value associated separated by a colon operator (:).

```
mydict = {'key1': value1, 'key2': value2, 'key3': value3}
# A value can be referred by using the key
mydict['key2']
value1
```

As already mentioned, dictionaries can hold any type of Python object, similarly to Python lists.

```
mydict = {'key1': [1, 2, 3], 'key2': 0.234, 'key3': 'a string example'}  
# Refer to the index-2 position in the list stored at key1  
mydict['key1'][2]  
3
```

An empty dictionary can be created using the keyword `dict()` or empty curly brackets `{}`.

```
# Two different ways of declaring empty dictionaries
```

```
emptydict = {}  
other_empty_dict = dict()
```

Analogous to lists, dictionaries can be nested to create a more flexible structure. Each nested key can be called in a single line of code by referring to the key where the nested dictionary is stored.

```
nesteddict = {'key1': {'nested1': {'subnested1': value1}}, 'key2': value2}
```

```
nested['key1']['nested1']['subnested1']  
value1
```

2.2.2.1. Dictionary Methods

.keys() – returns the keys of the dictionary in a list-like structure.

```
person = {'name': 'Donald', 'age': 19, 'bag': ['compass', 'notebook']}
```

```
person.keys()  
dict_keys(['name', 'age', 'bag'])
```

.values() – returns the values stored in all of the dictionary keys.

```
person = {'name': 'Donald', 'age': 19, 'bag': ['compass', 'notebook']}
```

```
person.values()  
dict_values(['Donald', 19, ['compass', 'notebook']])
```

.items() – returns tuples with key-value pairs.

```
person = {'name': 'Donald', 'age': 19, 'bag': ['compass', 'notebook']}
```

```
person.items()  
dict_items([('name', 'Donald'), ('age', 19), ('bag', ['compass', 'notebook'])])
```

2.2.3. Tuples

Tuples can be easily related to lists, with the exception that is *immutable*, i.e., their elements cannot be changed, it cannot be appended or popped. It is normally used to present data which is not supposed to be changing, such as calendar days, days of week, etc.

Tuples are built using round parenthesis (), and separating elements with commas.

```
mytuple = (4,5, 6)
```

Similar to list, the size of a tuple can be verified with the len() function. A tuple can be created with elements of different data types, and indexing works in the same ways as lists.

```
mytuple = ('string example', [5.5, 1, 2], 6)
print('size of tuple = ', len(mytuple))
print('0-th index element = ', mytuple[0])
size of tuple = 3
0-th index element = string example
```

2.2.3.1. Tuple Methods

.index() – locate the index of a certain element in the tuple.

```
mytuple = ('string example', [5.5, 1, 2], 6)
```

```
mytuple.index('string example')
```

```
0
```

```
mytuple = ('string example', [5.5, 1, 2], 6)
```

```
mytuple.index(1)
```

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: tuple.index(x): x not in tuple
```

Notice that try to locate the index of an element not found in the tuple (the number 1 is part of the inner list but is not a single number being stored in the tuple) causes an error.

.count() – counts the number of times an element appears in the tuple.

```
mytuple = (6, [5.5, 1, 2], 6)
```

```
# Notice that the number 6 appears two times in the tuple above
```

```
mytuple.count(6)
```

```
2
```

2.2.3.2. *Immutability*

As already mentioned, tuples are *immutable* objects, i.e., they cannot be changes at any way. This includes not only adding or removing elements, but also changing one or more elements already stored in the tuple.

For instance, trying to set one element of the tuple gives an error.

```
mytuple[1] = 3.2
```

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

Trying to append or pop an element of the tuple also causes an error.

```
mytuple.append(3.2)
```

Traceback (most recent call last):

```
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

2.2.4. Sets

A set is a special data container in Python with contains only *unique* elements, i.e., there is no repetition of elements in a set. It can be constructed using `set()`.

```
myset = set()
```

Elements are added to a set using the `.add()` method. Trying to add more than once the same element does not issue an error, but it is not added, since a set can only contain unique elements.

```
# add the number 0 to the set
```

```
myset.add(0)
```

```
print(myset)
```

```
# add the number 10 to the set
```

```
myset.add(10)
```

```
print(myset)
```

```
# trying add the number 0 again to the set won't add it
myset.add(0)
print(myset)
{0}
{0, 10}
{0, 10}
```

A set can be a convenient tool to select the unique elements on a list.

```
list1 = [1, 2, 1, 7, 8, 9, 5, 6, 4, 5]
set(list1)
{1, 2, 4, 5, 6, 7, 8, 9}
```

2.3. HANDLING FILES

A file consists in a hard copy of information stored in the computer hard-disk. This information can be formatted in a variety of ways: as an excel file (spreadsheet), as a word file, plain text (.txt) and even other formats. Note that a file is not restrict to data which can be read. It may also be video or audio. For specific data types, Python requires certain libraries. For basic file types, Python has native functionality that enables one to open, write, read and save a file.

To open a file, use the function `open(filename)` where `filename` is the name of the file to be opened. The file handler should be assigned to an object so it can be safely closed after performing the necessary operations.

```
myfile = open('sample.txt')
```

Notice that the file to be opened (in his case “`sample.txt`”) must lie in the same folder of the script, otherwise an error is raised. For a file stored in a subfolder, refer to the relative or complete path to find the file.

```
# Relative path in a subfolder
myfile = open('.\\subfolder\\sample.txt')
```

```
# Relative path in a parent folder
myfile = open('..\\sample.txt')
```

```
# Absolute path  
myfile = open('C:\\Data\\sample.txt')
```

It is good practice to use double backslash so python does not recognize as an escape character.

File reading can be done through the `read()` method, which reads the entire content of the file and stores it as a string.

```
myfile.read()
```

The content of the file is here

Trying to read again will yield no result, since the cursos has arrived at the end of the file.

```
myfile.read()
```

'''

To go back to the beginning of the file, use the `seek(pos)` method, where pos is an integer indicating the position to place the cursos (0 is the beginning of the file). To read each line and store it in a list, use the `readlines()` method instead of `read()`.

```
myfile.seek(0)
```

```
myfile.readlines()
```

[‘The content of the file is here’]

Always remember, when finished to work on the file, to close it and release memory using the `close()` method.

```
myfile.close()
```

To open a file and write on it, use the flag `w` inside the `open(filename,flag)`. Note that using such flag removes all the content of the file. To append to the end of the file (instead of cleaning it), use the `a` flag. On the other hand, for reading (default), one can explicitly use the `r` flag.

open the file in write mode. All the previous content is erased.

```
myfile = open('sample.txt', 'w')
```

open the file in append mode. The content is preserved and new content is appended to the end of the file.

```
myfile = open('sample.txt', 'a')
```

```
# explicitly open the file in read mode  
myfile = open('sample.txt', 'r')  
# alternatively  
myfile = open('sample.txt')
```

Another way of working with files is by using the with block. In this way, the file only stays open inside the block, and it is closed automatically once the program exits the with block.

open the file in read (default) mode and name the handler as f.

```
with open('sample.txt') as f:
```

iterate through each line of the file

```
for line in f:
```

```
    print(line) # print the complete line
```

var = 2 # here the file f is already closed since the program exits the with block``

2.4. EXERCISES

- 1) Using Python code, write an equation that uses addition and subtraction that is equal to -80.12
- 2) Write another equation that, using multiplication, division and exponentiation, is equal to -80.12
- 3) Consider the equation $4 + 0.5 + 2$. What is the type?
- 4) How can one determine the square root of a number, say 9?
- 5) How can one determine the square of a number, say 2?
- 6) Consider the string “Mary.” How can one choose the letter “a” from this string?
- 7) Through slicing, reverse the string “Mary.”
- 8) Show 2 different methods of printing the letter “y” from “Mary.”
- 9) Show 2 different methods of generating the list [1, 1, 1].
- 10) Change the last element of the list [1, 1, 1] to be equal 10.
- 11) Reverse the list [1, 2, 3, 4]
- 12) Sort the list [1, 4, 2, 3]
- 13) Let mydict = {‘name’: ‘Julius’, ‘age’: 20, ‘hobby’:’painting’} Through key indexing, print the element stored in the key hobby.

- 14) Do the same as above for the following object mydict = {'name': {'firstName': 'Julius', 'lastName': 'Smith'}, 'habits': {'work': 'lawyer', 'hobby': 'painting'}}
- 15) Print the word "bye" from the following dictionary newdict = {'key1': [{'nested_key': ['going further', ['bye']] }]}
- 16) Can a dictionary be sorted?
- 17) What is the main difference between tuples and lists?
- 18) Use a proper data structure to select only the unique elements of the list list2sort = [100, 90, 55, 80, 65, 73, 20]
- 19) State the Boolean result of the following code: $3 < 2$.

2.4.1. Solutions

- 1) There are infinite possibilities to solve this problem. One possible is $-20.06 - 70.06 + 10..$
- 2) There are infinite possibilities to solve this problem. One possible is $-1201.8/30^2$.
- 3) float
- 4) Using fraction power $3^{**}(1/2)$ or $3^{**}0.5$
- 5) Using power as $2^{**}2$
- 6) Mary[1] or mystring = 'Mary'; mystring[1]
- 7) 'Mary'[::−1] or mystring = 'Mary'; mystring[::−1]
- 8) Consider mystring = 'Mary'. Method 1: mystring[3] or Method 2: mystring[−1]
- 9) Method 1: [1, 1, 1] or Method 2: [1] * 3
- 10) mylist = [1, 1, 1]; mylist[−1] = 10
- 11) mylist = [1, 2, 3, 4]; mylist.reverse()
- 12) mylist = [1, 4, 2, 3]; mylist.sort()
- 13) mydict['hobby']
- 14) mydict['habits']['hobby']
- 15) newdict['key1'][0]['nested_key'][1]
- 16) A dictionary cannot be sorted since it consists in a mapping of key-value and not a sequence.
- 17) A tuple is immutable, a list is mutable.
- 18) set(list2sort)
- 19) False

2.5. PYTHON STATEMENTS

In many programs, logic requires comparisons and evaluation of condition in order to run a certain part of the code or not. This logical evaluation and blocks of code are constructed using Python statements. In such blocks, the indentation is very important, it defines the limits of the blocks. For instance, **if** (condition):

code **in** block

code **in** block

elif (condition2):

code **in** block

code **in** block

else:

code **in** block

code **in** block

code outside the block

Yet another example, more real code:

if True:

print('This goes inside the if statement.')

This goes inside the if statement.

x = 3 < 2 # This evaluates to False

if x == False:

print('This is printed since x is False.')

This is printed since x is False.

This is an example using an else block. This block is run whenever the if block evaluates to false.

x = 3 < 2 # This evaluates to False

```
if x:  
    print('This is printed if x is True.')  
else:  
    print('This is printed if x is False.')  
This is printed if x is False.
```

The elif block can be used following an if block to add other conditions to be evaluated, rather than only using the else block directly.

```
state = 'Alabama'
```

```
if state == 'California':  
    print('The state is California')  
elif state == 'Alabama':  
    print('The state is Alabama.')  
else:  
    print('The state is any other but not California or Alabama.')  
The state is Alabama.
```

One can use as many elif as it may be necessary in an if block. The first block to be evaluated to True will be run, and the code will progress from the code following the block (all other elif or else will be neglected).

```
number = 10
```

```
if number/2 == 3: # This evaluates to False (10/2 = 5)  
    print('The state is California')  
elif number/2 == 5: # This evaluates to True (10/2 = 5) and runs  
    print('The state is Alabama.')  
elif number*2 == 20: # This also evaluates to True (10/2 = 5) but it is not  
run because of the one above  
    print('The state is New York.')  
else:  
    print('The state is any other but not California, Alabama or New York.')  
The state is Alabama.
```

2.6. FOR AND WHILE LOOPS

A for loop consists in a way of iterating through an object (the object is must be iterable) and perform a certain operation. For example, consider the list [1,2,10,4]. At a for loop, each run will gather an instance inside the object (the instances, in this case, are the numbers 1, 2, 10 and 4) and perform certain operation. These operations do not directly change the instance unless explicitly defined. The general form of the for loop is as follows.

for item in iterator:

code goes here

The variable name item must be chosen, so proper judgment should be used in order to select a name with proper meaning, which will ease the process of code revision.

The for loop can be used to iterate on a list.

```
mylist = [1, 2, 3, 4]
```

for item in mylist:

print(item)

```
1  
2  
3  
4
```

Consider another example, where an if is used to filter which items will be printed from the list above.

```
mylist = [1, 2, 3, 4]
```

for item in mylist:

if item – 2 > 1:

print(item)

```
4
```

```
mylist = ['dog', 'cat', 'dolphin', 'chicken', 'tiger']
```

for item in mylist:

if item.startswith('d'): # startswith(x) check if string starts with x

print(item + ' starts with d.')

elif item.startswith('c'):

print(item + ' starts with c.')

```
else:
```

```
    print(item + ' start with any other letter than d or c.')
```

```
dog starts with d.
```

```
cat starts with c.
```

```
dolphin starts with d.
```

```
chicken starts with c.
```

```
tiger start with any other letter than d or c.
```

A for loop can be used to create a cumulative sum or cumulative multiplication.

```
mylist = list(range(100))
```

```
cum_sum = 0
```

```
for item in mylist:
```

```
    cum_sum += item
```

```
cum_sum
```

```
4950
```

A for loop can also be used to access each character of a string, since a string is actually a sequence of characters.

```
mystring = 'The dog is black'
```

```
for item in mystring:
```

```
    print(item)
```

```
T
```

```
h
```

```
e
```

```
d
```

```
o
```

```
g
```

```
i
```

```
s
```

```
b  
l  
a  
c  
k
```

When a sequence is made of elements which can also be iterable (such as tuples) the for loop can be used to unpack such element. For instance, consider the following list made of tuples containing pairs of numbers.

```
mylist = [(1, 2), (5, 6), (9, 10), (20, 30)]
```

In the following example, a for loop is used to iterate through each element of the list (tuples), without additional unpacking.

```
for elem in mylist:
```

```
    print(elem)  
(1, 2)  
(5, 6)  
(9, 10)  
(20, 30)
```

Consider how unpacking can be used to separate the elements inside the tuple pair in each instance inside the list.

```
for (t1, t2) in mylist:
```

```
    print(t2)  
2  
6  
10  
30
```

A loop can be used to iterate over a dictionary object. In this case, the instances returned are the dictionary keys.

```
mydict = {'key1': 'dog', 'key2': 'cat', 'key3': 'monkey'}
```

```
for item in mydict:  
    print(item)
```

```
key1  
key2  
key3
```

Note that the keys will not necessarily be returned in a sorted order. To retrieve the values stored on each key, use the `.values()` method.

```
for item in mydict.values():  
    print(item)  
dog  
cat  
monkey
```

To retrieve both the keys and the values, dictionary unpacking can be done using the `.items()` method.

```
for key_, value_ in mydict.items():  
    print(key_)  
    print(value_)  
key1  
dog  
key2  
cat  
key3  
monkey
```

A while loop is similar to a for loop, with the logic difference that the code inside the while block will run as long as a certain condition is True. This type of block must be handled carefully, since infinite loops may be generated if the condition to exit is never changed to False.

```
while (condition):  
    run code  
else:  
    final code
```

Consider the following example, where a variable condition is initialized to True, and an auxiliary variable `x` is continuously changed. Once the

variable x reaches the value 10, then the condition is changed to False in order to exit the while loop.

```
condition = True
```

```
x = 0
```

while condition:

```
    print(`The value of x is `, x)
```

```
    if x >= 10:
```

```
        condition = False
```

```
        x += 1
```

The value of x is 0

The value of x is 1

The value of x is 2

The value of x is 3

The value of x is 4

The value of x is 5

The value of x is 6

The value of x is 7

The value of x is 8

The value of x is 9

The value of x is 10

The commands break, continue and pass can be used to add functionality and deal with specific cases. The break statement exits the current closest loop.

```
x = 0
```

while True:

```
    print(`The value of x is `, x)
```

```
    if x >= 10:
```

```
        break
```

```
        x += 1
```

The value of x is 0

```
The value of x is 1
The value of x is 2
The value of x is 3
The value of x is 4
The value of x is 5
The value of x is 6
The value of x is 7
The value of x is 8
The value of x is 9
The value of x is 10
```

The continue statement returns to the beginning of the current closest loop.

```
x = 0
```

while True:

```
    print('The value of x is ', x)
    x += 1
    if x%2 == 0:
        continue
    if x >= 10:
        break
```

```
The value of x is 0
The value of x is 1
The value of x is 2
The value of x is 3
The value of x is 4
The value of x is 5
The value of x is 6
The value of x is 7
The value of x is 8
```

The value of x is 9

The value of x is 10

The pass statement do not perform any specific operation. Instead, it is used as a placeholder in a code block since a code block in Python must contain at least one line of code.

```
x = 0
```

```
while True:
```

```
    print('The value of x is ', x)
```

```
    x += 1
```

```
    if x%2 == 0:
```

```
        pass
```

```
    else:
```

```
        x += 1
```

```
    if x >= 10:
```

```
        break
```

The value of x is 0

The value of x is 2

The value of x is 4

The value of x is 6

The value of x is 8

2.7. BASIC PYTHON OPERATORS

There are some very useful operators in Python which can perform certain operations and simplify programming tasks. There are: + range + enumerate + zip + in operator + input function

- range

A generator function. used to create a sequence of integers. Its syntax is
`range(start, stop, step)`

Where start is the first number, stop is the last number of the sequence, and step is the distance between each number.

Since range is a generator, it must be used with the list command in order to generate a sequence list.

```
list(range(0, 10, 2))
```

```
[0, 2, 4, 6, 8]
```

Note that the last number (10) do not appear in the sequence. That is because the stop number is *exclusive* and not *inclusive*. When the step argument is omitted, than it is assumed to be equal to 1.

```
list(range(0, 10))
```

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

- enumerate

The enumerate function performs unpacking of a list, retrieving the index and the element value so one can iterate over it.

```
mylist = [10, 20, 30, 40]
```

```
for index_, value_ in enumerate(mylist):
```

```
    print(f'The value stored in index {index_} is {value_}')
```

```
The value stored in index 0 is 10
```

```
The value stored in index 1 is 20
```

```
The value stored in index 2 is 30
```

```
The value stored in index 3 is 40
```

Similarly with what is done on list, enumerate can be used to unpack strings, retrieving the position of each character and the value of it.

```
animal = 'elephant'
```

```
for index_, value_ in enumerate(animal):
```

```
    print(f'The value stored in index {index_} is {value_}')
```

```
The value stored in index 0 is e
```

```
The value stored in index 1 is l
```

```
The value stored in index 2 is e
```

```
The value stored in index 3 is p
```

```
The value stored in index 4 is h
```

```
The value stored in index 5 is a
```

```
The value stored in index 6 is n
```

```
The value stored in index 7 is t
```

- zip

This method helps to iterate over two lists at the same time, by “zipping” or packing them together, as if one was the index and the other was the value when using the enumerate function.

```
list1 = [10, 20, 30, 40]
```

```
list2 = ['a', 'b', 'c', 'd']
```

```
list(zip(list1, list2))  
[(10, 'a'), (20, 'b'), (30, 'c'), (40, 'd')]
```

Note that both lists should have the same size. Extra elements on the larger list will be ignored when performing zip function.

```
list1 = [10, 20, 30, 40, 50] # this list has 5 elements (50 is the last one)
```

```
list2 = ['a', 'b', 'c', 'd'] # this list has 4 elements
```

when doing zip, the number 50 in list1 is ignored since it is considered an extra element.

```
list(zip(list1, list2))  
[(10, 'a'), (20, 'b'), (30, 'c'), (40, 'd')]
```

The zip generator can be used in a for loop, similarly to the enumerate, to iterate over two lists at the same time.

```
list1 = [10, 20, 30, 40, 50] # this list has 5 elements (50 is the last one)
```

```
list2 = ['a', 'b', 'c', 'd'] # this list has 4 elements
```

when doing zip, the number 50 in list1 is ignored since it is considered an extra element.

```
for value1, value2 in zip(list1, list2):
```

```
    print(f'The value extracted from list1 is {value1}, and from list2 is {value2}.')
```

The value extracted from list1 is 10, and from list2 is a.

The value extracted from list1 is 20, and from list2 is b.

The value extracted from list1 is 30, and from list2 is c.

The value extracted from list1 is 40, and from list2 is d.

- in operator

The `in` operator can be used to quickly verify if a certain value is contained in a list.

```
y = 1
mylist = [1, 2, 3, 4, 5]
print('Is y contained in mylist? ', y in mylist)
Is y contained in mylist? True
y = 'y'
mylist = [1, 2, 3, 4, 5]
print('Is y contained in mylist? ', y in mylist)
Is y contained in mylist? False
```

- `input`

The `input` command is used to capture an input from the user when running the program. It waits such input before continuing to run. The input is stored as a string, and may be converted to other types by using `int()` or `float()` to convert into integer or floating-point number, respectively.

```
name = input('What is your name? ')
print('Welcome ' + name + '!')
age = input('What is your age? ')
future_age = int(age) + 10
print('In 10 years you will be ' + str(future_age) + ' years old.')
What is your name? John
Welcome John!
What is your age? 20
In 10 years you will be 30 years old.
```

2.8. FUNCTIONS

Similar to mathematics, functions are a set of instruction used to perform a certain operation and return something (though in Python it will not always return something explicitly). It is specially useful for commands which are supposed to be reused, i.e., they will be used multiple times. In this way, a function avoids repetition and improves readability of code.

The blueprint of a function takes the following general format.

```
def function_name(input1, input2, inputn):
```

```
    ""
```

Function Document String goes here

```
    ""
```

code goes here

```
    return something_to_return
```

As it can be seen in the template above, a function is created by using the keyword def followed by the given function_name. It is important to give a meaningful name to the function so one knows what is its purpose. Then the function name is followed by one or more inputs, i.e., argument(s) that are given to the function so it can perform the necessary operations.

For good programming practice, a function document string should follow the definition of the function, stating the functionality and the arguments that the function receives (with data types). The body of the function consists of the code performing the necessary operations, and usually at the end the return keyword defines which variable(s) are to be returned by the function to be used in the global namespace.

```
def add(x1, x2):
```

```
    ""
```

add(x1, x2)

This function is used to add two variables and return the sum of them.

Inputs:

x1 – float number

x2 – float number

Return:

y – the result of adding x1 and x2

```
    ""
```

y = 0

check if x1 and x2 are float or integers and perform addition, otherwise print a message

```
if (type(x1) == float or type(x1) == int) and (type(x2) == float or type(x2) == int):
```

y = x1 + x2

```
else:  
    print('Wrong data type.')
```

```
return y
```

```
result = add(9, 10)
```

```
print(result)
```

```
19
```

A function may not return explicitly anything. For instance, the following example contains a function which prints a statement at the screen. Notice that no value is returned (the keyword `return` is not used).

```
def print_animal_size(animal, size):
```

```
    ""
```

```
    print_animal_size(animal,size):
```

This function prints a sentence on the screen evaluating if the animal is more than 1 meter tall.

Inputs:

animal(string) – animal name;

size(float) – the animal size in meters

```
    ""
```

```
if size >= 1:
```

```
    print(f'The {animal} size is greater than or equal to 1 meter.')
```

```
else:
```

```
    print(f'The {animal} size is less than 1 meter.')
```

```
print_animal_size('leopard', 0.8)
```

```
print_animal_size('elephant', 3.2)
```

The leopard size is less than 1 meter.

The elephant size is greater than or equal to 1 meter.

3

A General Outlook on Machine Learning

This chapter will give the reader a general notion of what Machine Learning is for, what type of problems it can generally solve. Here we do not stick to the formalism inherent of a mathematical field. Rather the intention of this chapter is to instigate user curiosity and to give him a general idea of the concepts used on Machine Learning to solve everyday problems. To make it more trivial, we insert a character called Paul, an regular guy with some knowledge in programming but nothing special, ordinary intelligence but with a curiosity on how to automate things.

Paul wakes up, 6:00 am and it is already a beautiful day outside his small apartment in Amsterdam. As usual he grabs a cup of hot coffee with some bread and check his e-mail before leaving to work. His inbox contains the following list of e-mails:

Table 3.1: E-mail List

Sender	Subject
Lottery1000	You are out 1000th winner!
Amazing Credit Card	Sign up now and start shopping!
Adams, Mary	Meeting with clients from Japan
Se Manda Cheques	Ahora puedes nos contactar
Cordell Jean	Les nouvelles qui vous intéressent
Hill, John	Request for document

Looking at these first e-mails, Paul takes a deep breath and starts his morning process of cleaning his inbox, checking the important e-mails and

scheduling appointments as necessary. Almost everyday is like this, and today is not different, from the six new e-mails shown above, only two of them are relevant. Paul sees the first one, Lottery1000, he doesn't even bet on the lottery. And the e-mail says he is the 1000th winner! Too classical spam! Paul hover his mouse over the delete button and remove this e-mail from his inbox. The second one, Amazing Credit Card, well Paul didn't ask for any credit card so this is another spam, delete it. The third one, Paul knows Mary Adams, she is a coworker at his company of bank consultancy and apparently, she would like to discuss something about a meeting with clients from Japan. This is an important e-mail so shouldn't be deleted but checked. But before doing so, Paul goes to the next e-mail and over the entire list of e-mails on his inbox to remove all the spams. After doing so, Paul looks at his watch and notices that he has wasted an appreciable amount of time just checking if an email is spam or not! What a waste of time!

Paul stops to reflect a bit and think how much time he wastes every morning and sometimes even his afternoon just to remove spams from his inbox. Though it is a relatively simple task, of just looking and removing those not requested, it is a time he could invest in something much more meaningful. How nice it would be if someone could classify the spams and remove them instead of Paul himself! Meditating, he thinks:

“Well, I could hire someone to do this task for me!”

But thinking a little bit more, Paul sees that it doesn't worth, since he would have to go over the entire process of announcing the job, interviewing the people, training them so they could have a good grasp of what is spam and what isn't in Paul's inbox, and then pay this people in a monthly basis. Paul salaries are not so high, so he just thinks his money can have a better use.

So, if instead of hiring a person, Paul uses a machine, more specifically his own computer to do this task? It would require effort once to write a set of instructions to the computer in a programming language, and then the computer could do all the spam classification even while Paul would be taking some breakfast. Perfect idea! But how to do so? How can the computer know what is spam or what is not?

First of all, the e-mails will be classified as “spam” or “not spam.” Since computers does not understand words, but numbers, Paul needs to translate these labels to numbers. A computer can understand 0, 1, 2, 3,... binaries or numbers, etc. So, the problem of being a spam or not can be translated to 0 (not spam) or 1 (spam). “Great!” shouts Paul to himself while reclining

at his seat, just before he notices how delayed he is to his work, so almost falling he sprints to the shower and leave his home almost as a thunder.

When he arrives at his job, Mary Adams (the one of the e-mail, we will call her Mary from here now) approaches Paul and kindly ask:

“Hey Paul good morning! Have you read the email I sent you?”

Paul asks:

“What e-mail?”

And almost the same time as he is speaking, he remembers the e-mail Mary sent him, but he didn't have time to check because he was too involved in removing the spams. At this moment, Mary gets visibly angry but trying not so much to control herself she says:

“Well you should have received it! These clients from Japan are very valuable and they have a problem that needs a solution now! Read the e-mail and give me a position as soon as possible!”

“Ok...”

Paul arrives at this office with another cup of coffee, opens the laptop and checks Mary's e-mail.

“Dear collaborators,

A representative of our client in china, Xing Bank, is visiting us this week and he has a request and is willing to offer a fine payment to have an automated system of credit card approval. At the current moment, this system consists in the following steps:

- A client who wants a credit card from Xing Bank must fill a form at one of our institutions with information about his employment situation, salary level, age, etc.
- This form is sent online to a set of experts who collects additional information of the client such as financial history, relationship with other banks, etc.
- The experts analyze all this information and come up with the result: approval or disapproval of credit card submission.
- The result is written in the form and sent back to the head office for emission of the card. An e-mail is sent to the client stating the situation (approval or disapproval).

According the representative of Xing Bank, the entire process from

client request submission to result takes approximately 5–6 days. However Xing Bank is a growing business and it has open some new branches in China. With that the number of clients has raised appreciatively, and this actual process of credit card approval became unusable. They asked if we can come up with a solution, and they will provide additional information if necessary.

I am sure that you can help, so please contact me as soon as you have thought on a solution.

Regards,

Mary Adams”

After reading the e-mail, Paul starts to think: “It is interesting that they have this problem of receiving some credit card request and saying that some are good and some are not. It is as if those who are not good are like spam e-mails on their inbox. So they need an automatic system to classify their requests just as I need an automatic system to classify my e-mails!”

Thinking that he is starting to become crazy with this automation stuff, Paul stops for a second and has a drink of coffee. If an analytical eye on problem, he grasp the keyword of this problem: Classification! It is all about classifying things, through certain criteria or information.

Paul remembers when he was a child and he used to go to his uncle’s house. That man had a lot of cats and dogs at that time. Almost every time Paul went there, a new animal was there. How he new if that animal was a cat or a dog? Through classification. For example, did the animal mew or bark? Was it big or small? Ears falling or standing? In his mind, a set of criteria collected by the brain would be used to determine if the animal was a dog or a cat, in similar way the brain sees if an e-mail is a spam or if a team approves or disapproves credit cards.

For instance, once Paul visited his uncle and he had a new animal. At first look, Paul could see that the animal was relatively small, with a standing ear and when it emitted a sound, it was a meow. So then he knew it was a cat. Other times the animal was relatively big, with falling ears and it barked. Then it was a dog. Naturally, some characteristics were not fixed to ALL dogs or cats, as some dogs would have standing ears and some cats would be higher than the majority. Still Paul could easily classify them into cats and dogs because his brain was TRAINED, i.e., it has already seen lots of dogs or cats, so it would be difficult to find one which he wouldn’t know if it is a dog or a cat or worse, that he would classify it wrong!

If the characteristics of dogs and cats were to be read by a machine instead of the brain, in order to classify the animal into dog or cat, how it would be done? Well, in the same way as spam or not spam was transformed to 1 and 0, so the characteristics can be transformed to 1 and 0:

- Bark or don't bark: 1 or 0
- High or not high: 1 or 0
- Falling Ears or not falling ears: 1 or 0.

So according these three categories, if the animal has the majority of them 1, then it is a dog (1), otherwise it is a cat (0).

Comparing this classification with the e-mail spam system Paul wanted to do, he thinks on how he could think characteristics to determine if an e-mail is a spam or not in his inbox. He usually does not sign for anything which is not English or Dutch, so any other language may be spam. Also he does not play lottery, so any email with words related to this would be spam. Additionally he does not sign newsletters for discounts or shopping, so he already have some terms to classify the e-mails as he used to classify dogs and cats.

- English or Dutch (0) or any other language (1)
- Has the term lottery (0) or it does not have (1)
- Has the term discounts or shopping (0) or if it does not have (1)

With these three criteria, he would give it to the computer and tell it: “Ok, now you have some criteria to classify my e-mails, if they fall on all of them then it is a spam.” On this way, it would not be very efficient, because the machine would only classify as spam those e-mail which are not written in English or Dutch AND contains the term lottery AND the term discounts or shopping, which are not very common. On the other hand, if he says that: “Ok, now if the e-mail falls on any of them then it is a spam,” and a friend send an invitation to go to a shopping, then the machine would classify as spam. Thus, the problem is a little bit more complex than simply saying Yes or No to all criteria for classification.

What a machine needs actually is data, that is some information to learn how spam e-mails are, or to know how dogs and cats can be differentiate, or to learn what criteria is to be used when approving or disapproving credit card for clients.

For the example of the e-mails above, Paul would need to provide a series of e-mails already classified, i.e., he has classified them in the past

as he would normally do and say to the machine: Ok now you have a set of e-mails which are spam and some which are not, learn how to classify them! Or in the case of the credit card approval, he would need to give historical data of clients and the results (approval or not) to the machine and say: This is the data and how it behaves, learn and tell me if I should approve or not a new client's credit card.

Paul thinks on all of these ideas and he thinks he can manage to help the Xing Bank to build their automatic system using Machine Learning. But first he wants to test the concept by classifying his e-mails. He arrives on the following final set of attributes:

Result: Is spam (1) or is not spam (0)

Criteria 0: English or Dutch (0) or any other language (1)

Criteria 1: Sent during the day (0) or at night (1)

Criteria 2: I know the person (0) or I don't know the person (1)

Criteria 3: Contains any of the words lottery, prize, shopping, discount (1) or it doesn't (0)

Using this set of criteria, Paul goes over his e-mail again and starts classifying them. He gets a piece of paper and writes: Email 1: It is in English (0), it was sent during the day (0), I don't know the person (1) but doesn't contain "wrong" words (0). Actually the e-mail is not a spam, but appears to be from a new client (0) so he actually writes:

Email1 -> 0, 0, 1, 0

Result1 -> 0

Paul notices that what he has written is very similar to what he would do using a programming language. As he knows some Python, he changes from the piece of paper to the computer, opens a notepad and writes again, using Python syntax.

Email1 = [0, 0, 1, 0]

Result1 = 0

Then he goes to the second e-mail. This time, the e-mail is apparently in Spanish (1), sent during the day (0) by an unknown person (1) and containing "wrong" words (1), so Paul writes:

Email2 = [1, 0, 1, 1]

Result2 = 1

Paul looks at the structure and he sees that the variable Result1 and Result2 can be eliminated by calling the emails as spam or not spam, so that's what he do with variables Email1 and Email2:

NotSpam1 = [0, 0, 1, 0]

Spam1 = [1, 0, 1, 1]

The next e-mail is in English, sent by a known person during the day and containing “wrong” words so,

NotSpam2 = [0, 0, 0, 1]

Getting excited with this job, Paul classifies the 6 last e-mails on his inbox and come to the following set of data.

NotSpam1 = [0, 0, 1, 0]

NotSpam2 = [0, 0, 0, 1]

NotSpam3 = [0, 1, 0, 0]

Spam1 = [1, 0, 1, 1]

Spam2 = [1, 1, 0, 1]

Spam3 = [0, 0, 1, 1]

To have all emails collected in a single variable, Paul creates a “data” array with the name of his data:

Data = [NotSpam1, NotSpam2, NotSpam3, Spam1, Spam2, Spam3]

At this point, he remembers that computers cannot read, i.e., it doesn't know that an e-mail is a spam just because the variable name is Spam1, so he creates a variable Marking which will store if the variable is a spam (1) or not (0),

Marking = [0, 0, 0, 1, 1, 1]

Paul wants to know how he can give now this information to the machine, so it can learn and start classifying his e-mail. But what if the machine starts classifying wrong, even having learned with this data? It would be safer to test the algorithm, which means to train it and then give it a new point which he knows to which class it belongs (spam or not) and see if the machine

correctly classified it. To do that, Paul gets another e-mail and store it as MisteryEmail.

```
MisteryEmail = [1, 0, 0, 0]
```

Finally, Paul arrives at the phase of Machine Learning. Now it is the time to start the actual process that he has dreamed so much, no more wasting time classifying emails in the morning! Googling, Paul finds very useful information to build classification systems.

One basic algorithm for classification is called K-Nearest Neighbors (KNN). Paul finds that, by using a functionality in python he does not need to implement the algorithm from scratch, rather he can directly test it in his problem. This is done by importing the library sklearn and the submodule neighbors using the following line of code.

```
from sklearn.neighbors import KNeighborsClassifier
```

On the internet documentation, Paul finds an example on how to use this model to perform some prediction, though he even does not understand very well how this model works. He notices that first he needs to create the model and say some attributes of it, sometimes referred to as “model parameters.” According the template, KNN takes at least one necessary parameter called the number of neighbors, which cannot be higher than the number of data used to train the model. As Paul has only 6 e-mail samples to use for that, he chooses half of it (3) and then writes:

```
neigh = KNeighborsClassifier(n_neighbors=3)
```

Now comes the step of training the model. Paul sees that there is a function to do that called fit() which takes as arguments the input data (Data) and the desired output (Marking), so the code is:

```
neigh.fit(Data, Marking)
```

Now that the model is trained, how well does it predict if an email is a spam or not? Paul sees that this model can be used to perform predictions using the method predict() which takes as argument the unknown input data. So, to predict if the mistery e-mail (MisteryEmail) is a spam or not, he writes the following:

```
print(neigh.predict([MisteryEmail]))
```

Out: [0]

The program returns the value 0. As he has already attributed, 0 means that the email was predicted not as spam. Paul gets excited with this result,

and wants to check how well the model predicts the data which was used for the training itself. After looking in the internet, he finds that a nice table can be printed using the following code.

```
for i in range(len(Data)):  
    print("Predicted: , "neigh.predict([Data[i]])[0], " Real: , "Marking[i])
```

Out:

```
Predicted: 0 Real: 0  
Predicted: 0 Real: 0  
Predicted: 0 Real: 0  
Predicted: 1 Real: 1  
Predicted: 0 Real: 1  
Predicted: 1 Real: 1
```

From the training data, Paul notices that only 1 value was wrongly predicted. The system said it is not a spam, while it is. Paul does not let himself get discouraged by this small errors, as he knows that a perfect model is very rare and almost impossible in some situation. Instead, he realizes that he has much to learn about Machine Learning before building a working system to solve real-life problems.

Paul also notices that the same system he built could be intuitively extended to other applications. For example, if instead of e-mails, one had letter? What if it was clients who pay or doesn't pay? Actually the same concept applied in the example above could be extended to a variety of classifications. In summary, the steps used in the case above were:

- For each element (or record), there were a series of features (characteristics), e.g., Data = [[1,0,0],[0,1,0]] where the set [1,0,0] is a record and the numbers inside represent features.
- Each feature (characteristic) has a meaning, possibly physical but also mental or virtual.
- If the feature(s) correspond to classes, it is represented by integers. They may also be continuous (floating-point numbers).
- The model is created and trained, e.g., using the fit method in sklearn (model.fit()).
- After training, the model is tested.
- No model is perfect, i.e., the result may be very accurate but it

is still prone to errors. Knowing the accuracy (how good/bad a algorithm performs) is an essential part of developing a model.

The next chapters will discuss the basic concepts of machine learning, applications, algorithms, and examples of application.

4

Elements of Machine Learning

CONTENTS

4.1. What Is Machine Learning?.....	62
4.2. Introduction To Supervised Learning	63
4.3. Introduction To Unsupervised Learning.....	71
4.4. A Challenging Problem: The Cocktail Party	73

4.1. WHAT IS MACHINE LEARNING?

Machine learning a specific area of data science and an application of Artificial Intelligence (AI) focused on developing systems that can learn and improve from experience without being explicitly programmed. These systems are in their essence computer programs that can access data and learn for themselves.

The term “Machine Learning” was reported at the first time in 1959 by Arthur Samuel. However, Tom M. Mitchell provided a well-posed definition of algorithms used in the machine learning field. His definition is the following: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E (Mitchell, (1997).” Extending such concept, Alan Turing proposed as a replacement to the question “Can machines think?,” the following inquiry: “Can machines do what we do, as thinking entities?.” By stating such inquiry and developing it. Turing shows the various implications and characteristics associated with the development of thinking machines (Harnad, 2008).

Example

A certain website classify images as appropriate (non-sexual, non-violent) or inappropriate, watching users actions and certain characteristics to improve its classification algorithm. In this scenario, what is the task T?

- Classify images as appropriate or inappropriate;
- Watching users actions;
- The number of image(s) correctly classified;
- None of the above

Solution

According Mitchell’s (Mitchell (1997)) definition of machine learning, the problem above can be interpreted according the following tasks or properties:

- Experience(E) – Watching users actions and certain characteristics;
- **Task (T) – classify images as appropriate or inappropriate;**
- performance measure (P) – the number of image(s) correctly classified;

Answer: (a)

End of solution

Machine learning tasks can be subdivided into: supervised learning, unsupervised learning and reinforcement learning or active learning. These tasks are described with more details in the following sections of this chapter.

The algorithms used in machine learning are intimately related to other field of science. To mention some:

- Data Mining: While machine learning focuses on prediction extracted from known features, data mining is concerned with discovering unknown features in the data. However, methods and techniques used in machine learning and data mining overlap significantly.
- Optimization: Learning problems are formulated as an optimization problem, focusing on minimizing errors between the model output and the desired targets on a dataset, called loss functions.

For instance, in a classification problem, the loss function express the difference between the predicted labels (the ones produced by the model) and the observed labels. An optimization problem is formulated to minimize such loss function.

4.2. INTRODUCTION TO SUPERVISED LEARNING

By definition, supervised algorithms learn through examples of the past and apply the acquired experience in new data to predict results. The analysis starts from a known training dataset which is used by the algorithm to produce inferences in order to predict output values. At the beginning of the training, the results of the algorithm are to a certain level random and highly inaccurate. As the system develops experience and learns the patterns contained in the data, it corrects itself, reducing errors.

One example is the prediction of housing price. Figure 4.1 represents (hypothetically) real state prices in the region of Buenos Aires, in USD. Some of the data is also shown in Table 4.1.

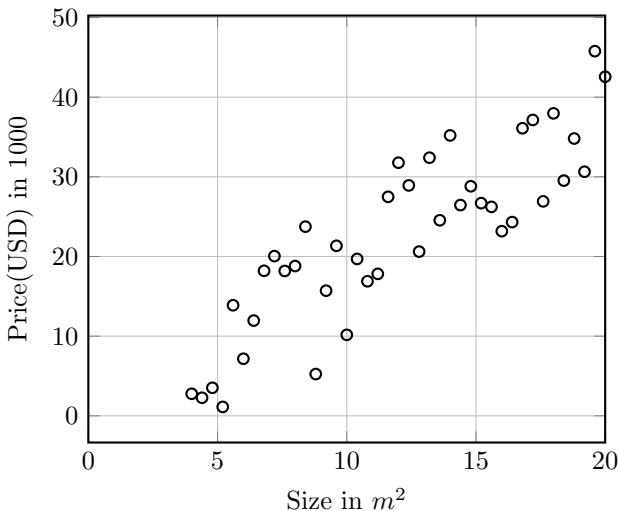


Figure 4.1: Housing price in Buenos Aires, in USD (hypothetical).

Table 4.1: Housing Price (USD) x Size (m^2) in Buenos Aires (Hypothetical Data)

Size (m^2)	Price (USD)
4.0	2.766
4.4	-4.2666
8.0	18.802
12.0	31.773
16.0	23.174
18.0	37.964
19.6	45.777
20.0	42.564

By visualizing the graph above, one could build a line representing a model able to perform predictions of housing price in Buenos Aires based on its size in square meter. Without performing any calculation, a possibility would be to directly connect the first and last points of the data. This is represented as a line over the data in Figure 4.2.

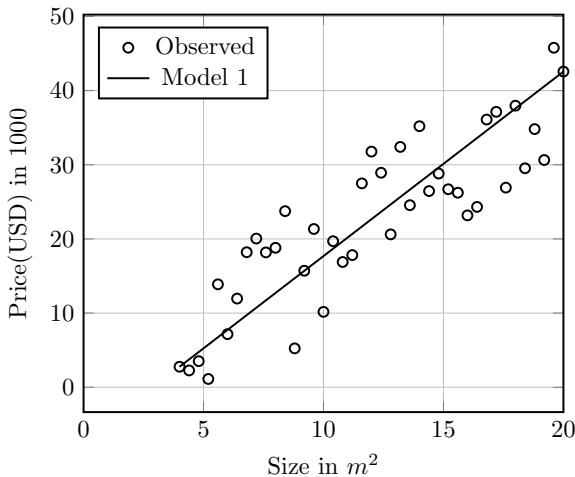


Figure 4.2: Housing price in Buenos Aires, in USD (hypothetical) with initial model.

A second choice, also not performing a single calculating directly, would be to connect the lowest and highest values in a line, thus constructing a second model as represented in Figure 4.3.

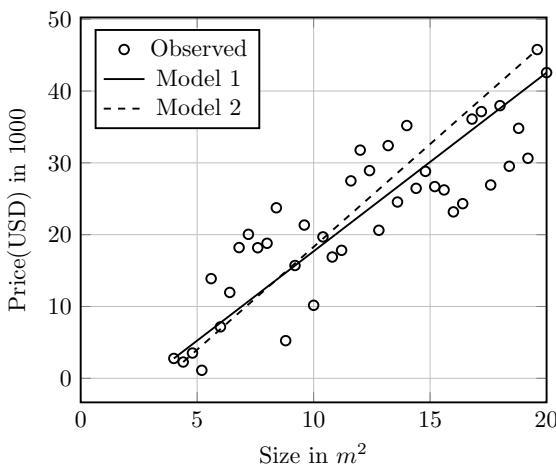


Figure 4.3: Housing price in Buenos Aires, in USD (hypothetical) with initial model (Model 1) and second model (Model 2).

Both of these models represent an algorithm (very simplistic) to try to predict the housing price. Such type of prediction is also referred to as

regression, since it involves the prediction of continuous values (price). If we want to express these models (models 1 and 2) in mathematical notation, we can use the generic line equation:

$$y = a_1x + a_0$$

where y is the output, i.e., price in this case, x is the independent variable (size in square meters) and a_1 and a_0 are parameters of the model. Adapting to the current problem, one could also write:

$$p = a_1s + a_0$$

where p is price and s is the size of the building. For model 1, the parameters can be calculated using the two connected points, which are represented in the Table 4.2.

Table 4.2: Data Used to Generate Model 1

Size (m^2)	Price (USD)
4.0	2.766
20.0	42.564

Substituting the size and price respectively in the line equation stated above gives two equations.

$$2.766 = 4a_1 + a_0$$

$$42.564 = 20a_1 + a_0$$

By subtracting the first from the second, one can obtain the value of the parameter a_1 .

$$42.564 - 2.766 = 20a_1 - 4a_1 + a_0 - a_0$$

$$39.798 = 16a_1$$

$$39.798 / 16 = a_1$$

$$2.487 = a_1$$

The second parameter, a_0 , can be obtained now by substituting the value found for a_1 at any of the equations used initially for this model.

$$2.766 = 4a_1 + a_0$$

$$2.766 = 4 \times 2.487 + a_0$$

$$-7.182 = a_0$$

Therefore, the equation (algorithm) that defines Model 1 can be expressed as:

$$p = 2.487s - 7.182$$

By simply replacing known values for housing size in the above equation, one could theoretically predict the price of it for the region of Buenos Aires. However, another problem arises: What about accuracy? Is this model accurate? How well can it predict the real prices? That is another point to be considered along this book.

For the second model, Table 4.3 shows the data used.

Table 4.3: Data Used to Generate Model 2

Size (m^2)	Price (USD)
4.4	2.267
19.6	45.777

A deduction similar to the one used for Model 1 can be performed to arrive in the following equation defining this model.

$$2.267 = 4.4a_1 + a_0$$

$$45.777 = 19.6a_1 + a_0$$

Subtracting the first equation from the second,

$$43.51 = 15.2a_1$$

$$a_1 = 2.862$$

$$a_0 = -10.318$$

The final equation for Model 2 thus is,

$$p = 2.862s - 10.318$$

The models developed above, though theoretically capable of being used to make prediction of house pricing, are not the best representations of machine learning algorithms, since they were developed by ignoring all the information passed by the complete dataset but the two points adopted for each case. In a machine learning system, the equation would be generated from the experiences acquired through the observation of all the points in the dataset, and the parameters would be adjusted not by direct calculation, but through an optimization algorithm, such as linear least squares (LLS) for linear systems or other algorithms for non-linear ones.

Another example of supervised learning is the problem of classifying data. For instance, consider Table 4.4 for Breast Cancer. It shows the classification of each cancer sample as malignant or benign as a function of the tumor size.

Table 4.4: Breast Cancer x Tumor Size

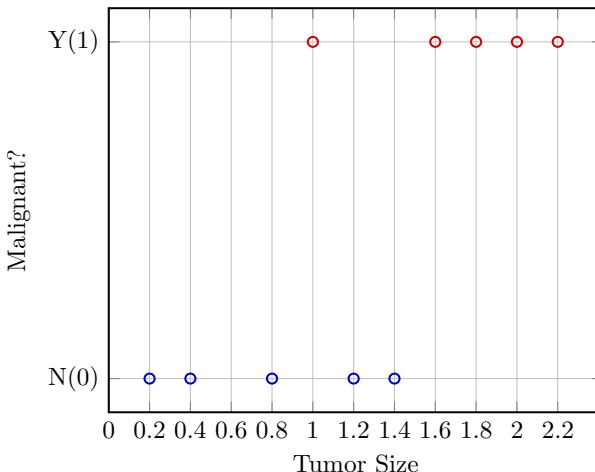
Malignant? (Y/N)	Tumor Size
N	0.2
N	0.4
N	0.8
Y	1.0
N	1.2
N	1.4
Y	1.6
Y	1.8
Y	1.2
Y	1.4

The above situation consists in a classification problem, where the classes are Malignant or Benign, each one being a label for Breast Cancer. Table 4.4 shows the classification as Y for yes (presence of Malign tumor) or N for no (absence of Malign tumor). In a mathematical sense, Y and N could be translated into 1 (presence) or 0 (absence) as it is with virtual signs. In this sense, Table 4.4 becomes Table 4.5.

Table 4.5: Breast Cancer x Tumor Size – In Terms of 0 and 1

Malignant? (Y/N)	Tumor Size
0	0.2
0	0.4
0	0.8
1	1.0
0	1.2
0	1.4
1	1.6
1	1.8
1	1.2
1	1.4

Such classification problem is usually represented in two types of graphs. The first one consists in plotting the data in a x-y place where y values are 0 and 1 for the class labels, as shown in Figure 4.4.

**Figure 4.4:** Breast Cancer x Tumor Size – representation in x–y plane.

A second possible visualization is the placement of all data points in a single row, with the labels being visible through coloring. This type of graph is shown in Figure 10.5.

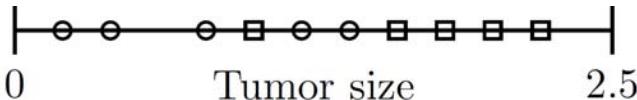


Figure 4.5: Breast Cancer x Tumor Size – representation in single row.

This a simple example of cancer classification (malignant or benign), but classification problems are not restricted to two classes (as in presence/absence or yes/no), rather it may involve as many classes as it may be necessary in a certain context. For instance, a much more complex cancer analysis would possibly have the labels:

- Absence of Tumor or Benign (0);
- Cancer of type I (1);
- Cancer of type II (2);
- Cancer of type III (3); etc.

In any case, the machine learning process consists in providing the system with the “medical” records relating the feature(s), e.g., tumor size and the target label(s), e.g., benign or malignant, so the machine can learn by identifying patters on the data and is possibly able to predict the classes for a new, unknown data.

Example

Suppose a company has contracted you, a data scientist, to help it to address the following problems:

- Problem 1: The company sells a variety of items, and it would like to predict its profit for the next 3 years based on past information.
- Problem 2: The company has e-mail accounts for each of its employees. It is requested to analyze each of the e-mail accounts and for each one decide if it has been hacked or compromised.

How should these problems be treated?

- Both involve supervised learning and are classification problems;
- Both involve supervised learning and are regression problems;
- Both involve supervised learning, but Problem 1 is a classification problem while Problem 2 is a regression problem;
- Both involve supervised learning, but Problem 1 is a regression problem while Problem 2 is a classification problem;
- They are not supervised learning problems.

Solution

Let's us consider each problem separately so we can correctly answer the question above.

- For Problem 1, the algorithm consists in using past number of sell data to predict profit. As profit is a continuous data and not a class, this problem consists in a regression.
- For Problem 2, the output of the algorithm is hacked/not hacked. Since these are classes, this is a classification problem.

Answer: (d)

End of solution

4.3. INTRODUCTION TO UNSUPERVISED LEARNING

When the information being learned by the system is not labeled nor classified, it uses unsupervised learning algorithm. Such algorithm evaluates the function so as to retrieve a hidden structure or pattern. Since there is no expected output *per se*, the system explores the data and make inferences, describing hidden structured.

A similar algorithm is semi-supervised learning. Both unlabeled and labeled data is used to train a system (usually a small set of labeled and a huge set of unlabeled data). Such algorithm is usually chosen when the data requires skilled and relevant resources so the system can learn from it.

The following sections describe some types of unsupervised learning algorithms and techniques.

4.3.1. Clustering

Suppose you are running a business and you sell stuff to people. Now, people buy stuff based on some criteria for instance: necessity, status, desire or any other thing. How can one know what drive's people purchase decisions?

In this sense, clustering algorithms works by finding and grouping data (people) according hidden patterns and inferences (people who buy because of necessity against people who buy for desire). These patterns and inferences are not known *a priori*, it is the machine learning algorithm who undercover it. And it does this through unsupervised learning. The groups are determined clusters, and the number of them are usually fixed, so the developer has some control over the granularity of the system. Some common types of clustering are:

- K-Means: cluster the data in a number K of mutually exclusive clusters. The choice of the number K may follow some heuristics, but there is no pragmatic rule about it.
- Hierarchical Clustering: Group the data into parent and child node(s). For instance, in a set of customers, they may be split first into male and female, then young and older, and other sub-clusters as required.
- Probabilistic clustering: clusters the data in a probabilistic scale.

After clustering, the algorithm will typically output the data points and the respective clusters to which each point belongs. At the end, it is up to the developer to understand what each cluster means in real-world.

4.3.2. Data Compression

In most real-world applications, lots of data is involved when developing machine-learning algorithms. How to know which data is relevant to a certain problem, when data is simply a set of numbers, letters and combinations of them. At this step, it comes to hand Data Compression. By reducing the data, it makes an algorithm faster and more efficient. It will only run the algorithm on relevant data and avoid training too much. The process of data compression is called dimensionality reduction.

Theory of information assumes that information can be redundant, thus not contributing significantly to a problem. Through such assumption, dimensionality reduction aims at reducing the dimensions (features) of a dataset to a fraction of the original content, leaving only the terms with highest contribution. There are a couple of algorithms currently used for dimensionality reduction:

- Principal Component Analysis (PCA): finds the linear combination of features that communicates most of the variance of the data.
- Single-Value Decomposition (SVD): factorizes the data into the product of other, smaller matrices.

Reducing the dimensionality of a dataset is an essential part of a machine learning problem dealing with big data, such as image processing. Each image consists in a set of matrices with values for each pixel representing an intensity of one of the “primary” colors (red, green and blue (RGB)), thus each pixel is made of three values stored each one in a matrix. Therefore, an image of 300 x 300 pixels will consist of three matrices, each one with 300 rows and 300 columns thus 9×10^4 values. The three matrices gives

$3 \times 9 \times 10^4 = 2.7 \times 10^5$ values. For a set with “only” 1000 images (considered a relatively small set), the total number of data available will be:

$$2.7 \times 10^5 \times 1000 = 2.7 \times 10^8$$

This shows how vital data compression is depending on the problem being solved.

4.4. A CHALLENGING PROBLEM: THE COCKTAIL PARTY

According Getzmann, Jasny, & Falkenstein (2016), and Jasny & Falkenstein (2016), the cocktail party effect refers to human brain’s power of showing selective attention, i.e., being able of focusing one’s auditory attention on a certain stimulus and to filter surrounding stimuli, as when a partygoer can focus on a single conversation in a noisy room. Though an apparently simple and regular task, scientists and engineers have been challenged to reproduce it in laboratory.

Naturally, an application of the cocktail party problem that scientists have invested their energy and time is in the field of music, where they are trying to develop algorithms which, as humans, can filter other sounds while capturing only the desired one, i.e., a certain music. Unfortunately, such task has been shown more challenging than expected.

Thanks to the work developed by Simpson, Roma, & Plumbley (2015) at the University of Surrey in the UK, great advances in the area of human voice recognition have been achieved. The scientists developed deep neural networks capable of separating such voices from the background in a wide range of songs. What it follows is a short description of the work they developed which serves as a great learning lesson in the field of unsupervised learning.

The starting point of the project consisted in splitting 63 songs in 20-second segments. Each song consists of individual tracks containing instrument or voice, and the fully mixed version of the song. From the 20-second track, a spectrogram was created showing how the frequencies of each sound vary with time. By doing so, the authors were able to capture a unique fingerprint characteristic of each instrument or voice. A spectrogram of the fully mixed version of the song was also created, which is basically a mixture of the spectrograms for every unique instrument voice present in the song.

By compiling such dataset, the task of separating voice or instrument becomes basically the identification and isolation of the characteristic spectrogram from the other superimposed spectrograms. To perform such job, a convolutional neural network was trained using 50 songs and tested in the remaining 13. Because each song was decomposed, the training set actually consisted into more than 20,000 spectrograms.

Hence, the process of preparing the convolutional neural network was: given a fully mixed spectrogram, it should reproduce the vocal spectrogram as output. This involves the optimization of a billion parameters, a quite intensive computational task.

The model performance (accuracy) showed to be so positive, that the Simpson, Roma, and Plumbley (2015) evaluated their results to those from a conventional cocktails party algorithm applied to the same data. They showed that deep neural network was able of capturing vocals with higher accuracy.

Exercises

- 1) Which of the following problems can be considered supervised learning? (Check all that apply)
 - a. From a collection of photos and information about what is on them, train a model to recognize other photos.
 - b. From a set of molecules and a table informing which are drugs, train a model to predict if a molecule is also a drug.
 - c. From a collection of photos showing four different persons, train a model which will separate the photos in 4 groups, each one for one person.
 - d. From a set of clients, group them according similarity to retrieve patterns.
- 2) According to the definition of Mitchell (1997), “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.” A predictive system is built by feeding a lot of blog comments, and it has to predict the behavior of users. In this context, T represents which component of the above description?
 - a. The process of learning from the data;
 - b. Blog comments;
 - c. None of those;
 - d. The probability of predicting the user behavior correctly.
- 3) Suppose you are working on the algorithm for Question 2, and you may predict user behavior as: violent, charismatic, happy or sad. Is this a classification or regression problem?
 - a. Regression;
 - b. Classification;
 - c. None of those;

- 4) You are predicting people's height based on its weight, age, gender and diet. Is this a classification or a regression problem?
- a. () Regression;
 - b. () Classification;
 - c. () None of those;
5. How would you personally define Machine Learning? _____

Solution

1) a.; b. 2) b. 3) b. 4) a.

5

Linear Regression With One Variable

CONTENTS

5.1. Introduction.....	78
5.2. Model Structure.....	78
5.3. Cost Function	80
5.4. Linear Regression Using Gradient Descent Method Using Python.....	96
Exercises.....	111

5.1. INTRODUCTION

Chapter 4 gave a general overview on the elements involved in machine learning algorithms. On this section, we explore specifically one supervised learning algorithm – linear regression- constrained to one single feature (variable) and one output. This model can be described as one of the simplest yet most didactic algorithms of machine learning.

As the name suggests linear regression with one variable is characterized by:

- Linear: A function or order 1, i.e., the highest exponential in a

variable is 1 as in: $y = 2x + 1$; $\frac{y}{20} = 0.4x$;

- Regression: As mentioned in Chapter 4, regression consists in one type of supervised learning, where the model is trained (learns) from past data, to predict certain values called target(s) or output(s);
- With one variable: A machine learning algorithm may involve as many variables as necessary, from one to thousands or billions, limited only the computational capacity. In this section, we deal with problems containing a single variables, i.e., predicting human weight based on size; house prices based on size; company profit based on number of sales, etc.

5.2. MODEL STRUCTURE

To understand the linear regression model, consider a hypothetical database with statistical units describing housing price in a location called Mountain City. Figure 5.1 shows a plotting of the square feet size of the buildings with the corresponding price in U.S dollars.

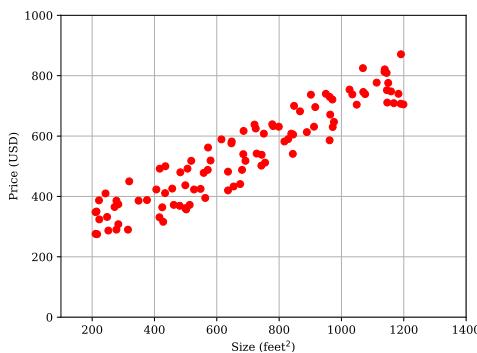


Figure 5.1: Housing price in Mountain City, in USD (hypothetical).

The main idea behind the linear regression is that the information on the y-axis (Price) is assumed to behave linearly with respect to the information on the x-axis (Size (ft^2)). As the data provided corresponds to the observed values, which we want to predict, this technique consists on **Supervised Learning**. Because the data is continuous (not classes), this is classified as a **Regression problem**.

Table 5.1 shows a small subset of the database presented in Figure 5.1. Notice how each record (or row) of the database corresponds to related information, i.e., a price is attributed to a housing size.

Table 5.1: Housing Dataset Sample

Housing Size (ft^2)	Price (USD)
648	582
548	425
1069	825
681	488
912	631
425	364
675	441
615	589
417	492
1147	711

Suppose the complete dataset contains 100 rows of data. The Housing Size variable is used as an input to predict the Prices variable. Using these concepts we can use the following notation:

- $m = 100$ (corresponds to the training examples or records);
- x (s) are the input variables, also called “features”;
- y (s) are the output variables, also called “targets.”

Each pair of data x, y is referred to as one training example. Therefore, using the data provided in Table 11.1, the pair (x_0, y_0) corresponds to the record (648,582), while the pair (x_9, y_9) corresponds to the record (1,147,711).

In any statistical predictive algorithm, the problem is structures according the following workflow:

- Define the training set (Table 5.1; Figure 5.1);
- Attribute a learning algorithm;
- Define a hypothesis h (e.g., the Housing Size is linearly correlated with the Price, thus linear regression is a good choice for predicting);
- With the hypothesis, perform predictions on the “target” variable.

In a linear regression, the hypothesis is the linear correlation, meaning that the target can be predicted using the following equation:

$$h(x) = a_0 + a_1 x$$

where x is the input variable (housing size), while a_i are the parameters of the model to be adjusted (calibrated). If there is only one input variable (as in the above case), then there are two parameters, a_0 and a_1 , and the problem is called a Univariate linear regression.

5.3. COST FUNCTION

One the model structure is defined, there should be a way of obtaining the optimal value of the parameters corresponding to the predictions on the target variable. If the value of such parameters are chosen at random, say

$a_0 = 1$ and $a_1 = 2$, what would be the predictions of the housing price? Figure 5.2 illustrates such case.

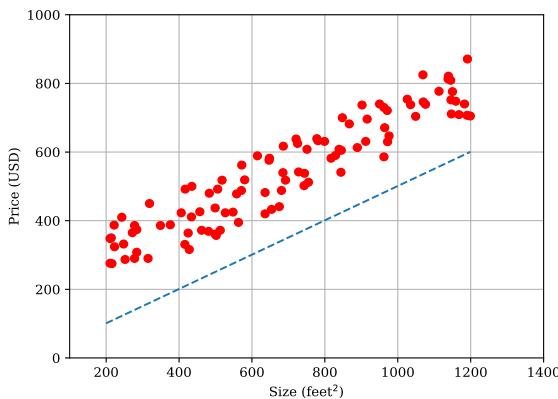


Figure 5.2: Prediction of Housing Price using “random” parameter values.

Obviously, the predicted values are not the best ones, since it can be seen that the line is shifted slightly below the data, with a reduced value of the bias a_0 . Increasing these parameters helps to achieve a slightly better solution, though still not the best one. After some tentative, we can arrive on the solution observed in Figure 5.3.

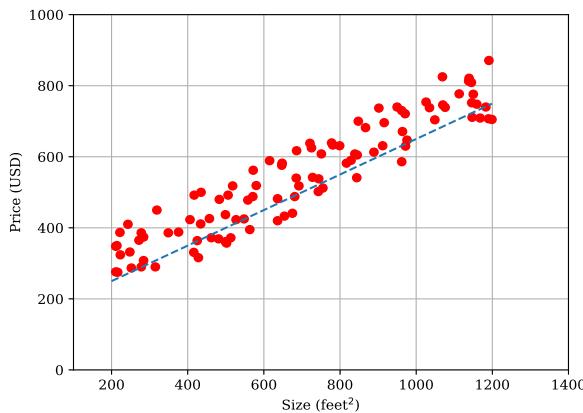


Figure 5.3: Prediction of Housing Price using “slightly better” parameter values.

Though much better, this output is still not the optimal. We could keep trying different values many times, but obviously this wouldn’t be the

best approach. There is a well-founded mathematical procedure to find such optimal values of the parameters. Consider that the idea is to find values of predictions $h(x_i)$ such that it is very close to the observed value y_i . Therefore, there should be a minimum difference between these two data points, which means:

$$(h(x_i) - y_i) = \text{minimum}$$

To guarantee that the difference is not dealing with negative numbers, which would mask the real minimum difference, it is better to square the difference, thus:

$$(h(x_i) - y_i)^2 = \text{minimum}$$

Of course, we don't want to find only the minimum difference for a point i , but for the complete set of records available on the training example. Thus, it would be better to define that we want to minimize the **sum of the squared differences**. In mathematical notation this is written as:

$$\min_{a_0, a_1} (h(x_i) - y_i)^2$$

Calling the condition above as J then one obtains:

$$\min_{a_0, a_1} J(a_0, a_1)$$

Suppose we want to use $J(a_0, a_1)$ to find the best parameter(s) of the linear regression model. Actually, to simplify the problem, we fix the parameter independent from x to be equal 0 ($a_0 = 0$), and we want to check the values of $J(a_1)$ and find an approximation of the minimum iteratively. By trying some values of a_1 and calculating the value of $J(a_1)$ we can obtain the result shown in Figure 5.4, highlighting the minimum cost (minimum value of J).

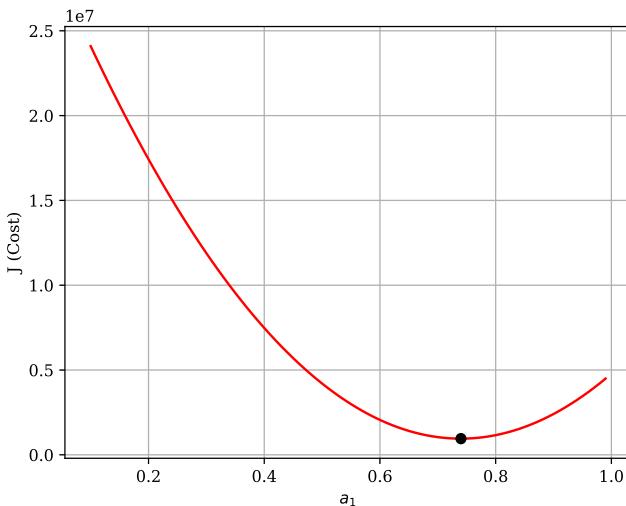


Figure 5.4: Values of J for Housing Price using $a_0 = 0$ as a fixed parameter.

The result above shows the Cost Function obtained after trying different values of a_1 while keeping a_0 fixed. The black dot indicates an approximation of the minimum. It cannot be said that this is exactly the minimum, since the values of J were obtained for discrete points. Still this value is very near the minimum when $a_0 = 0$. The value of a_1 is approximately 0.74. For this point the Cost Function value is $J(a_1) = 9.53 \times 10^5$.

As a second intuition, let's keep the parameter a_1 at the optimum found and determine now the best a_0 that minimizes the value of $J(a_0)$. After iterating over some values, we obtain Figure 5.5, which gives the cost as a function of the parameter a_0 .

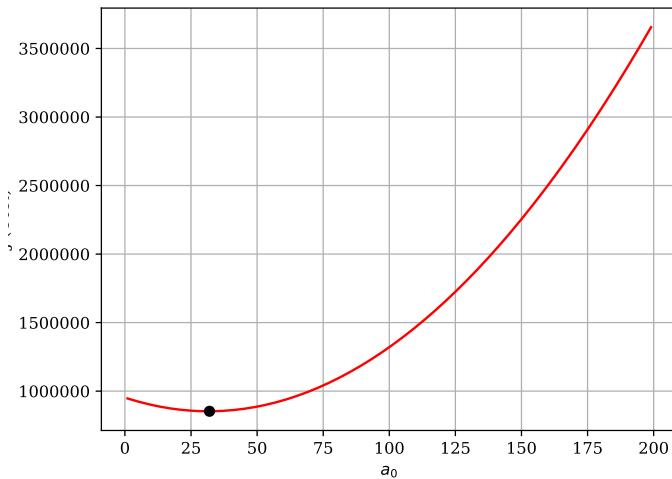


Figure 5.5: Values of J for Housing Price using $a_1 = 0.74$ as a fixed parameter.

According to Figure 5.5, there is a reduction of the Cost J as it is increased, until it reaches the approximate minimum at $a_0 = 32$. At this point, the Cost value is $J = 8.53 \times 10^5$.

For these two examples above, we have seen how the minimum cost J can be graphically found, by fixing one parameter and manipulating the other. However this procedure just served as a learning starting point and cannot be used to optimize a linear model. To do so, both parameters need to be adjusted at the **same time!** In an iterative, intuitive process this means to fix one parameter, vary the other and store the data. Then, change the previously fixed parameter to a new value and vary the second, collecting cost values. Doing this for a certain range of values will generate, not a curve as shown in Figures 5.4 and 5.5, but a surface where the valley indicates the presence of the minimum. For the problem stated above, the surface generated for the cost function is shown in the Figure 5.6, with an indication of the approximate minimum.

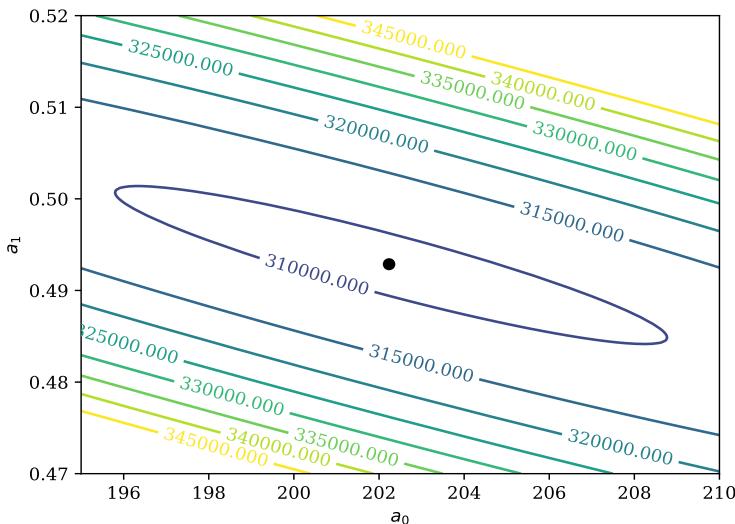


Figure 5.6: Surface showing values of J for Housing Price and the approximate minimum, pointing the values of a_0 and a_1 .

A look on the Figure above shows that the minimum cost lies exactly on the middle of the inner valley on the contour plot, exactly the valley. The values found at this position are:

$$a_0 = 202.23$$

$$a_1 = 0.49$$

$$J = 3.1 \times 10^5$$

Notice how the cost value J is much lower than the values found when trying to adjust the parameters one at a time! Still this value cannot be said to be the absolute minimum since the graphical analysis may be prone to small deviations. So how can we determine the **real** minimum value of the cost function J mathematically?

First we will work on the classical method to determine the optimum linear regression. Though straightforward, this method can only be directly used for linear regression. After we work on a iterative procedure that can be used for linear and non-linear models, thus being more general than the first one.

The classical method is the **Least Squares Method**, and it is approached on the following subsection.

5.3.1. Least Squares Method

The method starts from the presumption that a “perfect” model to represent the target data can be obtained by incorporating the line equation plus an error term. The error e incorporates all the variation in y that cannot be explained by the model. Thus,

$$y = a_0 + a_1x + e$$

The Least Squares Method consists in developing an algorithms to minimize the squared error (to keep positive) between the model and the

observed targets, that is $\min \sum_{i=2}^n e_i^2$. The idea is that, obtaining a minimum error means to obtain the minimum difference between the targets and the predictions. Rearranging the equation above to isolate the error one obtains:

$$e_i = y_i - a_0 - a_1x_i$$

Thus minimizing $\sum_{i=1}^n e_i^2$ mean minimizing $\min \sum_{i=1}^n (y_i - a_0 - a_1x_i)^2$. Call this function the Cost Function $J(a_0, a_1)$. Therefore,

$$J \approx \sum_{i=1}^n (y_i - a_0 - a_1x_i)^2$$

where n is the number of target examples. The minimum of the above equation can be obtained at the inflection point of the function. Mathematically it means the point where the derivatives (with respect to a_0 and a_1) are equal to zero. First apply the chain rule to determine the derivative with respect

to a_0 :

$$\frac{\partial J}{\partial a_0} = \frac{\partial J}{\partial x} \frac{\partial x}{\partial a_0}$$

$$\frac{\partial J}{\partial x} = 2 \sum_{i=1}^n (y_i - a_0 - a_1x_i)$$

$$\frac{\partial J}{\partial a_0} = -1$$

$$\frac{\partial J}{\partial a_1} = -2 \sum_{i=1}^n (y_i - a_0 - a_1 x_i) = 0$$

Divide both sides of the equation by $2n$ and rearrange to find the optimal value of a_0 :

$$-2 \frac{\sum_{i=1}^n y_i}{2n} + 2 \frac{\sum_{i=1}^n a_0}{2n} + 2 \frac{\sum_{i=1}^n a_1 x_i}{2n} = \frac{0}{2n}$$

Remember that the average value \bar{y} is defined as $\frac{\sum_{i=1}^n y_i}{n}$, then the equation can be severely simplified.

$$-\bar{y} + a_0 + a_1 \bar{x} = 0$$

$$a_0 = \bar{y} - a_1 \bar{x}$$

The equation above defines a way to determine a_0 , but to do so it is necessary to know the value of a_1 , which can be defined by calculating the value of the derivative of the cost function with respect to a_1 , as follows.

$$\frac{\partial J}{\partial a_1} = -2 \sum_{i=1}^n x_i (y_i - a_0 - a_1 x_i) = 0$$

Substituting the equation defining a_0 in the equation above one obtains:

$$-2 \sum_{i=1}^n x_i (y_i - \bar{y} + a_1 \bar{x} - a_1 x_i) = 0$$

Rearranging,

$$\sum_{i=1}^n x_i (y_i - \bar{y}) + a_1 \sum_{i=1}^n x_i (\bar{x} - x_i) = 0$$

$$a_1 \sum_{i=1}^n x_i (\bar{x} - x_i) = -\sum_{i=1}^n x_i (y_i - \bar{y})$$

$$a_1 = \frac{-\sum_{i=1}^n x_i (y_i - \bar{y})}{\sum_{i=1}^n x_i (\bar{x} - x_i)}$$

$$a_1 = \frac{\sum_{i=1}^n x_i (y_i - \bar{y})}{\sum_{i=1}^n x_i (x_i - \bar{x})}$$

Using the above equation, one can directly determine the optimum value of the parameter a_1 and in sequence the value of a_0 can be determined.

The fact that Least Squared Method represents a direct, non-iterative method of determining optimum parameters represent a huge advantage of iterative methods, which relies in performing a calculation over and over until a value has converged. However, the arithmetic solution was derived directly for a linear model (line equation). The same type of solution cannot be directly used for non-linear models, requiring necessarily an iterative process to determine the optimal parameters.

In the following section, it is demonstrated how Python can be used to solve the linear regression problem using as example the Housing Price data mentioned above.

5.3.2. Linear Regression and Least Squares Method Using Python

In this section, we implement a script to perform linear regression over the housing price dataset, generating a model able to perform predictions of price based on the housing size. Consider that the complete dataset is saved on a file “housing_data.csv.” The data stored in this file is contained in Appendix A.

First, initialize the empty lists (vectors) of feature x and target y .

```
x = [] # Square size
y = [] # Housing Price
```

Now, open the filename containing the data (consider that the data is on the same folder of the script). We ignore the first row, since it contains the file headers (name of the columns and not data). The procedure of collecting the data consists on looping over the entire file, row by row, transforming each data to appropriate format and appending to the lists.

```
with open('housing_data.csv', 'r') as file:  
    # read header  
    file.readline()  
  
    # loop over the file, filling the lists with x and y vectors  
    for line in file:  
        row = line.split(',')  
        x.append(float(row[0]))  
        y.append(float(row[1]))  
  
The line,  
with open('housing_data.csv', 'r') as file:  
    opens the file and attributes it to the object file. The following line,  
# read header  
    file.readline()
```

reads the first line of data and do not store it. This is necessary since the first row contains the headers and not actual data.

The following lines iterate over each row of the file, transforming data to floating-point and appending to the lists *x* and *y*.

The Least Squares Method required the calculation of the average value for both *x* and *y*. To do that, define a function called *average(x)*, which receives a list *x* as input, sum up all the values and divides by the number of elements in the list, characterizing the average.

```
# read header  
def average(x):  
    return sum(x) / len(x)
```

Using this function obtain the average of each variable of the problem:

```
avg_x = average(x)  
avg_y = average(y)
```

With that, we are ready to obtain the value of the parameter a_1 followed by a_0 . The value of these variables can be obtained by generating a list through list comprehension (or using an “explicit” for loop), and summing up all the values.

```
a1_num = sum([(xi - avg_x)*(yi - avg_y) for (xi, yi) in zip(x, y)])
a1_den = sum([(xi - avg_x)**2 for xi in x])
a1 = a1_num/a1_den
a0 = avg_y - a1*avg_x
```

On the above code, *a1_num* is used to define the numerator, while *a1_den* is used to define the denominator component of the expression to calculate a_1 . After obtaining this variable, the calculation of *a0* is straightforward using already determined variables. For illustration, print to the screen the values of a_0 and a_1 obtained.

```
print('Parameter a0 = ',a0)
print('Parameter a1 = ',a1)
```

Up to this point we solved the linear regression and determined the parameters to obtain the optimal linear model to predict housing price. For instance, to predict the price of housing with 300 ft²,

```
ypred = a0 + a1*300
print(ypred)
```

Though the linear regression is ready, how good is the model? We can evaluate accuracy by determining the **Coefficient of Determination (R^2)**. It is calculated according the following equation.

$$R^2 = 1 - \frac{SQ_{res}}{SQ_{total}}$$

where SQ_{\pm} is the squared sum of residues, which is by definition exactly the same as the cost function J , defined as:

$$SQ_{res} = \sum_{i=1}^n (y_i - a_0 - a_1 x_i)$$

The SQ_{total} is the total squared sum, defined as the accumulated squared difference between each point and the average value.

$$SQ_{total} = \sum_{i=1}^n (y_i - \bar{y})$$

First, let's obtain SQ_{res} and SQ_{total} in Python by generating a list using list comprehension and applying the sum() method.

```
sqres = sum([(yi - (a1*xi + a0))**2 for (xi, yi) in zip(x, y)])
```

```
sqtot = sum([(yi - avg_y)**2 for yi in y])
```

Generate R^2 by performing the division of $sqres$ by $sqtot$ as done in the following code:

```
R2 = 1 - sqres/sqtot
```

```
print('Coefficient of Determination R2 = ',R2)
```

Finally, print the final value of the cost function J by referring to the variable $sqres$, which stored exactly the same value.

```
J = sqres
```

```
print('Final Cost J = {:.3e}'.format(J))
```

The complete code of this example is stored in Appendix A. The final values obtained for the parameters a_0 , a_1 , for the coefficient of determination R^2 and for the Cost Function J are:

- $a_0 = 202.283$;
- $a_1 = 0.493$;
- $R^2 = 0.875$;
- $J = 3.09 \times 10^5$.

Notice that there is a small difference from the cost function J value from this mathematical procedure to the graphical one. Still it is important to observe that the one above represents the **exact global minimum** Cost value, while on the graphical procedure this value is never **exactly** determined, but always obtained an approximation.

5.3.3. Gradient Descent (GD) Method

Differently from the previous example, Gradient Descent Method consists in an iterative method where, at each step, the algorithm “searches” to which direction the minimum cost (or optimal point) is, and moves towards that direction. The iteration is performed until the algorithm converges, i.e., the steps become smaller and smaller.

As every iterative method, GD starts from an initial solution, non-optimal, also referred to as initial guess. For the linear regression, call this initial solution a_0^i and a_1^i . These parameter values are associated with a certain cost function $J(a_0^i, a_1^i)$ value. The objective is to have an algorithm where at each iteration, the cost function J is modified through modification of a_0^i and a_1^i , until it reaches a minimum, indicated by the convergence.

The update equation of GD method is as follows.

$$\theta_i^{n+1} = \theta_i^n - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$$

where θ is a variable to be optimized (e.g., a linear regression parameter), i is a variable index (as in a_0 and a_1 for the linear regression problem), n is an iteration index and α called learning rate. This last variable determines the “size of the step” when iterating, i.e., a multiplier of the gradient given in the derivative. If it is less than 1, then the step size will always be smaller than the gradient. If it is higher than 1, the step will always surpass the gradient value. It is important to note that all the variables of a problem must be updated simultaneously and not serially. For instance, the following pseudocode is a valid GD method application.

```
# var0 is the updated value of theta_0
var0 = theta_0 - alpha * der(J, theta_0, theta_1)
# var1 is the updated value of theta_1
var1 = theta_1 - alpha * der(J, theta_0, theta_1)
# now update the old values for the next iteration
theta_0 = var0
theta_1 = var1
```

Notice the difference from the previous code to the next one, where the first variable is directly updated after the calculation of its new value. This is **not** a correct GD method application!

```
# var0 is the updated value of theta_0
var0 = theta_0 - alpha * der(J, theta_0, theta_1)
# now update the old values BEFORE the next iteration
```

```

theta_0 = var0
# var1 is the updated value of theta_1
var1 = theta_1 - alpha * der(J, theta_0,theta_1) # notice that here, theta_0 =
var0. WRONG!
# now update the old values for the next iteration
theta_1 = var1

```

Let's develop some intuition on the GD algorithm, for a better understanding on how it can find the minimum of a function by "following" the gradient. Remember that the cost function, defining the error of the model, consists in the squared difference of the observed y_{obs} and the predicted value $y_{pred} = a_1x + a_0$

$$J = \sum_{i=0}^n (y_{obs} - (a_1x + a_0))^2$$

Which characterizes a quadratic (parabola) equation, with a clear minimum value at the bottom of the parabola. Consider the example of the finding the linear equation that best fits the following set of data.

Table 5.2: Data Example for GD Intuition

x	y
-5.0	-13.0
0.1	-2.8
5.0	7.0

Which is determined by the linear equation $y = 2x - 3$. Now suppose one does not know the parameters of the equation. In this sense, the value of cost function J can be calculated for different values of the parameters a_0 and a_1 . For simplicity, let's fix the value of $a_0 = -3$, which is the true, value and only try to find the value of the parameter a_1 . Using additional data points, the following parabola is obtained which clearly shows that $a_1 = 2$ is the bottom of the parabola.

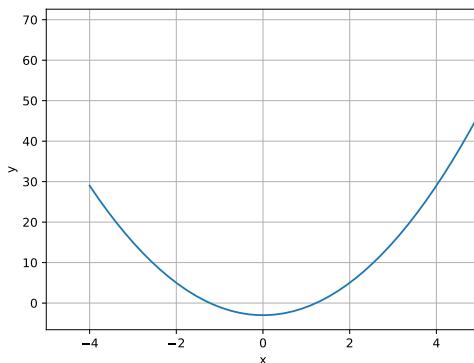


Figure 5.7: Values of J for linear regression with fixed $a_0 = -3$ and varying a_1 for the data in Table 5.2.

As already mentioned, to determine the optimum point, i.e., $a_1 = 2$ using Gradient Descent method, one has to choose an appropriate value for the learning rate α . This choice may lead to three scenarios.

- Scenario 1: The learning rate chosen is too small. This is observed by the necessity of many iterations until the value of the function converges to the minimum.

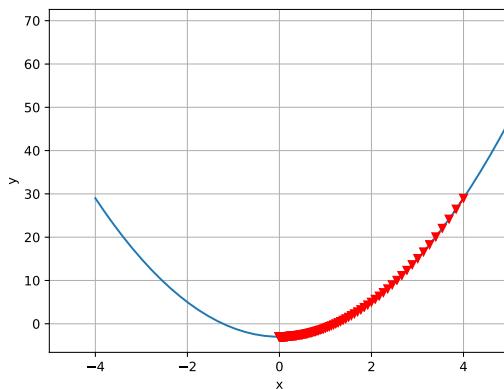


Figure 5.8: Value of learning rate α , is too small, requiring too much iterations.

- Scenario 2: The learning rate is just enough, with fast convergence to the minimum without oscillation or taking too much iterations.

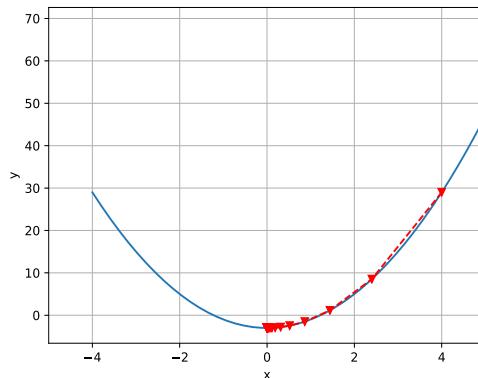


Figure 5.9: Value of learning rate α , is reasonable, requiring few iterations and no oscillations.

- Scenario 3: The learning rate is too high, leading to oscillations. In the extreme case, the algorithm may even diverge, not being able to determine the minimum of the function successfully. Even when it converges, α can be stated as too high when it produces an overshoot of the minimum (it “crosses” the minimum).

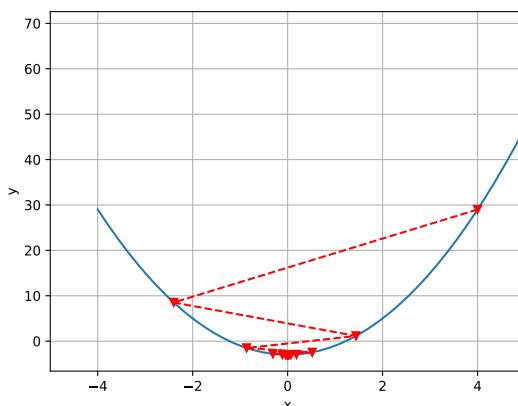


Figure 5.10: Value of learning rate α , is too high, producing relative divergence in the search for the minimum.

In highly non-linear problems, the gradient descent algorithm may also locate local optima of the cost function, not necessarily arriving in the global minimum. Searching using different initial guesses may help to find a “better” optimum of the function.

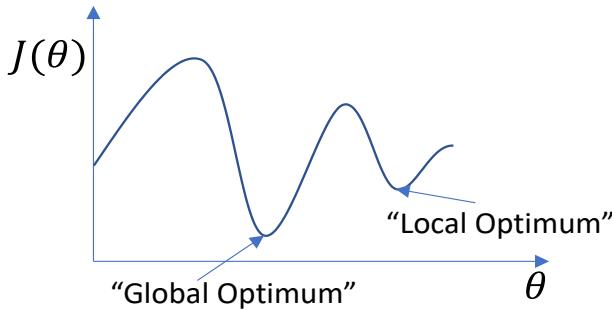


Figure 5.11: Example of local minima and global minima in non-linear cost function.

To apply the GD method in linear regression, it is necessary to determine the derivatives of the cost function J with respect to each parameter a_0 and a_1 for the parameter updating equation. These derivatives are shown in the equations below.

$$\frac{\partial}{\partial a_0} J(a_0, a_1) = \sum_{i=0}^n (y_{obs}^i - a_1 x^i - a_0)$$

$$\frac{\partial}{\partial a_1} J(a_0, a_1) = x^i \sum_{i=0}^n (y_{obs}^i - a_1 x^i - a_0)$$

5.4. LINEAR REGRESSION USING GRADIENT DESCENT METHOD USING PYTHON

In this section, GD update is used to determine the best parameters of a linear regression model. Python programming language is used to implement the algorithm in a generalized form. In this way, the same algorithm may be used to solve a different set of problems. Additionally, the example is enriched with visuals for best experience of the reader.

Case background: The company “Money buys all,” interested in increasing its sales, collected in a table its total sales together with the total money spent on advertising for the last 10 years. The data is shown in Table 5.3.

Table 5.3: Company Sales and Advertising Investment from 2009 to 2018

Year	Sales (1000 USD)	Advertising (1000 USD)
2009	330.17	20.00
2010	498.82	24.44
2011	478.52	28.89
2012	451.32	33.33
2013	565.38	37.78
2014	601.42	42.22
2015	845.30	46.67
2016	619.36	51.11
2017	969.84	55.56
2018	995.62	60.00

The company hired a data scientist to implement a machine learning model for prediction of the future expected sales, considering that the advertising investment is planned to increase by 10% at each year.

Being by *importing the necessary libraries*. In this problem we use *matplotlib.pyplot* for visualization (plots), and *mplot3d* for 3D plots.

```
import matplotlib.pyplot as plt # visualization
from mpl_toolkits.mplot3d import Axes3D # 3D visualization
```

Define the *sales* and the *advertising* as vectors in Python. Looking at the data, would it make sense to use linear regression model to predict the sales based on the advertising investment?

```
sales = [330.17, 498.82, 478.52, 451.32, 565.38, 601.42, 845.30, 619.36,
969.84, 995.62]
```

```
advertising = [20.00, 24.44, 28.89, 33.33, 37.78, 42.22, 46.67, 51.11, 55.56,
60.00]
```

```
plt.plot(advertising,sales, 'ob')
plt.xlabel('Advertising')
plt.ylabel('Sales')
plt.grid()
```

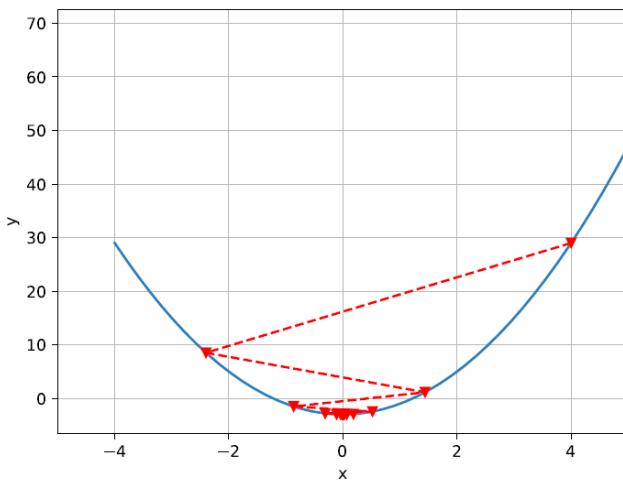


Figure 5.12: Plot of dataset.

Figure 5.12 indicates that a linear regression model may be a good model for prediction of sales. However, the initial value of the two parameters a_0 and a_1 are not initially known. So define it as arbitrary values, for instance:
 $+ a_0 = 0 + a_1 = 2$

Therefore, the initial model can be written as:

$$y = a_0 + a_1 x = 2x$$

Let's visualize the predicted values using this model (y) and the observed values of the sales. It is not expected that the results will be good, since this is an initial (very bad) guess on the model parameters.

```
y = [0 + 2*adv for adv in advertising]
plt.plot(advertising, sales, 'ob', label = 'Observed')
plt.plot(advertising, y, '--r', label = 'Predicted')
plt.xlabel('Advertising')
plt.ylabel('Sales')
plt.grid()
```

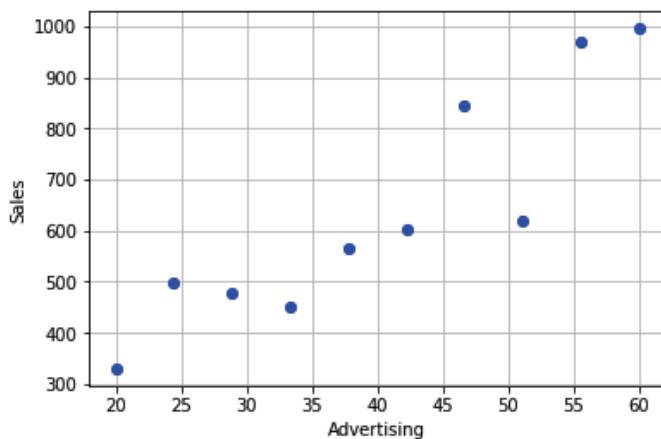


Figure 5.13: Dataset with prediction.

Obviously, this model is not a good predictor for the sales generated per advertising. The objective of the next steps is to adjust the value of the two parameters a_0 and a_1 , in order to obtain a reasonable model that can be used to predict the sales with high accuracy.

5.4.1. Cost Function Analysis

As already mentioned, the optimization process in GD method consists in minimizing a function (cost function), which measures the errors of the model compared to the observed data. Call this cost function $\pm(a_0, a_1)$. One way it can be defined is as the sum of the squared errors (SSE) or sum of squared residuals. The error e is defined as:

$$e = y_{obs} - y_{pred}$$

where y_{obs} is the observed output (sales) and y_{pred} is the predicted output. From the equation of the linear regression, the error can thus be rewritten as:

$$e = y_{obs} - (a_0 + a_1 x) = y_{obs} - a_0 - a_1 x$$

Let us define a function to calculate the error according the equation above.

def error(yobs, x, a0, a1):

“

```
e = error(yobs, ypred)
```

Inputs:

yobs (float) – observed value

x (float) – input

a0 (float) – parameter of linear model

a1 (float) – parameter of linear model

“”

```
return yobs - a0 - a1*x
```

It is important, at each step, to define test cases so we are assured that the code is properly set up. The following code tests the error for a simple example with known output.

```
# Test case
```

```
a0 = 1; a1 = 1
```

```
x = 1
```

```
ypred = a0 + a1*x
```

```
yobs = 2
```

```
error_true = yobs - ypred
```

```
error_calc = error(yobs, x, a0, a1)
```

```
if abs(error_true - error_calc) < 1e-5:
```

```
    print("Passed!")
```

```
else:
```

```
    print("Wrong error!")
```

```
Passed!
```

The cost function $J(a_0, a_1)$ is the sum of squared errors (SSE), which can be defined as the sum of all errors, squared.

$$J(a_0, a_1) = SSE(a_0, a_1) = \sum_{i=0}^{n-1} e(a_0, a_1)^2$$

Let us define a function to obtain this value

```
def J(yobs, xobs, a0, a1):
```

“”

```
e = error(yobs, ypred)
```

Inputs:

```
yobs (list) – list of observed values  
xobs (list) – list of observed inputs  
a0 (float) – parameter of linear model  
a1 (float) – parameter of linear model  
“”  
J = 0  
for y, x in zip(yobs, xobs):  
    J += error(y, x, a0, a1)**2  
  
return J
```

Define a simple test case to evaluate if the SSE is correctly obtained.

```
# Test case  
a0 = 1; a1 = 1;  
x = [1, 2, 3]  
yobs = [1, 2, 3]  
ypred = [2, 3, 4]  
errors = [-1, -1, -1]  
errors2 = [1, 1, 1] # squared errors from the list above  
J_real = 3 # the result of summing up all values from the list above  
J_calc = J(yobs,x, a0, a1)  
if abs(J_calc – J_real) < 1e-5:  
    print("Passed!")  
else:  
    print("Wrong SSE!")  
Passed!
```

For visualization of the error, let us define a function which can plot the cost function $J(a_0, a_1)$ for a range of values in a_0 and a_1 , and insert a point where is the current value of J according the selected a_0 and a_1 . To generate that visualization we will need *numpy* library. The template of this function is

```
def plotJ(yobs, xobs, a0, a1, [a0min, a0max], [a1min, a1max])
```

```
import numpy as np  
def plotJ(yobs, xobs, a0, a1, a0range, a1range):  
    """
```

plotJ(yobs, xobs, a0, a1, a0range, a1range)

Plot a 3D surface of the cost function and show some points being evaluated.

Inputs:

yobs (list) – list of observed values

xobs (list) – list of observed inputs

a0 (list) – parameter of linear model

a1 (list) – parameter of linear model

a0range (list (2,)) – limits of a0 to be plotted

a1range (list (2,)) – limits of a1 to be plotted

""

Create the matrices A0 and A1 to be used in the 3D plot as values of the parameters

```
A0, A1 = np.meshgrid(range(*a0range), range(*a1range))
```

initialize the cost function surface with the matrix A0

```
Jmat = np.zeros_like(A0)
```

```
row = 0
```

```
col = 0
```

loop and fill the values of the matrix J

```
for a0list, a1list in zip(A0, A1):
```

```
    col = 0
```

```
    for a0p, a1p in zip(a0list, a1list):
```

```
        Jmat[row, col] = J(yobs, xobs, a0p, a1p)
```

```
        col += 1
```

```
    row += 1
```

calculate the error at the current values of parameters

```
Jpoint = []
for a0val, a1val in zip(a0, a1):
    Jpoint.append(J(yobs, xobs, a0val, a1val))

# generate the plotting of the surface and the point being evaluated
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot(111, projection = '3d')
ax.plot_surface(A0, A1, Jmat, alpha = 0.5, rstride = 1, cstride = 1)
ax.plot(a0, a1, Jpoint, 'o-r')
ax.set_xlabel('a0')
ax.set_ylabel('a1')
ax.set_zlabel('J(a0, a1)')
```

As in the previous examples, define a test case to evaluate the correctness of the above function. An example code is shown below.

```
xobs = [1, 2, 3]
yobs = [2, 3, 4]
a0 = [0, 1]; a1 = [0, 1]
plotJ(yobs, xobs, a0, a1, [-3, 3], [-4, 4])
```

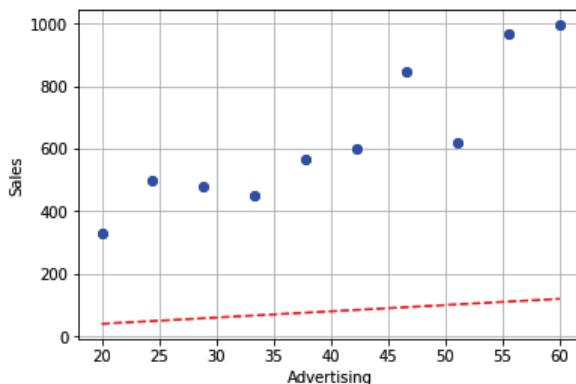


Figure 5.14: Test of plotJ function.

Finally, as part of the Cost Function Analysis section, it is relevant for the GD algorithm to define the derivatives of the cost function $J(a_0, a_1)$

with respect to the parameters a_0 and a_1 . They are defined according the following equations:

$$\frac{\partial J}{\partial a_0} = -2 \sum_{i=0}^{n-1} (y_{obs} - a_0 - a_1 x) = -2 \sum_{i=0}^{n-1} e(a_0, a_1)$$

$$\frac{\partial J}{\partial a_1} = -2 \sum_{i=0}^{n-1} x(y_{obs} - a_0 - a_1 x) = -2 \sum_{i=0}^{n-1} x e(a_0, a_1)$$

The following function is used to determine the two derivatives mentioned above, at a single point.

def derJ(yobs, xobs, a0, a1):

“”

derJ(yobs, xobs, a0, a1)

Calculates the derivatives of the cost function J with respect to a0 and a1.

Inputs:

yobs (float) – observed value

x (float) – input

a0 (float) – parameter of linear model

a1 (float) – parameter of linear model

Returns:

dJda0 (float) – derivative of J with respect to a0

dJda1 (float) – derivative of J with respect to a1

“”

 dJda0 = 0

 dJda1 = 0

for y, x **in** zip(yobs, xobs):

 dJda0 += -2*error(y, x, a0, a1)

 dJda1 += -2*x*error(y, x, a0, a1)

return dJda0,dJda1

As it was done with all previous function, evaluate this one at a certain point with known value to test the correctness of the code implemented above.

```

xobs = [1, 2, 3]
yobs = [2, 3, 4]
a0 = 1; a1 = 1
dJda0, dJda1 = derJ(yobs, xobs, a0, a1)
if abs(dJda0 + dJda1) < 1e-5:
    print("Passed")
else:
    print("Wrong implementation")
Passed

```

5.4.2. Algorithm Implementation

Now that some basic functions are already defined, the environment is prepared to implement the algorithm of the GD optimization itself. The update of the parameters is performed by adjusting it according the gradient of the function at the current position weighted by a value α known as the *learning rate*. The update equations are defined as follows:

$$a_0(n+1) = a_0(n) - \alpha \frac{\partial J(a_0(n), a_1(n))}{\partial a_0}$$

$$a_1(n+1) = a_1(n) - \alpha \frac{\partial J(a_0(n), a_1(n))}{\partial a_1}$$

We define now the update() function, which will receive the actual value of the derivative, the learning rate and the parameter and apply the above equation.

```

def update(aold, alpha, dJ):
    anew = aold - alpha * dJ
    return anew

```

The final step of the algorithm implementation consists on creating the loop to update the parameters of the linear equation until they converge to a value, assumed to be the optimum set. This is done through the function shown below.

```

def solveGD(yobs, xobs, initguess, alpha, MAXITER = 100, verbose = False):
    a0 = initguess[0] + 1000

```

```
a1 = initguess[1] + 1000
```

```
a0new = [initguess[0]]  
a1new = [initguess[1]]
```

```
iter_ = 0
```

```
if verbose:
```

```
    print(f'{iter_:5s}\t{a0:5s}\t{a1:5s}\t{J:5s}\t')
```

```
Jval = [J(yobs, xobs, a0new[0], a1new[0])]
```

```
while abs(a0 - a0new[-1]) > 1e-5 and abs(a1 - a1new[-1]) > 1e-5 and  
iter_ < MAXITER:
```

```
    a0, a1 = a0new[iter_], a1new[iter_]
```

```
if verbose:
```

```
    print(f'{iter_:5d}\t{a0:5.2f}\t{a1:5.2f}\t{Jval[-1]:5.2e}\t')
```

```
dJda0, dJda1 = derJ(yobs, xobs, a0, a1)
```

```
a0new.append(update(a0, alpha, dJda0))  
a1new.append(update(a1, alpha, dJda1))  
Jval.append(J(yobs, xobs, a0new[-1], a1new[-1]))
```

```
iter_ += 1
```

```
return {'a0': a0new[-1], 'a1': a1new[-1], 'history': {'a0': a0new, 'a1':  
a1new, 'J': Jval}}
```

Once the algorithm is implemented, let us test it by solving the initial problem of this section. The same initial guess is used and appropriate value for the learning rate α is selected.

```
a0 = 0
```

```
a1 = 2
```

```
optim = solveGD(sales, advertising, [a0, a1], alpha = 0.00001, MAXITER = 100, verbose = True)
```

```
iter a0 a1 J
```

0	0.00	2.00	3.46e+06
1	0.11	6.88	1.50e+06
2	0.18	10.03	6.74e+05
3	0.23	12.08	3.30e+05
4	0.26	13.40	1.86e+05
5	0.28	14.26	1.26e+05
6	0.29	14.81	1.00e+05
7	0.30	15.17	8.97e+04
8	0.31	15.40	8.53e+04
9	0.31	15.55	8.34e+04
10	0.31	15.65	8.27e+04
11	0.32	15.71	8.23e+04
12	0.32	15.75	8.22e+04
13	0.32	15.78	8.22e+04
14	0.32	15.79	8.21e+04
15	0.32	15.81	8.21e+04
16	0.32	15.81	8.21e+04
17	0.32	15.82	8.21e+04
18	0.32	15.82	8.21e+04
19	0.32	15.82	8.21e+04
20	0.32	15.82	8.21e+04
21	0.32	15.82	8.21e+04
22	0.32	15.83	8.21e+04
23	0.32	15.83	8.21e+04
24	0.32	15.83	8.21e+04
25	0.32	15.83	8.21e+04

```
26 0.33 15.83 8.21e+04  
27 0.33 15.83 8.21e+04  
28 0.33 15.83 8.21e+04  
29 0.33 15.83 8.21e+04
```

It can be seen from the results above that the parameters successfully converged to the final values:

$$a_0 = 0.33$$

$$a_1 = 15.83$$

with a final value of the cost function $J \approx 8.2 \times 10^4$. This means a reduction of two orders of magnitude from the initial guess, which started with 3.5×10^6 . Figure 5.15 illustrates the final result obtained, with the optimum linear regression according GD algorithm.

```
a0 = optim['a0']  
a1 = optim['a1']  
sales_pred = [a0 + a1*x for x in advertising]  
plt.plot(advertising, sales, 'ob', label = 'Observed')  
plt.plot(advertising, sales_pred, '--r', label = 'Predicted')  
plt.xlabel('Advertising')  
plt.ylabel('Sales')  
plt.grid()
```

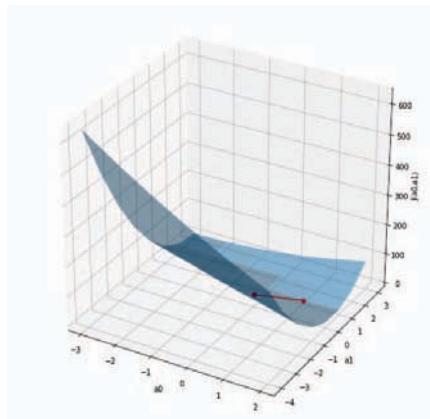


Figure 5.15: Optimum linear regression model using GD.

In the next Figure, observe how the linear regression model evolves through the iterations, reaching the final optimum slope and displacement from the origin. It is important to notice that changing the learning rate will change this convergence. If it is increased above a certain threshold, it will not converge to the optimum, but will diverge, which is not desired. Therefore, it is essential to maintain the learning rate inside a window so the model converges.

```
plt.plot(advertising,sales,'ob',label = 'Observed')
for a0, a1 in zip(optim['history'][['a0']],optim['history'][['a1']]):
    sales_pred = [a0 + a1*x for x in advertising]
    plt.plot(advertising,sales_pred,'--')

plt.xlabel('Advertising')
plt.ylabel('Sales')
plt.grid()
```

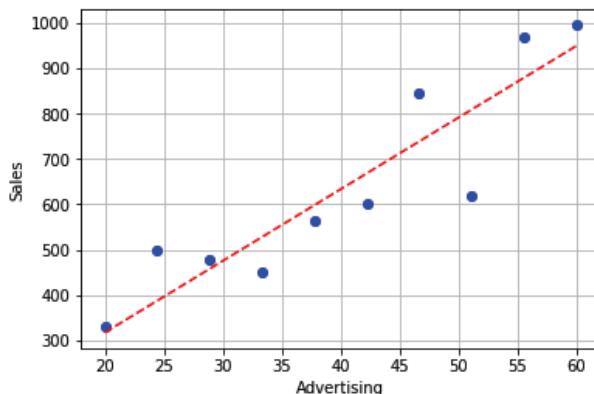


Figure 5.16: Evolution of optimization process in 2D plot.

Finally let's conclude the analysis by reusing the 3D plot function and visualize how GD method “walks” along the cost function, finally reaching the optimum values of a_0 and a_1 .

```
a0min = min(optim['history'][['a0']])
a0max = max(optim['history'][['a0']])
a1min = min(optim['history'][['a1']])
```

```
a1max = max(optim['history'][‘a1’])
plotJ(sales,advertising,optim[‘history’][‘a0’],optim[‘history’][‘a1’],[-1,2],[-1,20])
```

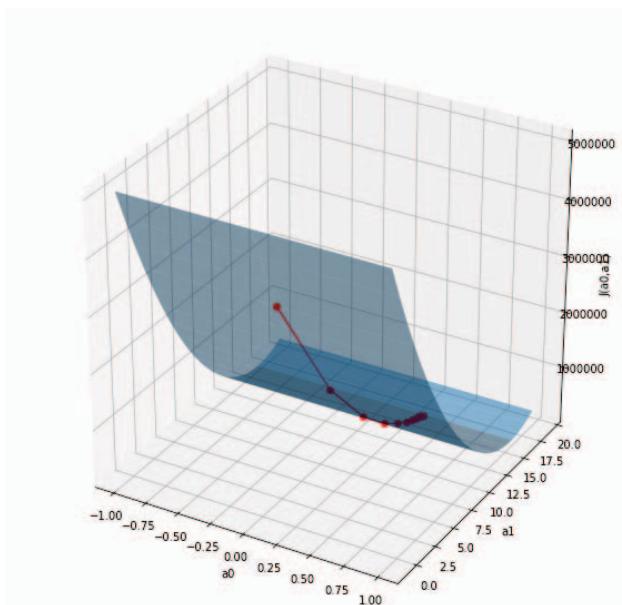
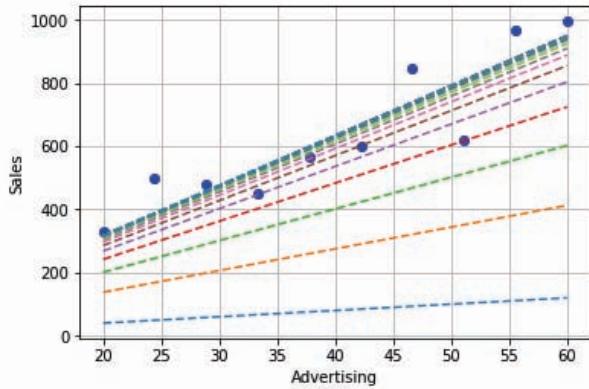


Figure 5.17: Evolution of optimization process in cost function.

EXERCISES

- 1) A consulting office received a project of predicting the score of a hotel booking website based on the number of clicks. The number of clicks C consists in the how many times an user clicked on a website feature. The score R related to how well should this hotel booking website be ranked, since more numbers of click would mean more users accessing, which may book a hotel or not. In this sense, it would be interesting to predict the score R of a certain website, once it is known the number of clicks on it, which can be easily collected from the server information. The score R, on the other hand, is obtained through a rather complex formula and needs theoretically lots of information to be well established.

Consider the following training set, containing the number of clicks for different websites and their scores. The score changes annually, so different records in the table below may refer to the same website. Using a linear regression as hypothesis ($y = a_0 + a_1x$), and considering N as the number of training samples,

C (Million)	R
2	3.0
4	3.1
1	2.2
4	2.6

For the training set above, what is the number of training samples? (Hint: should be a number between 0 and 10).

- 2) In the training of linear regression model, one may often refer to the “Cost Function.” What it is (conceptually)? How it is defined mathematically? Give an example.
- 3) Consider the following linear regression equation (model) with one variable.

$$y = 2.4 + 3.7x$$

Can one say that this model is a good predictor for the following dataset? Why is it good/bad? Give your reasons in measurable terms? (You may relate to the magnitude of the errors and the observed values, or the cost function)

C (Million)	R
2	3.0
4	3.1
1	2.2
4	2.6

- 4) A linear regression model is defined according the values of its parameters $a_0 = 0.0$ and $a_1 = 2.2$. What is the predicted value y for $x = 2$?
- 5) While developing a model to predict flood levels in a city using precipitation (rain) levels, an engineer found values for a_0 and a_1 such that $J(a_0, a_1) = 0$. How is that possible? Choose the correct answer.
- (a) This is actually not possible. He made a mistake.
 - (b) All the data fitted perfectly a straight line, so the model shows no error.
 - (c) This always happen if the model has good results (predictions), since there will be no errors.
 - (d) This would only happen if all observed values $y^i = 0$ for all values of $i = 0, 1, 2, \dots, N$
- 6) What is the purpose of the Least Squares Method and the Gradient Descent method in linear regression? What are the differences between them?

6

A General Review on Linear Algebra

CONTENTS

6.1. Introduction.....	114
6.2. Matrices And Vectors	114
6.3. Addition	116
6.4. Multiplication.....	117
6.5. Matrix-Vector Multiplication.....	118
6.6. Matrix-Matrix Multiplication.....	121
6.7. Inverse And Transpose.....	123
Exercises.....	127

6.1. INTRODUCTION

This chapter will provide a broad overview on the main concepts of linear algebra. This will help the reader to bring back some of the basic concepts on matrix and vector operation, as well as to help in the next chapters to develop a better understanding on the concepts introduced. For those who feel confident when working with Linear Algebra, they may safely skip this chapter. However, it is advisable to go through it since there are many basic concepts that may be easily forsaken for one that is not dealing continuously with linear algebra calculations.

In the first part the concepts of matrices and vectors are reviews. Then a short outlook on the main operations one can perform with matrices and vectors is mentioned, always focusing on examples. The last part reviews the more complex matrix operations, namely inverse and transpose. Though this section does not bring programming explicitly, it forms the basic concepts one should have when working with scientific computing.

6.2. MATRICES AND VECTORS

A matrix can be seen as a rectangular (or squared) array of numbers, properly placed in rows and columns. Rows are formed by stacking number horizontally, while columns are created through vertical stacking. The following is an example of a matrix we will call M.

$$M = \begin{bmatrix} 211 & 132 & 895 \\ 456 & 23 & 482 \\ 1 & 90 & 314 \end{bmatrix}$$

M is referred to as a 3x3 matrix with domain $R^{3 \times 3}$. Consider now a second matrix named N .

$$N = \begin{bmatrix} 211 & 132 \\ 456 & 23 \\ 1 & 90 \end{bmatrix}$$

N is a 3x2 matrix. In this sense, the dimension of a matrix is always: **number of rows x number of columns**. A third example to clarify even more is the matrix K.

$$K = [211 \ 132 \ 32]$$

In this case, the matrix above contains a single row with 3 elements, i.e., 3 columns. So its dimension is 1×3 , and the domain is $R^{1 \times 3}$.

Each number in a matrix is called a **matrix element** or **entry**. Their exact position can be easily addressed by its row and column. For example, consider again the matrix M referred above. The **entry** 132 is placed in the first row, second column, therefore it is the element $M_{1,2}$. Similarly, the entry 90 is placed in the third row, second column. Therefore, it can be addressed as element $M_{3,2}$. Since this matrix has only 3 rows and 3 columns, a reference to an entry $M_{4,1}$, for instance, is not correct, or undefined.

A vector consists in a type of matrix composed by a single row or single column. If it contains one row, then it is called a row-vector, while if it has one column it is a column-vector. Therefore, the dimension of a vector is always $(n \times 1)$ or $(1 \times n)$, depending if it is a row or column vector.

One example of vector that was already mentioned is K. It consists in a row vector. Another example, now a column vector, is A.

$$A = \begin{bmatrix} 211 \\ 132 \\ 32 \end{bmatrix}$$

Note that A has the same values of K, but now arranged as a column vector. In such case, it is said that A is the **transposed** vector K. However, we will revise such concept later in this chapter.

Since one of the dimensions of a vector is always one, the dimension of it may be addressed by a single reference, being it a row or a column vector.

For instance, in the vector A, the element $A_1 = 211$. Still, it would not be wrong to refer to the same value as $A_{1,1}$, though it is not recommended.

Up to here we have addressed the first element of a matrix or vector as starting with 1. However there are systems that consider the initial counting from 0 (zero). For example, Python programming language creates matrices 0-indexed, i.e., starting the counting from 0 instead of 1. On the other side,

MATLAB(R) generates matrices 1-indexed, i.e., starting the counting from 1. Therefore, it is important to be aware of the system being used when dealing with matrices. specially programmatically.

6.3. ADDITION

Addition of matrices can be easily performed, in a very similar way to addition of single numbers. For each entry i,j in each matrix, sum the corresponding numbers and generate the resulting matrix. For example, consider the following two matrices.

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 10 & 30 & 50 \\ 100 & 300 & 500 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 4 & 6 \\ 20 & 40 & 60 \\ 200 & 400 & 600 \end{bmatrix}$$

The sum of A and B, generating matrix C, is performed by using scalar sum at each entry. Therefore,

$$C_{1,1} = A_{1,1} + B_{1,1} = 1 + 2 = 3$$

$$C_{1,2} = A_{1,2} + B_{1,2} = 3 + 4 = 7$$

$$C_{1,3} = A_{1,3} + B_{1,3} = 5 + 6 = 11$$

Following the same reasoning for each and every row and column of A and B, one will obtain the following matrix C as a result.

$$C = \begin{bmatrix} 3 & 7 & 11 \\ 30 & 70 & 110 \\ 300 & 700 & 1100 \end{bmatrix}$$

If you have followed the reasoning and tries to do this with different examples, soon a critical property becomes evident:

Matrix addition can only be performed in between matrices with the same dimension

Therefore, trying to add, for instance a matrix 2x2 with one 3x2 will simply be mathematically impossible. The number of rows and the number of columns must be all the same so addition can be done.

6.4. MULTIPLICATION

6.4.1. Scalar Multiplication and Division

Scalar multiplication refers to the product between a single number (real) and a matrix. It is done by multiplying each entry of the matrix with the single number. For example, consider the following scalar multiplication and the resulting matrix.

$$4 \diamondsuit \text{times} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 10 & 20 & 40 \end{bmatrix} = \begin{bmatrix} 4 & 12 & 20 \\ 8 & 16 & 24 \\ 40 & 80 & 160 \end{bmatrix}$$

Notice that, in such case, the dimension of the original matrix is kept the same (above is 3x3), with the entries being now multiples of the single number.

Extensively, a division of matrix by singular value can be seen as the multiplication of the same matrix by the inverse number. For instance, the following division,

$$\begin{bmatrix} 2 & 6 & 8 \\ 1 & 4 & 6 \\ 10 & 20 & 40 \end{bmatrix} / 2$$

Can be performed by inverting the number 2, becoming $1/2$ and doing the multiplication as already explained.

$$\frac{1}{2} \times \begin{bmatrix} 2 & 6 & 8 \\ 1 & 4 & 6 \\ 10 & 20 & 40 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 \\ 1/2 & 2 & 3 \\ 5 & 10 & 20 \end{bmatrix}$$

6.5. MATRIX-VECTOR MULTIPLICATION

Not all matrices can be multiplied by all vectors. There are some rules that should be observed before performing such operation. The basic rule is:

The inner dimension numbers should be the same.

By inner dimension numbers it is meant that the number of columns of the first multiplicand should be the same as the number of rows of the second multiplicand. Therefore, if the vector is the first multiplicand, it must be a row-vector. For instance,

$$[2 \ 6 \ 8] \times \begin{bmatrix} 2 & 6 \\ 1 & 4 \\ 10 & 20 \end{bmatrix}$$

In the above case, the vector (first element) has dimension 1x3 (1 row and 3 columns), while the matrix (second element) has dimension 3x2 (3 rows and 2 columns). When performing the multiplication, the following dimensions are multiplied: 1x3 x 3x2. The dimensions 3 (from vector) and 3 (from matrix) are the inner dimensions while the dimensions 1 (from vector) and 2 (from matrix) are the outer dimensions. Since the inner dimensions are the same (3), the multiplication can be performed.

The resultant matrix will have the same dimension as the outer ones of the multiplicands involved. In the above case, the resultant matrix will have dimensions 1x2.

$$[2 \ 6 \ 8] \times \begin{bmatrix} 2 & 6 \\ 1 & 4 \\ 10 & 20 \end{bmatrix} = [90 \ 196]$$

The resultant matrix can be obtained by multiplying the row elements of the first multiplicand by the column elements of the second one and summing them up. This is shown in the development below, if the resultant matrix is called R.

$$R_{1,1} = 2 \times 2 + 6 \times 1 + 8 \times 10 = 4 + 6 + 80 = 90$$

$$R_{1,2} = 2 \times 6 + 6 \times 4 + 8 \times 20 = 12 + 24 + 160 = 196$$

Consider a second example, where matrix A is declared as follows.

$$A = \begin{bmatrix} 1 & 2 & 3 & 2 \\ 4 & -3 & 2 & 2 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

The matrix above has dimensions 3x4 (3 rows and 4 columns). It is multiplied by a vector (the vice-versa is not valid), this one must have exactly 4 rows, so **the inner dimensions meet!** As it is a vector, the other dimension is one (1), so it can be defined as a column-vector. This will be called x and written as below.

$$x = \begin{bmatrix} -1 \\ 2 \\ 3 \\ 2 \end{bmatrix}$$

Using the algorithm described above, the resultant matrix y (with dimension 3x1) is calculated as follows.

$$y_{1,1} = 1 \times -1 + 2 \times 2 + 3 \times 3 + 2 \times 2$$

$$y_{2,1} = 4 \times -1 + -3 \times 2 + 2 \times 3 + 2 \times 2$$

$$y_{3,1} = 0 \times -1 + 0 \times 2 + 1 \times 3 + -2 \times 2$$

The result is shown below.

$$y = \begin{bmatrix} 16 \\ 0 \\ -1 \end{bmatrix}$$

6.5.1. Application Example

The following (Table 6.1) dataset shows the poverty level and the teen birth rate for 4 US states.

Table 6.1: Dataset of Poverty Level and Teen Birth Rate for Four US States

State	Poverty Level	Teen Birth Rate (15–17)
Alabama	20.1	31.5

Massachusetts	11	12.5
Mississippi	23.5	37.6
New York	16.5	15.7

Adapted from: *Mind On Statistics, 3rd edition* apud PennState (n.d.).

After analyzing this data and developing a linear regression model, one arrived in the following equation.

$$(\text{Predicted Birth Rate}) = 4.3 + 1.4 (\text{Poverty Level})$$

Using this equation and the data provided, shown that the valued for Predicted Birth Rate can be obtained using Linear Algebra (i.e., operations with matrix) instead of performing explicitly the calculation for each record.

Solution

The first step consists in rewriting the scalar equation in matrix one. To do so, incorporate a bias of 1 (one) being multiplied by 4.3, so the equation becomes:

$$(\text{Predicted Birth Rate}) = 4.3 (1) + 1.4 (\text{Poverty Level})$$

In matrix form, the predicted birth rates can be determined through the solution of the following matrix-vector multiplication.

$$\begin{bmatrix} 20.1 & 1 \\ 11 & 1 \\ 23.5 & 1 \\ 16.5 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1.4 \\ 4.3 \end{bmatrix} = \begin{bmatrix} y_{pred}^1 \\ y_{pred}^2 \\ y_{pred}^3 \end{bmatrix}$$

As already shown, the values of the predicted birth rates y_{pred}^1 , y_{pred}^2 and y_{pred}^3 can be determined by using the rule of multiplication, in which case generates the following set of equations.

$$y_{pred}^1 = 20.1 \times 1.4 + 1 \times 4.3$$

$$y_{pred}^2 = 23.5 \times 1.4 + 1 \times 4.3$$

$$y_{pred}^3 = 16.5 \times 1.4 + 1 \times 4.3$$

6.6. MATRIX-MATRIX MULTIPLICATION

Multiplication using matrices is far less simple than the operations shown previously. While on addition and subtraction, the matrices would necessarily have the same size or be a single number, in multiplication the following should rules should hold:

- Between the two matrices, the “inner” dimensions should be the same. For example, the multiplication of a matrix 2x3 with one 3x4 is possible, since the number of columns of the first is 3 and it matches the number of rows of the second. In this case, 2 and 4 are the outer dimensions and 3 is the common inner dimensions. On another hand, multiplication of a matrix 2x3 by another 2x3 is not possible since the inner dimensions (3 and 2) do not match.
- The matrix generated from matrix-matrix multiplication will have the same dimension as the outer dimensions of the matrices being multiplied. For instance, the multiplication of a matrix 2x3 with a 3x4 generates a matrix 2x4.

The multiplication process follows the same algorithm as shown for the matrix-vector operation, but with additional columns on the second operand. Consider the following example.

$$\begin{bmatrix} 1 & 3 & 2 \\ 5 & 6 & 7 \end{bmatrix} \cdot \begin{bmatrix} 4 & 5 \\ 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The multiplication of the above matrices is done through multiplying each element of the rows of the first matrix by each columns of the second matrix and summing it up, generating a single element. Therefore, the element $y_{1,1}$, representing the element of the row 1 and column 1 of the resulting matrix can be determined as follows.

$$y_{1,1} = 1 \times 4 + 3 \times 1 + 2 \times 3 = 13$$

Remember that, since the first matrix has dimensions 2x3 and the second one has dimensions 3x2, the resulting matrix will have dimensions 2x2 (2 rows and 2 columns). The element $y_{1,2}$, representing the element on the first row, second column can be calculated as,

$$y_{1,2} = 1 \times 5 + 3 \times 2 + 2 \times 4 = 19$$

Similarly, all elements of the matrix can be calculated. This will generate the following resulting matrix.

$$\begin{bmatrix} 13 & 19 \\ 47 & 65 \end{bmatrix}$$

Another way of structuring the solution of this multiplication is to construct it as multiple operations between matrix and vectors. In this case, the second matrix is broken into column-vectors. Thus,

$$\begin{bmatrix} 4 & 5 \\ 1 & 2 \\ 3 & 4 \end{bmatrix} = HSTACK \left(\begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix}; \begin{bmatrix} 5 \\ 2 \\ 4 \end{bmatrix} \right)$$

where *HSTACK* means horizontal concatenation operator. By building these vectors, one can determine the resulting vector of matrix-vector multiplication at each one, such as shown below.

$$\begin{bmatrix} 1 & 3 & 2 \\ 5 & 6 & 7 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 13 \\ 47 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & 2 \\ 5 & 6 & 7 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 19 \\ 65 \end{bmatrix}$$

Now perform horizontal concatenation of the resulting vector, reconstructing the matrix. The same result is obtained using both approaches (direct matrix-matrix multiplication and deconstruction of second matrix).

6.6.1. Properties of Matrix Multiplication

The following are important properties regarding matrix multiplication. It also shows clear differentiation of this operation from the regular, scalar multiplication.

- Non-commutative

Consider two matrices, A and B. In general, $A \times B \neq B \times A$

Example:

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3 & 4 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 13 \\ 6 & 19 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 4 \\ 1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 10 & 15 \\ 7 & 16 \end{bmatrix}$$

The dimensions of the matrices may also differ. If A is $m \times n$ and B is also $m \times n$, then $A \times B = C$ where C is $m \times m$, while $B \times A = C$, where C has dimensions $n \times n$.

- Associative

Let A, B and C be different matrices, where $\pm \times \times$ is a valid operation (according inner dimensions). If $\pm \times =$, then $A \times B \times C = D \times C$. Additionally, if $B \times C = E$, then $A \times B \times C = A \times E$. In summary, associating two matrices gives the same answer as performing the multiplication of the matrices directly.

- Identity multiplication

Consider the identity matrix I , which is composed by the number 1 in the main diagonal and 0 in all other elements. For instance, $I_{3,3}$.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The multiplication of any matrix A by the identity I is the same as the multiplication of the identity I by the matrix A (commutative). This statement can be expressed mathematically as follows.

$$A \cdot I = I \cdot A = A$$

6.7. INVERSE AND TRANPOSE

6.7.1. Inverse

Consider a single, real number r , the inverse of r is

$$\frac{1}{r} = r^{-1}$$

Multiplying the number by its inverse yields always 1.

$$r \times \frac{1}{r} = 1$$

However, not all numbers have an inverse. Consider the number 0. The inverse of it would be $1/0$, which is obviously undefined.

With matrix, similar reasoning is applicable. Consider a generic matrix A . Then multiplying the matrix by its inverse A^{-1} yields the identity matrix I . For instance, consider a 2×2 matrix A . Then the following operation can be performed, IF A has an inverse.

$$A \cdot A^{-1} = I = [1 \ 0 // 0 \ 1]$$

As it occurs with single numbers, not all matrices have an inverse. Those which cannot be inverted are called *singular* or *degenerate*.

Application Example

Let C be an invertible 3×3 matrix.

$$C = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 4 \\ 1 & 2 & 1 \end{bmatrix}$$

Obtain the matrix C^{-1} , i.e., the inverse of C .

Solution

From the definition of inverse, one obtains the following expression.

$$C \cdot C^{-1} = I$$

Expanding the matrices above and calling c_{ij} as the elements of the matrix C^{-1} at row i and column j ,

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 4 \\ 1 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The elements of C^{-1} now can be determined using the algorithm for matrix multiplication. However it is not straightforward. One can notice that by writing the equations to determine the elements of the inverse matrix. Three of these equations are expressed below.

$$1 \times c_{1,1} + 0 \times c_{2,1} + 1 \times c_{3,1} = 1$$

$$1 \times c_{1,2} + 0 \times c_{2,2} + 1 \times c_{3,2} = 0$$

$$1 \times c_{1,3} + 0 \times c_{2,3} + 1 \times c_{3,3} = 0$$

To obtain the value of all the 9 elements of the matrix C^{-1} , one needs to solve a set of 9 coupled equations. There is a variety of methods to simplify the obtaining of such elements without “manually” solving the set of equations, which would require several substitutions and recurrences. The final result to be obtained is as follows.

$$C^{-1} = \begin{bmatrix} 0.875 & -0.25 & 0.125 \\ -0.5 & 0. & 0.5 \\ 0.125 & 0.25 & -0.125 \end{bmatrix}$$

6.7.2. Transpose

To transpose a matrix means to change its “direction,” transforming rows into columns and columns into rows. For example, let A be a generic matrix with the form:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}$$

Matrix A transposed, A^T , consists in shifting the indexes, so element a_{ij} is placed at index ji , in such way that the matrix shown below is obtained.

$$A^T = \begin{bmatrix} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \dots & \dots & \dots & \dots \\ a_{1,m} & a_{2,m} & \dots & a_{m,n} \end{bmatrix}$$

Application Example

Obtain B^T from matrix B .

$$B = \begin{bmatrix} 4 & 5 & 6 \\ 1 & 0 & 1 \\ 9 & 8 & 7 \end{bmatrix}$$

Solution

Call $B^T = C$. In this case, each element $c_{i,j}$ of matrix C can be obtained as:

$$\begin{aligned} c_{1,1} &= b_{1,1} & c_{1,2} &= b_{2,1} & c_{1,3} &= b_{3,1} & c_{2,1} &= b_{1,2} & c_{2,2} &= b_{2,2} & c_{2,3} &= b_{3,2} & c_{3,1} &= b_{1,3} \\ c_{3,2} &= b_{2,3} & c_{3,3} &= b_{3,3} \end{aligned}$$

In summary, the matrix C is,

$$C = \begin{bmatrix} 4 & 1 & 9 \\ 5 & 0 & 8 \\ 6 & 1 & 7 \end{bmatrix}$$

EXERCISES

- 1) Consider the following two matrices.

$$A = \begin{bmatrix} 1 & -5 \\ -3 & 4 \end{bmatrix}$$

$$C = \begin{bmatrix} -2 & 0 \\ 4 & 1 \end{bmatrix}$$

What is $A - C$

$$\begin{bmatrix} 1 & -5 \\ -7 & 3 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -5 \\ 1 & 5 \end{bmatrix}$$

$$\begin{bmatrix} -2 & 0 \\ -12 & 4 \end{bmatrix}$$

- a) None of above.

Answer: a

- 2) Let $d = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$. What is $d \cdot \frac{1}{3}$?

$$\begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 2/3 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -2 \\ -1 \\ 0 \end{bmatrix}$$

- a) None of above.

Answer: b

- 3) Consider a 3-dimensional column vector u , defined as,

$$u = \begin{bmatrix} 7 \\ 6 \\ 5 \end{bmatrix}$$

What is u^T ?

$$\begin{bmatrix} 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 6 & 7 \end{bmatrix}$$

Answer: a

- 4) A certain industrial process consists in manipulating an electric signal to generate a certain output. According the engineering department, the parameter vector v used in the equation describing such process is,

$$v = [6.4 \quad 3.2]$$

The equation representing the dynamics of this process, in matrix form is:

$$o = v^T \cdot i$$

where o is the output vector, v^T is the vector v transposed and i is the input vector. Let i be the following set of input signals.

$$i = \begin{bmatrix} 1 & 4 & 2 & 8 \\ 0 & 5 & 2 & 10 \end{bmatrix}$$

What is θ ?

$$[4.2 \quad 10.3 \quad 21.9 \quad 12.5]$$

$$[7.4 \quad 15.4 \quad 7.2 \quad 21.2]$$

$$[6.4 \quad 41.6 \quad 19.2 \quad 83.2]$$

Answer: c

- 5) Two matrices, U and V have dimension 4x4. Mark True or False.
- (____) $T = U \cdot V$, then T is 8x8.
 - (____) $T = U \cdot V$, then T is 2x2.
 - (____) Considering a vector v being 4x3, then $T = U \cdot V \cdot v$. The matrix T , in this case, has dimension 4x3.
 - (____) $U \cdot V = V \cdot U$

Answer and Explanation:

- (False) The dimensions of the resultant matrix is the same as the outer dimensions of the matrices being multiplied. In this case the outer dimensions are 4 and 4, so T is 4x4.
- (False) Same explanation as above.
- (True) According the associative property, $(U \cdot V) \cdot v$ is the same as $U \cdot (V \cdot v)$. The resultant matrix of $U \cdot V$ is 4x4, and when multiplying this one by 4x3 the result will be 4x3.
- (False) According the non-commutative property, in general changing the order of the matrices in multiplication yields different results or may not be operable if the (inner) dimensions doesn't match.

7

Linear Regression With Multiple Inputs/ Features

CONTENTS

7.1. Gradient Descent For Multiple Variables Linear Regression.....	134
7.2. Normal Equation	137
7.3. Programming Exercise: Linear Regression With Single Input And Multiple Inputs.....	140

Chapter 6 described the theoretical principles with practical examples of linear regression using one variable as input. In such case it was mentioned the example of estimating housing price using its size as a predictor (input).

In this chapter, we will consider the expanded version of such model, i.e., the linear regression model using multiple inputs. *For example*, consider as an example the prediction of housing prices in a certain neighborhood. Instead of relating it only with the housing size, there are other obviously factors that affects the price of a house, such as the number of bedrooms and bathrooms, age of home, number of floors, etc. In this case, it is assumed that instead of having the price as a function of the size only, it becomes function of multiple variables.

Table 7.1: Example of Multiple Feature House Pricing Table

Nr. Bedrooms	Nr. Floors	Size (m^2)	Age (years)	Price (1000 US\$)
5	2	100	12	360
2	1	300	40	412
3	3	250	20	210
...

Each of the features used to predict the price in the table above is an input x_i , and the price is the output $y = y(x_0, x_1, x_2)$. In Table 7.2 the above table is rewritten incorporating the mathematical labels to each feature. A index column is also added.

Table 7.2: Example 2 of Multiple Feature House Pricing Table

i	Nr. Bedrooms x_1	Nr. Floors x_2	Size (m^2) x_3	Age (years) x_4	Price (1000 US\$) y
1	5	2	100	12	360
2	2	1	300	40	412
3	3	3	250	20	210
...

The above table can help to understand some primer concepts in linear regression modeling using multiple features. The vector $x^{(i)}$ is the input vector (features) of the i th training example. For instance, consider the

vector $x^{(2)}$.

$$x^{(2)} = [2 \ 1 \ 300 \ 40]^T$$

Note that the value of the price in $i = 2$ is not added to the vector $x^{(2)}$, since this variable is not an input, but an output. Also, note that the operator T is used to denote transpose operation. This is because, by convention, the features vector is represented as a column-vector.

One can also refer to single elements of the input vector by using a subscript index, as in x_1^2 . In 1-index based system (in contrast with 0-index based), this would indicate the 1st element of the vector x^2 , which is the value 2. In 0-index based system, x_1^2 would indicate the 2nd element of the vector, while x_0^2 would refer to the first one.

In linear regression with one input, the model was written as,

$$y = a_0 + a_1 x$$

Intuitively, in multiple input linear regression the model is written by incorporating the other variables as multiplying each one by a parameter a_j , where j refers to each feature.

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \dots + a_j x_j$$

Observe that the parameter a_0 is not directly multiplied with any input variable. This is because a_0 is considered a **bias** of the model, i.e., even if all the features have value 0 (zero), the output y may not be zero if $a_0 \neq 0$. For consistency, add a *dummy* variable $x_0 = 1$ so each parameter a_j is systematically multiplied by a feature x_j , where $j = 0, 1, 2, \dots$

$$y = a_0 x_0 + a_1 x_1 + \dots + a_j x_j$$

Notice the use of the equation with the example mentioned in the beginning of this chapter (house pricing), at $i = 2$.

$$y = a_0 1 + 2a_1 + 1a_2 + 300a_3 + 40a_4$$

Some observations can be stated from the equation above and generalized to any linear regression model:

- The value of the five parameters is initially unknown. They can be determined by coupling five linearly independent equations, in which case it will yield a unique exact solution to the model, with perfect fit. When this happens, it is said that the system is **Determined**. With less than 5 records, the system is **Undetermined** and there are infinite solutions, i.e., no solution can be stated to be correct. Generalizing, for a linear model with j features, it is necessary **at least** $j+1$ records i to obtain a solution, with the special case that if the number of records is exactly equal to $j+1$, then there is a single, perfect solution.
- In the case that there are more than the necessary points to obtain a **determined** system, then the model can be adjusted to minimize the errors, characterizing an **Overdetermined** system. In this case, the solution found is not perfect (mathematically there is no exact solution) unless some of the points are colinear, summing $j+1$ linearly independent points.

In matrix form, the multiple linear regression model can be written as follows.

$$Y = AX$$

where Y is a column-vector of all the output records, A is a row-vector of parameters and X is a matrix of features, with records being stored in columns and features in rows.

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_n^{(2)} \\ \dots & \dots & \dots & \dots & \dots \\ x_1^{(j)} & x_2^{(j)} & x_3^{(j)} & \dots & x_n^{(j)} \end{bmatrix}$$

7.1. GRADIENT DESCENT FOR MULTIPLE VARIABLES LINEAR REGRESSION

The calibration (training) of the parameters a_j in the multiple linear

regression model follows the same principle as in the single variable linear regression, with the additional requirement of calibrating more parameters at each iteration.

As in the single variable case, the cost function J consists in the sum of the squared errors, i.e., differences between predicted and observed. In matrix form,

$$J = \sum_{i=1}^m (AX - Y)^2$$

The algorithm consists in updating the elements of A simultaneously, using the gradient of cost J to “point” the direction, and a learning rate α so to keep each update step smaller than the gradient (or higher if $\alpha > 1$).

$$a_j^{t+1} = a_j^t - \alpha \sum_{i=1}^m (AX - Y)^2 x_j$$

where t is the iteration step.

7.1.1. Feature scaling

In most applications, feature values drastically differs in magnitude and values. Such difference may distort the cost function and cause problems when performing the training. Such problem is even worst when performing classification, since most models rely on measurements of the distance between points. Therefore, there is a need of normalizing/ standardizing the data. This approach also assists when applying gradient descent, since the convergence is already proved to be faster than without rescaling (Ioffe and Szegedy, 2015).

There are different methods to perform features scaling, but all of them relies n the same principle: to have all the features with the same scale.

7.1.1.1. Min-Max Normalization

The idea of min-max normalization or min-max scaling method is to change the feature range to $[0,1]$ or $[-1,1]$. The equation above provides the general formula of this approach.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x' is the rescaled variable, $\min(x)$ is the minimum value of the variable x in the training set, $\max(x)$ is the maximum value of x in the training set.

For example, consider the house age feature (in years). If the maximum age is 40 years and the minimum is 2 years, then in the rescaled house age 40 becomes 1, 2 becomes 0 and a house with age 20 becomes $(20-2)/(40-2) = 18/38 = 9/19$ or approximately 0.474.

7.1.1.2. Mean Normalization

With a slight difference from the min-max normalization, mean normalization consists in subtracting the average value of x .

$$x' = \frac{x - \text{mean}(x)}{\max(x) - \min(x)}$$

Note that in this case, the range will not be exactly [0,1] or [-1,1]. It will depend on the value of the average and the spreading of the data (difference between maximum and minimum).

Example: Again, consider the house age feature (in years). Training data is shown below.

$$x = [40, 30, 20, 22, 25, 32, 38, 21]$$

From this dataset, the following values are obtained:

$$\max(x) = 40$$

$$\min(x) = 20$$

$$\text{mean}(x) = \frac{1}{8}(40 + 30 + 20 + 22 + 25 + 32 + 38 + 21) = 28.5$$

Thus the normalized value x' for 40 years, for instance, becomes 0.575 (observe that, in min-max that would be equals to 1).

7.1.1.3. Standardization

Consists in transforming the data so it has zero-mean and unit variance.

$$x' = \frac{x - \text{mean}(x)}{\text{std}(x)}$$

where $\text{mean}(x)$ is the original mean of the data and $\text{std}(x)$ is the original standard deviation.

7.1.1.4. Scale to Unit Length

The concept behind unit length scaling consists in transforming the complete feature vector (instead of each feature) in a one-length vector. In most cases, it corresponds to use the Euclidean distance as the vector length metric, though other metrics can also be used.

$$x' = \frac{x}{\|x\|_2}$$

7.1.2. Learning Rate Choice

Since the learning rate controls the “step size” in the GD algorithm, it can define if the solution is encountered or if the iterations diverge, with no actual solution being found.

Such evaluation can be done by checking if the error reduces at each iteration step. If it doesn’t, probably the learning rate must be adjusted (reduced) until convergence is observed. However, if α is too small, gradient descent may be too slow to converge. Therefore, it is necessary to find a balance to α , which will depend on each problem.

In summary, if α is too small, GD converges be too slow. On the other hand, if it is too large $J(\theta)$ may not converge, or it may also converge slowly.

Usually, α values are evaluated per order of magnitude, such as:

$$10^{-3}, \dots, 10^{-2}, \dots, 10^{-1}, \dots, 10^0, \dots$$

7.2. NORMAL EQUATION

The normal equation method differs in its essence from the GD method. While the latter searches for the solution using an iterative approach, the first finds the optimum parameters **analytically**. This means that no iteration

is needed, and the solution is theoretically always found, with no divergence as it occurs with GD method.

The derivation starts from the general multiple inputs linear regression model.

$$y(x) = a_0x_0 + a_1x_1 + a_2x_2 + \dots + a_jx_j$$

The solution can be find by minimizing the least-square residuals, expressed in the cost function J .

$$J(a_0, a_1, \dots) = \sum_{i=1}^m (y(x^i) - y_{obs}^i)^2$$

Naturally, the problem can be represented using matrix notation, since it consists on a system of linear equations. The coefficients to be found are the vector

$$\begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_j \end{bmatrix}$$

The linear regression model, as previously shown, can then be expressed in the matrix form.

$$Y(X) = A^T X$$

The features matrix, in this case, consists of m rows, where each row is a sample i (the sample $x(i)$). In this way, the cost function can be rewritten using matrix multiplication.

$$J(A) = (XA - Y_{obs})^T (XA - Y_{obs})$$

Rewriting,

$$J(A) = ((XA)^T - Y_{obs}^T)(XA - Y_{obs})$$

Performing the distributive multiplication,

$$J(A) = (XA)^T (XA) - (XA)^T Y_{obs}^T - Y_{obs}^T (XA) + Y_{obs}^T Y_{obs}$$

Both XA and Y_{obs} are vectors, so the order is not relevant as long as the dimensions are corresponding. Performing a simplification of the equation above one obtains,

$$J(A) = A^T X^T X A - 2(XA)^T Y_{obs} + Y_{obs}^T Y_{obs}$$

Recall that, to obtain the minimum of the cost J , it is necessary to derive with respect to the parameters A and make it equal to zero. Performing the derivation of the matrix equation above may not seem so simple, but it is done in the same way as a scalar equation.

$$\frac{\partial J}{\partial A} = 0 = 2X^T X A - 2X^T Y_{obs}$$

Rearranging,

$$X^T X A = X^T Y_{obs}$$

At this step, it is incorporated a **very important assumption**. Let $X^T X$ be **invertible**. In such case, both sides can be multiplied by $X^T X$, obtaining the Normal Equation in its final form.

$$A = (X^T X)^{-1} X^T Y_{obs}$$

The assumption mentioned just above ($X^T X$ be **invertible**) is a crucial point of the Normal Equation solution method. It may justify the use of GD method for big linear regression problems, since the inversion of $X^T X$ can be computationally intensive or even non-feasible (not invertible). It may come to the point that this analytical method becomes slower than the GD method for very large number of samples. Table 7.3 summarizes both approaches pros and cons.

Table 7.3: Comparison Between Gradient Descent and Normal Equation

Gradient Descent	Normal Equation
Choice of α	No need of α
Iterative method	Analytical method (no iteration)
Works even for large datasets	May become slow for large datasets
–	Need to compute $(X^T X)^{-1}$

Can be used to solve any linear regression optimization

$(X^T X)^{-1}$ must be invertible

Consider the case where $X^T X$ is not invertible. This may happen because the matrix is singular or degenerate. The following possibilities should be taken into consideration:

- Redundant features (linearly dependent). Example: Expressing the same variable in two different units, as in size (meters) and size (inches). In such case, one of the linearly dependent variables must be deleted.
- Excessive features, such that the number of features is greater than the number of records or samples. In such case, features must be eliminated or one can use regularization.

7.3. PROGRAMMING EXERCISE: LINEAR REGRESSION WITH SINGLE INPUT AND MULTIPLE INPUTS

7.3.1. Introduction

In this section, we develop a Python program to solve the linear regression model for (a) one input and (b) multiple inputs. An example dataset illustrates the performance and predictability of the model.

Before reading these examples, it is strongly recommended to go through the basics of Python coding so as to get a better understanding of the code structure and how the algorithm is translated into code.

The problem is structured in a series of files contained in **Appendix A.3**. The heading of each file is the suggested file name as it is referred along this section.

Files included in this exercise (Appendix A.3)

- `example1.py` – A Python script that steps through the exercise for single input;
- `example1_multi.py` – Python script that steps through the exercise for multiple inputs.

In the first part (linear regression with single input), we step through the file `example1.py`, which generates a hypothetical dataset, process it and use Python functions to fit a linear regression model to the dataset. In a similar way, `example1_multi.py` performs similar operations to fit a linear

regression to the dataset with multiple inputs.

7.3.2. Linear Regression with Single Inputs

The owner of a fast-food restaurant wants to open a new store in another city. To do that, it was collected data for profits and populations from different cities where a restaurant was already established. Therefore, the task consists in the implementation of linear regression model with single input to predict the restaurant profit according the city's population.

Then the data will be used to select to which city the restaurant should expand next.

7.3.2.1. Importing Libraries

To work in this problem, the following libraries are necessary:

- numpy: for numerical computing;
- matplotlib: for visualization (plotting).

From the library *matplotlib*, we use specifically *pyplot* (for plotting) and *PdfPages* to generate a hard copy of the plot as a pdf file. All the code to import the libraries is placed on the top of the file *example1.py*

#%-----Part 0: Importing Libraries-----

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
```

7.3.2.2. Generating the Data and Plotting

The file *example1.py* contains the instructions to generate the data for the linear regression problem. Naturally, in real-world problems the data is not generated but imported from a file, since it is collected from real research. However we chose to generate it for easy reproducibility. One data (X) is the population of a city and the second data (y) is the profit of a restaurant in that city. A negative profit value indicates loss. A random noise between 0 and 10 is added to the profit data to give it a more “real” shape (not perfect fit).

After generation, the data is plotted and a hard copy of the plot is generated as a pdf file.

#%-----Part 1: Data Generation and Plotting-----

```
np.random.seed(999)
X = np.stack((np.ones((100,)), np.random.exponential(5, 100))).T
y = X.dot([-4, 1.2]) + np.random.rand(100)*10
with PdfPages('example1_fig1.pdf') as pdf:
    plt.figure()
    plt.plot(X[:,1], y, 'b.')
    plt.xlabel('population (Thousands)')
    plt.ylabel('profit (1000 USD)')
    plt.grid()
    pdf.savefig()
    plt.show()
```

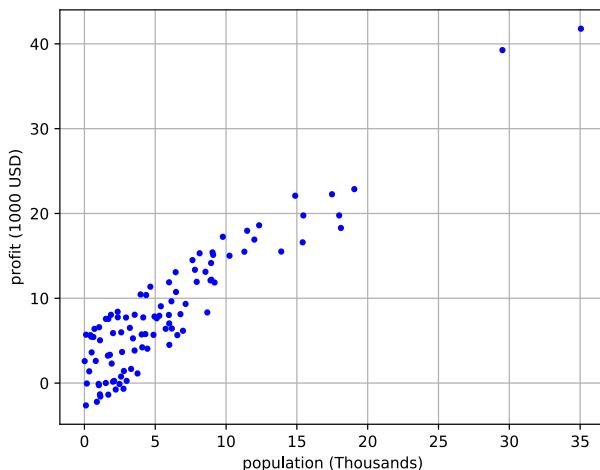


Figure 7.1: Scatter plot of profit x population data.

7.3.2.3. Gradient Descent – Compute the Cost $J(a)$

As the iterations of the GD algorithm progress, it is useful to monitor the value of the cost function $J(a)$, so as to know if the process converged. For this purpose, the function `computeCost(X,y,a)` is implemented, which takes

as inputs the vector X , output y and parameters a to compute the cost function $J(a)$ as previously defined.

```
def computeCost(X, y, a):
```

```
    ""
```

```
J = computeCost(X, y, theta):
```

Computes the cost for linear regression using the parameters a to fit points in X and y dataset

```
    ""
```

```
J = 1/(2*len(y))*np.sum((X.dot(a) - y)**2)
```

```
return J
```

7.3.2.4. Gradient Descent

With the function to compute the cost, the gradient descent algorithm can be implemented using a function. The gradientDescent function receive as inputs the input vector X , the observed output values y , an initial guess on the parameters a , the maximum number of iterations allowed MAX_ITER and a Boolean input verbose which indicates if the iteration process should be written to the screen.

```
def gradientDescent(X, y, a, alpha, MAX_ITER, verbose = False):
```

```
    ""
```

```
gradientDescent(X, y, theta, alpha, MAX_ITER)
```

Computer the optimum a values (parameters) using GD algorithm.

Inputs:

X – input vector (1D numpy array)

y – output vector (1D numpy array)

a – initial guess of parameters (2x1 numpy array)

alpha – learning rate (float)

MAX_ITER – maximum number of iterations (int)

verbose – True to print the iteration process, False otherwise

Returns:

a – optimum parameters

”

iter_ = 0 # initial iteration step

J_history = [] # a list to store the cost at each iteration

J_history.append(computeCost(X, y, a)) # store the cost at each iteration

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4s}\t{a[0]:4s}\t{a[1]:4s}\t{J_history:4s}")

while iter_ < MAX_ITER:

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4d}\t{a[0]:4.2f}\t{a[1]:4.2f}\t{J_history[-1]:4.2f}')

dJda0 = np.sum((a[0] + a[1]*X[:,1] - y)) # partial derivative of J / a0

dJda1 = np.sum((a[0] + a[1]*X[:,1] - y)*X[:,1]) # partial derivative of

J / a1

update the-parameters

a[0] = a[0] – alpha*dJda0

a[1] = a[1] – alpha*dJda1

iter_ += 1

J_history.append(computeCost(X, y, a)) # store the cost at each iteration

return a,J_history

The next step consists in evaluating the algorithm implemented above on the dataset provided. To do so, initialize the parameters a as a vector of zeros (one value for a_0 and other for a_1). Then set the number of iterations MAX_ITER and the learning rate ALPHA.

```
a = np.zeros((2, )) # initial gues on parameters  
# GD parameters  
MAX_ITER = 1500  
ALPHA = 0.0002
```

Once these parameters are set, we can run the gradient descent algorithm by calling the function `gradientDescent`. The iteration process can be visualized as it runs by passing the `verbose` argument as `True`. Then we can print the final results by using `print()` statement and plotting the final result of the linear regression.

```
# run GD  
a, J_history = gradientDescent(X, y, a, ALPHA, MAX_ITER, True)
```

```
# print final result to screen  
print('Final parameters')  
print(f'a0 = {a[0]:.2f}')  
print(f'a1 = {a[1]:.2f}')
```

```
# plot the data and the linear fit  
with PdfPages('example1_fig2.pdf') as pdf:  
    plt.figure()  
    plt.plot(X[:,1], y, 'b.')  
    plt.plot(X[:,1], X.dot(a), 'r')  
    plt.xlabel('population (Thousands)')  
    plt.ylabel('profit (1000 USD)')  
    plt.legend(['Observed', 'Predicted'])  
    plt.grid()  
    pdf.savefig()  
    plt.show()
```

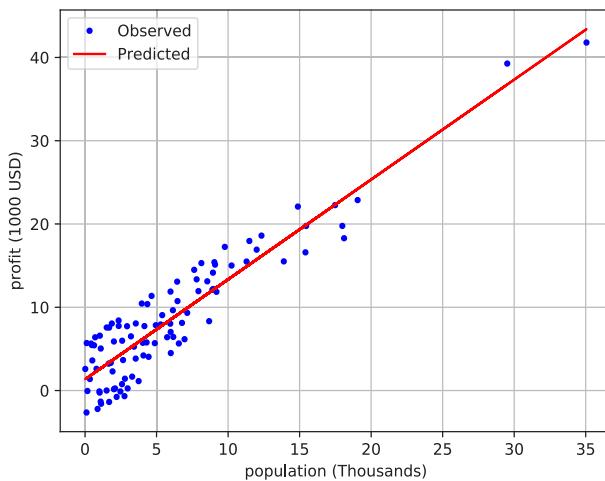


Figure 7.2: Plot of observed and predicted profit x population data.

One can visually note that the linear regression looks reasonable for the data and that predictions are relatively accurate, though a precise statement could only be done by using some accuracy metrics. Predictions can be done by choosing an appropriate input with the bias.

#%% -----Part 5: Perform predictions -----

```
# predict the profit for a population of 5000
```

```
yhat = np.array([1, 5]).dot(a)
```

```
print(f"For a population of 5000, the predicted profit is {yhat:.2f} Thousands USD.")
```

7.3.4. Linear Regression with Multiple Inputs

In this second part, we implement the same model (linear regression) but with the possibility of dealing with multiple inputs, as may as they may be. For that, some modifications are done in the code of *example1.py* so as to obtain the code *example1_multi.py*. The modifications are as follows.

7.3.4.1. Data Generation

Data in this example consists of 2 input vectors (+bias) and one output. For easy reproducibility, the data is generated directly in the script, as it was done for the linear regression with single input.

```
#%% -----Part 1: Data Generation and Plotting-----
```

```
np.random.seed(999)
```

```
X = np.stack((np.ones((100,)),np.random.exponential(5, 100),np.random.exponential(30, 100))).T
```

```
y = X.dot([-7, -1.2, 3])+ np.random.rand(100)*100
```

```
with PdfPages('example1_multi_X1.pdf') as pdf:
```

```
    plt.figure()
```

```
    plt.hist(X[:,1])
```

```
    plt.xlabel('population (1000)')
```

```
    plt.ylabel('Frequency')
```

```
    plt.grid()
```

```
    pdf.savefig()
```

```
    plt.show()
```

```
with PdfPages('example1_multi_X2.pdf') as pdf:
```

```
    plt.figure()
```

```
    plt.hist(X[:,2])
```

```
    plt.xlabel('Income per capita')
```

```
    plt.ylabel('Frequency')
```

```
    plt.grid()
```

```
    pdf.savefig()
```

```
    plt.show()
```

```
with PdfPages('example1_multi_fig1.pdf') as pdf:
```

```
    plt.figure()
```

```
    plt.plot(np.sort(y), 'b.')
```

```
    plt.xlabel('data point')
```

```
    plt.ylabel('profit (1000 USD)')
```

```
plt.grid()  
pdf.savefig()  
plt.show()
```

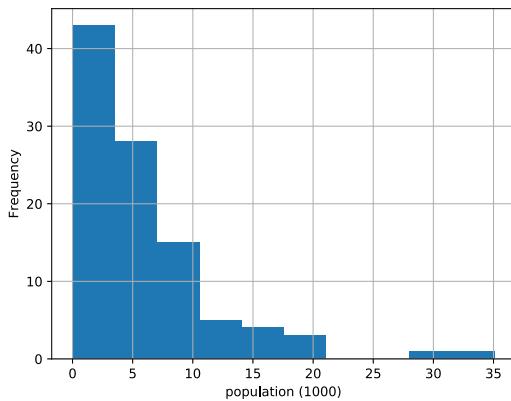


Figure 7.3: Histogram of input feature 1.

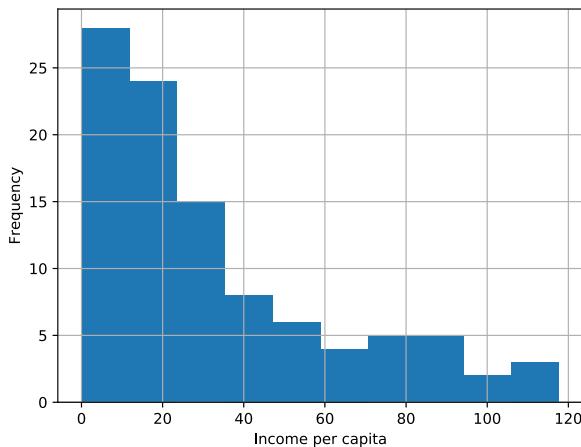


Figure 7.4: Histogram of input feature 2.

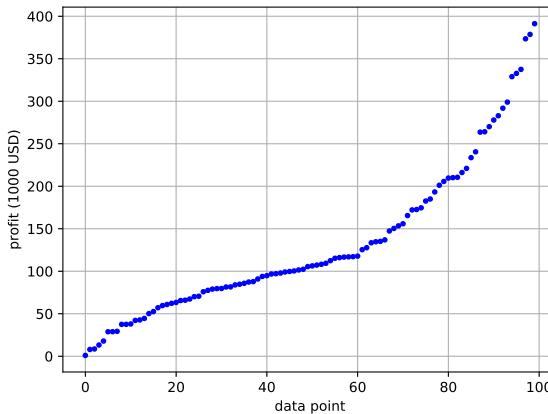


Figure 7.5: Scatter plot of profit for multiple input example.

7.3.4.2. Cost Function

The function `computeCost(X,y,a)` requires slight modification to work with multiple inputs. Using matrix multiplication notation, it can be used for any number of inputs with no restriction.

#%% -----Part 2: Cost Function-----

```
def computeCost(X,y,a):
```

```
    ""
```

```
J = computeCost(X, y, theta):
```

*Computes the cost for linear regression using the parameters a
to fit points in X and y dataset*

```
    ""
```

```
J = 1/(2*len(y))*np.sum((X.dot(a) - y)**2)
```

```
return J
```

7.3.4.3. Gradient Descent

The modification of `gradientDescent` function consists in:

- in the print statement, remove the parameters as it may not be feasible to print parameters in every case (e.g., 10,000 parameters);
- keep track of the change on the cost function, so as to interrupt the iterative process when the cost has converged (the difference between the previous and the actual one is very small).
- use matrix notation and vectorized approach to obtain both the derivatives of cost dJ / da and the updated values of a .

All these modifications are gathered in the modified function `gradientDescent`.

#%% -----Part 3: Gradient Descent-----

def gradientDescent(X, y, a, alpha, MAX_ITER, verbose = False):

""

gradientDescent(X, y, theta, alpha, MAX_ITER)

Computer the optimum a values (parameters) using GD algorithm.

Inputs:

X – input vector (1D numpy array)

y – output vector (1D numpy array)

a – initial guess of parameters (2x1 numpy array)

alpha – learning rate (float)

MAX_ITER – maximum number of iterations (int)

verbose – True to print the iteration process, False otherwise

Returns:

a – optimum parameters

""

iter_ = 0 # initial iteration step

J_history = [] # a list to store the cost at each iteration

J_history.append(computeCost(X, y, a)) # store the cost at each iteration

print the iteration process if verbose is True

if verbose == True:

print(f"{'iter':4s}\t\t{'J':4s}")

```
J_old = computeCost(X,y,a)*1000
while iter_ < MAX_ITER and abs(J_old-J_history[-1]) > 1e-5:

    # print the iteration process if verbose is True
    if verbose == True:
        print(f'{iter_:4d}\t{J_history[-1]:.2f}')

    dJda = np.zeros((X.shape[1],))
    for i in range(X.shape[1]):
        dJda[i] = np.sum((X.dot(a)-y)*X[:,i])

    # update the-parameters
    a -= alpha*dJda

    iter_ += 1
    J_old = J_history[-1]
    J_history.append(computeCost(X, y, a)) # store the cost at each iteration

return a, J_history
```

7.3.4.4. Feature Normalization

Since we are dealing with inputs with different scales, it is a good practice to scale them to equal ranges. One way of doing so is normalizing, so the minimum is 0 and the maximum is 1 for all features (except bias).

```
# feature normalization
```

```
X[:,1:] = (X[:, 1:] - np.min(X[:, 1:], axis=0))/(np.max(X[:,1:], axis = 0) -
np.min(X[:,1:], axis = 0))
```

The rest of the code resembles very similarly the linear regression with single input, with small differences. The complete code can be read in Appendix A.3, under *example1_multi.py*. Figure 7.5 shows a comparison with the final prediction and the observed values.

```
# plot the data and the linear fit
with PdfPages('example1_multi_fig2.pdf') as pdf:
    plt.figure()
    plt.plot(np.sort(y), 'b.')
    plt.plot(np.sort(X.dot(a)), 'r')
    plt.xlabel('data point')
    plt.ylabel('profit (1000 USD)')
    plt.legend(['Observed', 'Predicted'])
    plt.grid()
    pdf.savefig()
    plt.show()
```

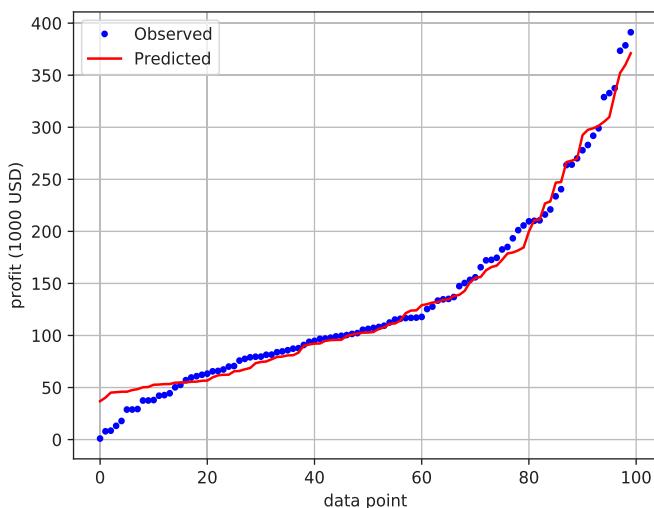


Figure 7.6: Plot of observed and predicted profit for multiple inputs example.

7.3.4.5. Normal Equation

The solution of the problem using the Normal Equation can be found using a single line of code.

```
a = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
```

Clearly this approach has several advantages when compared with GD as already stated. The main cons are that the matrix inversion is computationally expensive and may make unfeasible the solution for problems with higher dimensions.

8

Classification Using Logistic Regression Model

CONTENTS

8.1. Logistic Regression Model Structure.....	156
8.2. Concept Exercise: Classifying Success In Exam According Hours Of Study	158
8.3. Programming Exercise: Implementation Of The Exam Results Problem In Python From Scratch	160
8.4. Bonus: Logistic Regression Using Keras.....	165

In the previous chapters, we have seen the linear regression model, a technique used to predict continuous values given some input(s). For classification purposes, this model is not appropriate. This becomes clear by considering the following hypothetical problem: prediction of cold in patients. The output of this problem is the health condition of the patient, which can fall into two classes: sick or healthy. These two classes can be transformed into one, i.e., the health condition of the patient, where $y = 0\%$ healthy means the patient is sick and $y = 100\%$ healthy means the patient has no sickness.

The problem of using linear regression in such case is that this model generates continuous values as outputs, for example $y = 200$, which would have no meaning in such case. That is where Logistic Regression shows to be more appropriate model, as it will be shown along this chapter.

According Susan Li (2017), categorical variables can be predicted using the machine learning algorithm Logistic Regression. This algorithm generates as output the probability of a categorical dependent variable. This variable is binary in the sense that it contains data coded in the domain $[0,1]$, where 1 usually means positive (yes, success, etc.) and 0 means negative (no, failure, etc.), though the result is not mathematically restricted to this interpretation. This means that the logistic regression model generates a probability $P(Y=1)$ as a function of the input(s) X .

Some assumptions are made when employing logistic regression models. The first one to be mentioned is that the dependent variable must be binary, since logistic regression is essentially binary. Per convention, as already mentioned, the factor level 1 of the dependent variable should be a representation of the desired outcome (positive) so as to avoid confusion. As per every machine learning model, only meaningful variables should be considered. Ideally, no multicollinearity should be present, i.e., the independent variables should not be dependent one of the other. For an accurate fitting, large datasets are required.

8.1. LOGISTIC REGRESSION MODEL STRUCTURE

Consider the linear relationship between the predictor variables and the log-odds of the event that $y = 1$ (positive), as stated below.

$$l = \log_b \frac{p}{1-p} = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

where: l – log-odds, b – base logarithm, p – probability, (0 to 1 exclusive), and β_n – model parameters.

Exponentiation of the above equation reveals the probability inside the log function.

$$\frac{p}{1-p} = b^{\beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n}$$

Performing some algebraic manipulation, the probability can be isolated from the above equation.

$$p = \frac{b^{\beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n}}{b^{\beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n} + 1} = \frac{1}{1 + b^{-\beta_0 x_0 - \beta_1 x_1 - \beta_2 x_2 - \dots - \beta_n x_n}}$$

According the above formula, once the values of the parameters β are known, one can estimate the probability p of an event by knowing the predictors x . The base b may be any value, though it is usually employed e , 10 or 2.

For instance, let $b = 10$ and coefficients $\beta_0 = -3$, $\beta_1 = 2$ and $\beta_2 = 4$. The model can be written as follows by substitution of values.

$$\log_{10} \frac{p}{1-p} = -3x_0 + 2x_1 + 4x_2$$

Consider that $x_0 = 1$ for every model (bias). Thereof, some inferences can be stated about this model:

$\beta_0 = -3$ is the **y-intercept** of the model, i.e., the odds of success ($y=1$) when $x_1 = x_2 = 0$. This odd can be found through substitution of the values in the rearranged equation as follows.

$$p = \frac{1}{1 + b^{-\beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2}} = \frac{1}{1 + 10^{(-3+2\times0+4\times0)}} = \frac{1}{1+10^3} \approx 10^{-3}$$

$\beta_1 = 2$ means that the log-odds increase by 2 when x_1 increases by one. In other words, increasing x_1 by one increases the log-odds of y by a factor of 10^2 .

$\beta_2 = 4$ means that the log-odds increase by 4 when x_1 increases by 1. In other words, increasing x_1 by one increases the log-odds of y by a factor of 10^4 . Therefore, it can be stated that the effects of x_2 are double the effects of x_1 . The effect on the log-odds is 100 greater.

The estimation of the parameters β is done via **logistic regression**.

8.2. CONCEPT EXERCISE: CLASSIFYING SUCCESS IN EXAM ACCORDING HOURS OF STUDY

In this exercise, logistic regression will be used to investigate the following problem:

How the hours dedicated to study affect the success on an exam? To answer such question, a teacher collected the results of 20 students and asked them how many hours they have dedicated to study. The logistic regression model is used to indicate the probability of passing an exam given the hours of study.

Table 8.1 summarizes the data collected from the 20 students. In the row “Pass,” 1 indicates success (passed the exam) while 0 indicates fail.

Table 8.1: Hours of Study and Exam Result for 20 Students

Hours	0.45	0.75	1.00	1.25	1.75	1.50	1.30	2.00	2.30	2.50	2.80	3.00	3.30	3.50	4.00	4.30	4.50	4.80	5.00	6.00
Pass	0	0	0	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	1	1

Suppose that, after running a program, the teacher obtained the following results for the logistic regression model (not exact).

Table 8.2: Logistic Regression Results for Hours of Study and Exam

	Coefficient	Standard Error	z-value	p-value (Wald)
Intercept	-4	1.8	-2.3	0.02
Hours	1.5	0.6	2.4	0.02

At a level of significance of 5%, it can be stated that the hours of study indeed affects the results of an exam (p-value 0.02). The coefficients of the logistic regression, according the table above are $\beta_0 = -4$ and $\beta_1 = 1.5$. The equation of the logistic regression can be then written by incorporating such results.

$$\ln \frac{p}{1-p} = 1.5\text{Hours} - 4 = 1.5(\text{Hours} - 2.7)$$

$$p = \frac{1}{1+e^{-(1.5h-4)}}$$

where h is hours of study (in decimals, i.e., 1.8 is 1:48). Because $\beta_1 = 1.5$, each hour of study adds log-odds of passing by 1.5 or odds or $e^{1.5} = 4.5$.

Since the x-intercept is 2.7, it means that the logistic model estimates even odds (50% success probability) for a student who studied 2.7 hours. 2 hours of study would give the student the following probability of passing the exam:

$$p = \frac{1}{1+e^{-(1.5\times 2-4)}} = 0.27$$

On the other hand, for a student who dedicated 5 hours of study,

$$p = \frac{1}{1+e^{-(1.5\times 5-4)}} = 0.97$$

Table 8.3 summarizes the probabilities of passing the exam for some hours of study according the logistic regression model. Notice that the model result is significant at 5% level.

Table 8.3: Summary of Probabilities

Hours of study	Log-odds	Odds	Probability
1	-2.50	0.08	0.08
2	-1.00	0.37	0.27
3	0.50	1.65	0.62
4	2.00	7.39	0.88
5	3.50	33.12	0.97

8.3. PROGRAMMING EXERCISE: IMPLEMENTATION OF THE EXAM RESULTS PROBLEM IN PYTHON FROM SCRATCH

In this section, we implement the logistic regression model in Python using only basic libraries for numerical computation(numpy) and data visualization (matplotlib) for the plots. Though it can be found numerous implementations of such simple model in Python, a basic model can better help one to understand on how logistic regression model works.

8.3.1. Dataset

For this exercise we will use the already mentioned dataset comprehending the hours of study and the results (pass or fail) of 20 students.

hours of study

```
X = np.array([0.45, 0.75, 1.00, 1.25, 1.75, 1.50, 1.30, 2.00, 2.30, 2.50, 2.80,  
3.00, 3.30,  
3.50, 4.00, 4.30, 4.50, 4.80, 5.00, 6.00]).reshape(-1, 1)
```

results (0 – fail; 1 – pass)

```
y = np.array([0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1])
```

8.3.2. Algorithm

We will implement the logistic regression equation (also known as sigmoid) using matrix notation for better performance and readability. Consider x the input vector containing the feature(s) and bias as a 1×2 vector. The component z can be defined as the matrix multiplication of the parameters θ and the input vector x

$$z = \theta^T x$$

The sigmoid equation $g(\theta x)$ can be written in terms of z as follows.

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

```
def g(theta,X):  
    z = np.dot(X, theta)  
    return 1 / (1 + np.exp(-z))
```

8.3.3. Loss Function

The training process consists on determining the best θ parameters of the logistic regression model for the given dataset. These parameters are initially set at random values and updated iteratively using an optimization process, such as Gradient Descent algorithm. The metrics used to measure goodness of fit is the loss function J .

def J(h, y):

```
return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()
```

8.3.4. Gradient Descent

The parameters are updated using gradient descent, which measures the magnitude and distance of each parameter update through the derivatives of the loss function J .

$$\frac{\partial J}{\partial \theta_i} = \frac{1}{m} \left(g(\theta^T X) - y \right) X$$

dJdtheta = np.dot(X.T, (h(z) - y)) / len(y)

A learning rate is used to control the size of the step during the optimization process, to guarantee convergence.

$$\theta = \theta - \alpha * \frac{\partial J}{\partial \theta}$$

alpha = 0.1

theta -= alpha * dJdtheta

The process portrayed above is repeated several times under the value of the parameters θ converges, i.e., the changes between each iterations becomes negligible.

8.3.5. Prediction

When the logistic regression is already trained, it can be used to perform predictions on the data, i.e., answer the question: What is the probability of the output given input(s) ...? Similarly, it can be assumed that probabilities higher than a threshold, say 0.5 (50%), can be considered as success (100%) while probabilities lower than 0.5 (50%) are considered failure (0%). The threshold is not fixed and will depend on the problem.

def predict_probability(X, theta):

```
    return g(theta, X)
def predict(X, theta, threshold):
    return predict_probability(X, theta) >= threshold
```

8.3.6. Assembling All Together

We can write the logistic regression model using Object-Oriented Programming approach in Python. We assemble all the pieces mentioned above in a single class LogisticRegression, which encapsulates both the data defining the structure of the model and the methods (equations) to fit the parameters and perform predictions on data.

class LogisticRegression:

```
    def __init__(self, learning_rate=0.02, max_iterations=1000, fit_intercept=True):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.fit_intercept = fit_intercept

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def __loss(self, h, y):
        return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

    def fit(self, X, y, verbose = False):

        if self.fit_intercept:
            X = self.__add_intercept(X)

        # weights initialization
```

```
self.theta = np.zeros(X.shape[1])

i = 0
while i < self.max_iterations:
    z = np.dot(X, self.theta)
    h = self.__sigmoid(z)
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.learning_rate * gradient

    if(verbose == True and i % 100 == 0):
        z = np.dot(X, self.theta)
        h = self.__sigmoid(z)
        print(f'loss: {self.__loss(h, y)} \t')
        i += 1

def predict_prob(self, X):
    if self.fit_intercept:
        X = self.__add_intercept(X)

    return self.__sigmoid(np.dot(X, self.theta))

def predict(self, X, threshold):
    return self.predict_prob(X) >= threshold
```

A complete code can be seen in the Appendix A.4 – *Programming Exercise: Logistic Regression in Python*, in the subsection *logistic_simple.py*. Running this Python code in a command prompt yields the following results.

A plotting of the dataset is generated.

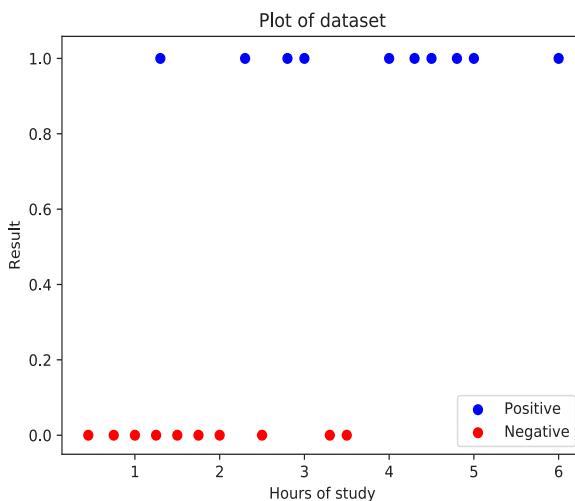


Figure 8.1: Dataset showing hours of study and exam results.

The following output is written in the screen.

```
loss: 0.6713380809713301
loss: 0.5324290007470507
loss: 0.4868393003271744
loss: 0.46622039560928474
loss: 0.45582653509145354
loss: 0.4501763170490853
loss: 0.44693809556844544
loss: 0.44501039759231287
loss: 0.4438302870072999
loss: 0.44309246345736497
time taken = 0.54 s
(preds == y).mean() =  0.8
theta = [-3.14101786  1.16595394]
Press enter to exit...
```

Analyzing such results, one can see that the initial loss is 0.67 and it reduces to approximately 0.44 after 1000 iterations (the printing is generated every 100 iterations). The time taken to run the model is 0.54 seconds, though

it could be severely reduced by changing the verbose variable to False. A final accuracy of 80% is achieved. This means that, for the 20 students, 16 results are correctly predicted, with a wrong prediction of the remaining four students. The final parameters obtained are shown in the θ variable. At the end, a plot of the dataset and the logistic regression prediction curve is shown, as illustrated in Figure 8.2.

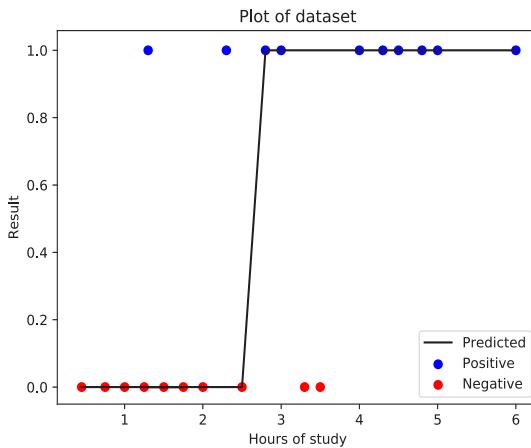


Figure 8.2: Dataset (points) and prediction (curve) showing hours of study and exam results.

8.3.7. Scikit Learn Solution

Compare now the solution obtained using sklearn module. The whole modeling approach is contained in 2 lines of code (plus the import command line and the predict()).

```
from sklearn.linear_model import LogisticRegression  
model = LogisticRegression(C = 1e10)  
model.fit(X, y)  
preds = model.predict(X)
```

8.4. BONUS: LOGISTIC REGRESSION USING KERAS

Another approach that can be used to implement the logistic regression model is to use Keras, a deep learning library for Python which works with TensorFlow or Theano as its backend. Models are building using a

modular approach (building blocks) thus making the procedure of creating complex models much easier than an implementation from scratch or using procedural programming.

```
model = keras.models.Sequential()
```

```
model.add(keras.layers.Dense(units=1, input_dim=1, activation='sigmoid'))
```

```
sgd = keras.optimizers.SGD(lr = 0.5)
```

```
model.compile(optimizer = sgd, loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
history = model.fit(X, y, epochs = 1000, verbose = 0)
```

```
ypred = model.predict(X).reshape(-1, 1)
```

9

Regularization

CONTENTS

9.1. Regularized Linear Regression	171
--	-----

Consider the problem of fitting a model to predict stock prices. The following plot illustrates such hypothetical data, with stock price (of a hypothetical asset) as a function of a feature x .

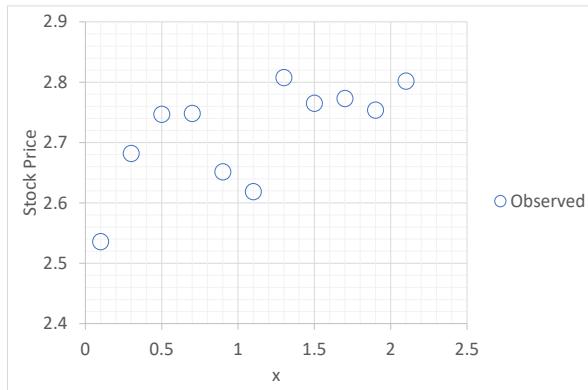


Figure 9.1: Stock price dataset.

Let three different models be fitted to this data: one linear, a quadratic and a n th-order polynomial model respectively as shown in the figures below.

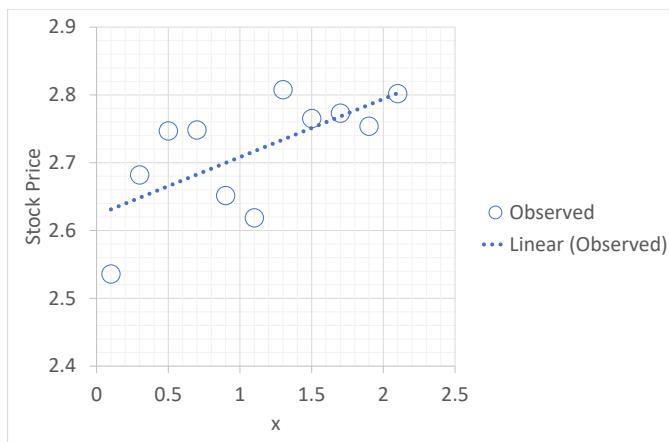


Figure 9.2: Stock price linear model $y = ax + b$.

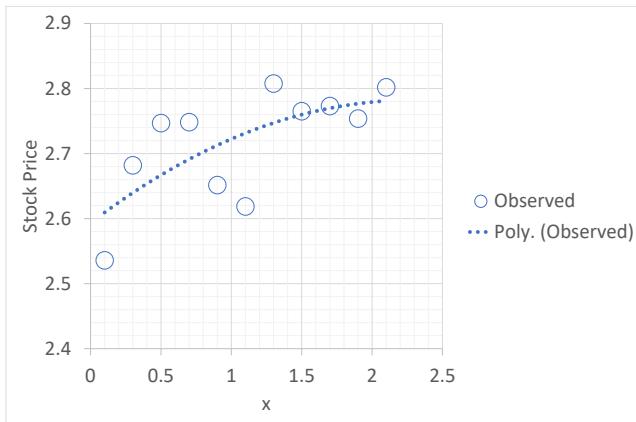


Figure 9.3: Stock price quadratic model $y = ax^2 + bx + c$.

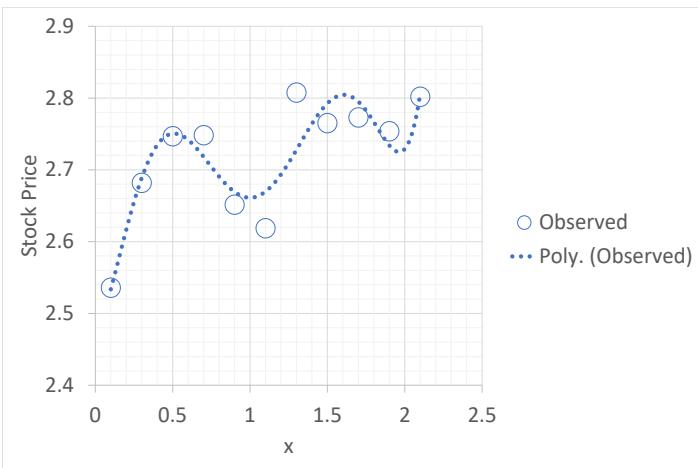


Figure 9.4: Stock price 6th order polynomial model $y = ax^6 + bx^5 + \dots + c$.

One can clearly see that the last model, with highest order, contains smaller errors, i.e., the predicted data is very similar to the observed one. However, this does not mean necessarily that this model is better than the others. In such case, it occurs *overfitting*, in which case the training set is very well fitted, but the model may fail to generalize to new data (testing set). Therefore, it is necessary to have a balance between goodness of fit and model generalization.

Overfitting can be reduced by following some steps:

- Reducing the number of features;
- Regularization.

Reduction of number of features can be achieved through manual selection of features to keep (require some *feeling* of what are the “best” or most important features); using a model selection algorithm, which determines which features are to be kept automatically.

Using regularization, one can keep all the features, but reduce the magnitude of higher-order coefficients, thus reducing the effect of these terms. This technique has been reported to work well even when working with many features, each of them contributing for the prediction of the output.

Regularization is applied in the cost function, through penalization of the coefficient. For instance, consider a quadratic model of the form,

$$f(x) = ax_1^2 + bx_1 + c$$

Then the higher order coefficient a can be penalized by adding it to the cost function to be minimized.

$$J = \min \sum (f(x) - y_{obs})^2 + \lambda a$$

where λ is a weighting function which gives the importance of the parameter regularization. In a model with multiple inputs, such as,

$$f(x) = a_{1n}x_1^n + a_{2n}x_2^n + a_{3n}x_3^2 + b_{1n}x_1^{n-1} + b_{2n}x_2^{n-1} + \dots + c$$

Then the cost function penalizing the highest order coefficient a_{jn} can be written as follows.

$$J = \min \sum (f(x) - y_{obs})^2 + \lambda \sum a_{jn}^2$$

The choice of λ is vital for the correct fitting of the model. For instance, choosing a very big value for λ may fail to eliminate overfitting and gradient descent algorithm may fail to converge. Additionally, it may cause underfitting, where the model will fail to fit training data well. This occurs because the cost function will understand that the parameter reduction task is of highest priority, while the data fitting is not so important. It may even cause the parameters to become almost equal to zero, in which case, the model becomes a flat line.

9.1. REGULARIZED LINEAR REGRESSION

In regularized linear regression, the fitting problem is solved by minimizing the following cost function:

$$J = \frac{1}{2m} \sum \left[\sum_{i=1}^m (f(x) - y_{obs})^2 + \lambda \sum_{j=1}^n a_j^2 \right]$$

Using gradient descent algorithm, the process of iteration consists in modifying the parameters according the following policy (with no regularization of the bias):

repeat:

$$\begin{aligned} a_0\text{new} &= a_0\text{old} - \alpha * \frac{1}{m} * \text{sum}((f(x) - y)^2) * x_0 \\ a_j\text{new} &= a_j\text{old} - \alpha * [\frac{1}{m} * \text{sum}((f(x) - y)^2) * x_j - \lambda/m \\ &\quad * a_j\text{old}] \end{aligned}$$

Rearranging and rewriting the second equation,

$$a_j\text{new} = a_j\text{old} (1 - \alpha * \lambda/m) - \alpha * \frac{1}{m} * \text{sum}((f(x) - y)^2) * x_j$$

since α , λ and m are always positive, it can be stated that,

$$1 - \alpha * \lambda/m < 1$$

which shows that the regularization works directly by reducing the value of the parameters.

Using Normal Equation approach regularization adds a $(n+1) \times (n+1)$ matrix of 0's (non-regularized) and 1's (regularized) in the diagonal to the equation.

$$a = (X^T X + \lambda \begin{bmatrix} 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ 0 & 0 & 0 & 1 \end{bmatrix})^{-1} X^T y$$

9.1.1. Exercises

- 1) When using a logistic regression model to predict certain data, what statements are true?
 - a) One can decrease the overfitting possibility by adding more features to the training set.
 - b) Regularization always improves model performance.
 - c) A new feature always result in equal or better model performance.
- 2) A logistic regression model is trained at two different circumstances, one using $\lambda = 10$ and the other using $\lambda = 0$. The results obtained for the fitted parameters are (not necessarily in the same order as it was presented the λ values) equal to:

Parameters in training 1:

$$\begin{bmatrix} 93.21 & 33.42 \end{bmatrix}$$

Parameters in training 2:

$$\begin{bmatrix} 2.31 & 0.79 \end{bmatrix}$$

Which training corresponds to the training using $\lambda = 10$?

- 3) Check the correct statement regarding regularization.
 - a) A large value of λ may cause underfitting.
 - b) Regularization cause the cost function to become non-convex, in which case the gradient descent algorithm may fail to converge.
 - c) Regularization is not very helpful for logistic regression since its output is already limited by $0 < f(x) < 1$.
 - d) A large value of λ do not reduce model performance, however it should be avoided because of numerical problems.

10

Introduction to Neural Networks

CONTENTS

10.1. The Essential Block: Neurons	174
10.2. How To Implement A Neuron Using Python And Numpy.....	176
10.3. Combining Neurons to Build a Neural Network	177
10.4. Example Of Feedforward Neural Network.....	178
10.5. How To Implement A Neural Network Using Python and Numpy.....	179
10.6. How To Train A Neural Network	183
10.7. Example Of Calculating Partial Derivatives	187
10.8. Implementation Of A Complete Neural Network With Training Method	189

This chapter provides a simple and concise explanation on how neural networks work and how to implement one from scratch using Python. The development of the concepts along this chapter does not assume any prior knowledge of machine learning or neural networks at all. The reader may be surprised to see that neural networks are actually not complicated as it may sound like (though it may grow in complexity).

10.1. THE ESSENTIAL BLOCK: NEURONS

A neural network is actually built on the assembling of multiple basic building blocks called neurons. The name of it comes from an analogy with the human neural system, which is also composed by specialized cells called neurons.

In biology, a neuron is composed by three essential parts: one that receives a (chemical/ electrical) signal from a source, a body which somehow processes such signal depending on its “magnitude,” and outputs the transformed signal at its other end.

Similarly, a “mathematical” neurons takes input(s) signal, transforms it (math operations) and produces a transformed signal, i.e., an output. The following figure shows how a neuron with 2 inputs and one output looks like.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \rightarrow (W) \rightarrow (\sum) \rightarrow (\text{Activation}) \rightarrow (y)$$

where x_1 and x_2 are the input signals, W is a weight matrix which performs the multiplication of each input by a weight.

$$x_1 \rightarrow x_1 \times w_1$$

$$x_2 \rightarrow x_2 \times w_2$$

After weighting, the signal is transmitted to the (\sum) , which performs the addition of all the signals plus a bias b .

$$(x_1 \times w_1 + x_2 \times w_2 + b)$$

The value produced by the addition performed above is then send to the activation function which produces the output by applying a (non) linear transformation of the signal.

$$y = f(x_1 \times w_1 + x_2 \times w_2 + b)$$

In classification problems (the first use of neural networks), the activation functions consists in a mathematical transformation that transforms the unbounded inputs into bounded one with a predictable form. One of the most popular functions used for such purpose is the sigmoid function as shown below, where z consists in any (transformed) signal given to the sigmoid function.

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

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
x = np.arange(-7, 8)
y = 1/(1+np.exp(-x))
with PdfPages('sigmoid.pdf') as pdf:
    plt.figure(figsize=(3, 3))
    plt.plot(x, y)
    plt.title('Sigmoid Function')
    pdf.savefig() # saves the current figure into a pdf page
    plt.show()
    plt.close()
```

Notice that the sigmoid function bounds the outputs (y) to values between the range $[0, 1]$. It can be interpreted as a compression, in such way that high values becomes 1 (100 % probability) and very negative values becomes 0 (0 % probability).

10.1.1. Example

Let the 2-input neuron shown above have the following weight and bias values.

$$W = [0, 1]$$

$$b = 2$$

where the values stored in the matrix W represents the weights $W = [w_1, w_2]$, so $w_1 = 0$ and $w_2 = 1$, but writing in vector form. Suppose such neuron receives an input signal of $X = [3, 4]$. Therefore, its processing can be represented in the following form.

$$WX^T + b = w_1 \times x_1 + w_2 \times x_2 + b$$

$$WX^T + b = 0 \times 3 + 1 \times 4 + 2$$

$$WX^T + b = 6$$

$$y = f(WX^T + b) = f(6) = 0.997$$

The above shows that, given the input vector $X = [3, 4]$, the neuron outputs a signal of 0.997.

The neural network built with neurons, which receives input and gives output(s) is also called a **feedforward** neural network, or FFNN.

10.2. HOW TO IMPLEMENT A NEURON USING PYTHON AND NUMPY

Bearing the concepts presented above, let's do a relatively simple implementation of a Neuron using Python and with help of the numpy library. First of all, import the numpy library and define the sigmoid function, which consists in our activation function.

```
import numpy as np
```

```
def sigmoid(z):
    return 1/(1+np.exp(-z))
```

Next, let's implement the Neuron as a Python class, thus using an object-oriented approach to construct this basic unit of the neural network. The Neuron class has an initialization function, which receives the initial weights and bias, and a process function, which produces the neuron output after assembling the inputs with the weights and bias with the sigmoid transformation.

```

class Neuron:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias

    def process(self, inputs):
        z = np.dot(inputs, self.weights,) + self.bias
        return sigmoid(z)

```

To check if the implementation above is correct, test it against the example given above, since the expected result is already known. Any error on typing the above code will produce either an error running the code or not the same result as expected.

```

weights = [0, 1]
bias = 2
# create a Neuron object called myneuron
myneuron = Neuron(weights, bias)
# Input vector X
X = [3, 4]
print(myneuron.process(X)) # 0.9975...

```

10.3. COMBINING NEURONS TO BUILD A NEURAL NETWORK

In its essence, a neural network is simple a bunch of neurons connected in parallel and/or series, as shown in the following figure.

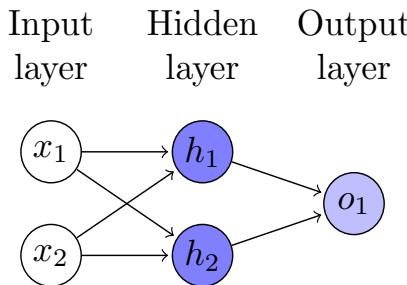


Figure 10.1: Simple neural network.

This network has 2 inputs (x_1 and x_2), two neurons h_1 and h_2 in a layer called hidden layer. A hidden layer is defined as any layer between an input layer and an output layer. There may be multiple hidden layers in a neural network. At the end of the above network there is an output neuron o_1 . Notice that the above structure is called a network because the inputs of the neuron o_1 are the outputs of the hidden layer neurons.

10.4. EXAMPLE OF FEEDFORWARD NEURAL NETWORK

Using the neural network shown above, let's evaluate it. Assume that all neurons have the same weight $W = [0, 1]$ and bias $b = -1$. Also, the activation function of every neuron is the sigmoid function. Let h_1 and h_2 be the output of the hidden neurons and o_1 the output of the network.

Consider that the input given is $X = [3, 4]$.

$$h_1 = h_2 = f(WX^T + b)$$

$$h_1 = h_2 = f(3 \times 0 + 4 \times 1 + (-1))$$

$$h_1 = h_2 = f(3) = 0.952$$

$$o_1 = f(W[h_1, h_2]^T + b)$$

$$o_1 = f(0.952 \times 0 + 0.952 \times 1 + (-1))$$

$$o_1 = f(-0.048) = 0.488$$

The above calculation shows that, for a given input of $X = [3, 4]$, the above neural network output the value 0.488. It is important to notice that a neural network is not restricted to the architecture shown above. Rather, it can theoretically have any number of inputs, hidden layers and output neurons. Still, the basic idea is always the same, feed the input to the network and receive the outputs at the other end of it.

10.5. HOW TO IMPLEMENT A NEURAL NETWORK USING PYTHON AND NUMPY

To write the code that simulated the neural network shown in Figure 10.1, we will use the same type of approach already presented, i.e., object-oriented one. For that we will write a class Neural Network that will be constructed initially by fixed weights and bias, and the two hidden layer neurons as well as the output neuron. A process function will perform the internal calculations of the neural network as demonstrated already and returns the output. The code implementation is shown below.

class NeuralNetwork:

“”

*A 2-input neural network with 2 neurons in hidden layer
and 1 output neuron. All neurons have the same weight and bias.*

w = [0, 1]

bias = -1

“”

def __init__(self):

weights = [0, 1]

bias = -1

Add the neurons when constructing the object

self.H = [] # Store the hidden neurons in a list

self.H.append(Neuron(weights, bias)) # h1

self.H.append(Neuron(weights, bias)) # h2

self.O = [] # Store the output neurons in a list

self.O.append(Neuron(weights, bias)) # o1

def process(self, x):

out_h = [] # list with the outputs of the hidden layer

for h **in** self.H: *# iterate over the neurons in the hidden layer*

out_h.append(h.process(x))

```
out = [] # list with the outputs of the hidden layer
for o in self.O: # iterate over the neurons in the output layer
    out.append(o.process(out_h))

return out
```

Now we can test the implementation above using the input already evaluated

$X = [3, 4]$. expected result is 0.488.

```
mynetwork = NeuralNetwork()
```

```
X = [3, 4]
```

```
print(mynetwork.process(X)) # [0.4881...]
```

As expected, the neural network output is approximately 0.488. Optionally, instead of hard coding the neurons inside the structure of the class NeuralNetwork, which would make it strictly build with two neurons in the hidden layer and one neuron in the output layer, a method add() can be implemented to directly define a layer with a certain number of neurons. Additionally the number of inputs it can receive may vary so this generalization should also be added. The following code shows this improvement as an optional implementation to have a more generic neural network class.

```
class NeuralNetworkv2:
```

```
    """
```

An improved neural network class.

Layers are added using the add() method.

Initially, all neurons have the same weight and bias.

```
w = [0, 1, 1,...]
```

```
bias = -1
```

```
"""
```

```
def __init__(self):
```

```
    self.Layers = [] # empty list to contain the layers
```

```

def add(self, dim, input_shape = 0):
    """
        add(self, input_shape) adds a layer to the neural network structure.
        For the first layer, the input_shape must be given. Subsequent layers are
        configured to accept inputs coming from the previous layers.
    """

    bias = -1 # bias (equal to all neurons)
    if dim < 1:
        print('The layer should hold at least 1 neuron')
        return 0
    if len(self.Layers) == 0: # check if this is the first layer
        if input_shape < 1:
            print('The first layer must contain at least 1 input.')
            return 0
        else:
            self.Layers.append([]) # add an empty list to hold the neurons on
            the new layer
            weights = np.ones((input_shape,))
            weights[0] = 0
            for i in range(dim):
                self.Layers[-1].append(Neuron(weights, bias))
    else: # if this is not the first layer
        weights = np.ones((len(self.Layers[-1]),))
        weights[0] = 0
        self.Layers.append([])
        for i in range(dim):
            self.Layers[-1].append(Neuron(weights, bias))

def predict(self, x): # equivalent to the old process function with a better
naming
    # create an empty list to store the outputs of the network
    output = []

```

```
for x_data in x: # iterate at each record
    in_x = x_data # set x as the initial input to the network
    for layer in self.Layers: # iterate over the layers
        out_h = []
        for neuron in layer: # iterate over the neurons of the layer
            out_h.append(neuron.process(in_x))
        in_x = out_h # set the input of the next layer as the output of the
previous one.
    output.append(out_h)
return output
```

```
def __str__(self): # to show when using print()
    num_layers = len(self.Layers)
    summary = 'Neural Network Summary\n\n'
    for i, layer in enumerate(self.Layers):
        summary += f'layer {i}: {len(layer)} neurons\n'
    return summary
```

```
mynetwork = NeuralNetworkv2()
# add 2 neurons in the hidden layer
mynetwork.add(2, input_shape = 2)
# add 1 output neuron
mynetwork.add(1)
```

```
# print network summary
print(mynetwork)
```

```
X = [[3, 4]]
print(mynetwork.predict(X)) # [0.4881...]
# Testing for multiple inputs...
```

```
X = [[3, 4], [3, 4], [3, 4]]
print(mynetwork.predict(X)) # [[0.4881...], [0.4881...], [0.4881...]]
```

10.6. HOW TO TRAIN A NEURAL NETWORK

Consider the following dataset, which shows the height, weight and gender of different individuals.

Table 10.1: Dataset of (Hypothetical) Weight, Height, and Gender

Name	Weight (kg)	Height (cm)	Gender
Mary	54	156	F
John	70	173	M
Carl	85	184	M
Sara	60	160	F

where M indicates Male and F indicates Female individuals.

The task consists in train to predict the gender of a person given its weight and height. Therefore, there are two inputs and one output (gender), which resembles the architectures that were already presented previously.

Let P be the probability of an individual being female. So an individual which is known to be female is 1 (100%) and an individual which is known to be male is 0 (0%). To obtain the height and weight as values in the domain [0, 1], perform normalization of it. To do so, we need the minimum and maximum values.

- minimum weight = 54 / maximum weight = 85
- minimum height = 156 / maximum height = 184

Normalization can be done using the following equation.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x' is the normalized value, x is the regular value, $\min(x)$ is the minimum value and $\max(x)$ is the maximum value. Doing the normalization and applying the probability to the gender column, one obtains the following table.

Name	Weight (kg)	Height (cm)	Gender
Mary	0	0	1
John	0.5161	0.607	0
Carl	1	1	0
Sara	0.193	0	1

The neural network training only works if one is able to somehow measure or quantify how “accurate” it is so one can try to make it better. The way to quantify it is by using the loss metric.

One of the possible loss metrics is the MSE (mean squared error).

$$MSE = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2$$

where n is the number of data points, y is the observed output value (probability) and \hat{y} is the predicted output. In the present case, these values are: + n is 4 (Mary, John, Carl, Sara) + y are the known Gender values (1, 0, 0, 1) where 1 is Female.

The difference $(y - \hat{y})^2$ is called the squared error. They should be squared to make all of them positive, so there is no error cancellation when summing them up. And we divide the total sum by the number of samples, thus obtaining a mean squared error. The lower is these values, the better is the model. Ideally a perfect model would have $MSE = 0$, showing that it predicts exactly as it should, without any errors.

Therefore, the whole idea of training a neural network is to try to minimize the loss.

As an example, consider that a neural network outputs only 0 for all the data records. What is the value of MSE in such case?

Name	Weight (kg)	Height (cm)	y	\hat{y}
Mary	0	0	1	0
John	0.5161	0.607	0	0
Carl	1	1	0	0
Sara	0.193	0	1	0

$$MSE = \frac{1}{4}[(1-0)^2 + (0-0)^2 + (0-0)^2 + (1-0)^2] = 0.5$$

The following code shows the implementation of the MSE loss function using Python and Numpy. Notice that this function receives both the observed and predicted outputs in order to calculate the loss.

```
def loss(y_obs, y_pred):
    """
    loss(y_obs, y_pred) calculates the MSE between observed (y_obs)
    and predicted (y_pred) values.
    """
    return np.mean((y_obs - y_pred)**2)
```

Test the loss function

```
y_obs = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])

print(loss(y_obs, y_pred)) # 0.5
```

With the goal clearly stated (minimize the loss), the next step consists in determining the algorithm, which will calculate the optimum neural network parameters (weights and bias) to obtain a good accuracy of the model when presented to a dataset.

To do so, one had to know how changing the weights and bias affects the loss, so it can be updated in the direction that it is minimized. The effect of changing a weight or bias can be found through partial derivatives, since the loss is actually a function not only of the inputs but also of the weights and bias. Considering the inputs as constants (they do not change during the calibration of the weights),

$$MSE = MSE(W, b)$$

To obtain the partial derivative of MSE with respect to one weight, say w_1 , it is necessary to make use of the chain rule so as to break down this derivative into two parts.

$$\frac{\partial \text{MSE}}{\partial w_1} = \frac{\partial \text{MSE}}{\partial y_{\text{pred}}} \frac{\partial y_{\text{pred}}}{\partial w_1}$$

where y represents model output, and the partial derivative means how such value changes as the weight changes. For the first part of the equation, one obtains.

= =

To obtain $\frac{\partial y_{\text{pred}}}{\partial w_1}$, apply the chain rule knowing that the weight w_1 only affects h_1 thus,

$$\frac{\partial y_{\text{pred}}}{\partial w_1} = \frac{\partial y_{\text{pred}}}{\partial h_1} \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{\text{pred}}}{\partial h_1} = w_5 \times f'(w_5 h_1 + w_6 h_2 + b_3)$$

The above comes from the fact that,

$$y_{\text{pred}} = f(w_5 h_1 + w_6 h_2 + b_3)$$

The second derivative is calculated as,

$$\frac{\partial h_1}{\partial w_1} = x_1 \times f'(w_1 x_1 + w_2 x_2 + b_1)$$

In the above equations, $f'()$ is the derivative of the sigmoid function which is defined as,

$$f'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = f(z) \times (1 - f(z))$$

With those equations, one can assemble the parts back together and obtain the derivatives of the loss function with respect to each weight and bias in a similar fashion. The calculation of derivatives backwards is known as backpropagation, or backprop.

10.7. EXAMPLE OF CALCULATING PARTIAL DERIVATIVES

To show how the gradient of the loss can be analytically obtained, consider one of the records extracted from the height and weight/gender table, as shown below.

Name	Weight (kg)	Height (cm)	y	\hat{y}
John	0.5161	0.607	0	0

Consider that the weights of the neural network are initialized to 1 and biases to 0. Performing a feedforward pass through the network, one obtains the following output.

$$h_1 = f(w_1x_1 + w_2x_2 + b_1) = f(0.5161 + 0.6907 + 0) = 0.77$$

$$h_2 = f(w_3x_1 + w_4x_2 + b_2) = 0.77$$

$$o_1 = f(w_5h_1 + w_6h_2 + b_3) = f(0.77 + 0.77) = 0.823$$

It seems that the neural network outputs $y_{pred} = 0.823$ which indicates high favor towards female gender (with 82.3% of confidence), while the subject is actually a male. Calculate the derivative of the loss function with respect to w_1 as follows.

$$\frac{\partial MSE}{\partial w_1} = \frac{\partial MSE}{\partial y_{pred}} \frac{\partial y_{pred}}{\partial h_1} \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial MSE}{\partial y_{pred}} = -2(y_{obs} - y_{pred}) = -2(0 - 0.823) = 1.646$$

$$\frac{\partial y_{pred}}{\partial h_1} = w_5 \times f'(w_5 \times h_1 + w_6h_2 + b_3) = f'(0.77 + 0.77) = f(1.54) \times (1 - f(1.54)) = 0.1453$$

$$\frac{\partial h_1}{\partial w_1} = x_1 \times f'(w_1x_1 + w_2x_2 + b_1) = 0.5161 \times f'(0.5161 + 0.6907) = 0.09147$$

Assembling all the partial derivatives to obtain the gradient of the loss with respect to the weight w_1 .

$$\frac{\partial MSE}{\partial w_1} = 1.646 \times 0.1453 \times 0.09147 = 0.02187$$

The above shows that, if w_1 increases, the loss value also increases, thus getting worst. Therefore, it shows an indication that w_1 should decrease, so as to reduce the error. Let's prove that by using smaller value of w_1 , say -1.

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1) = f(-0.5161 + 0.6907 + 0) = 0.54$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.77$$

$$o_1 = f(w_5 h_1 + w_6 h_2 + b_3) = f(0.75 + 0.77) = 0.79$$

Notice how the probability output at o_1 has reduced (from 82% to 79%), which proves that reduction of the weight also reduced the loss. However, it is more interesting to implement an algorithm that systematically updated the weights in a way to find the minimum loss, rather than trial and error.

One iterative method that may be implemented is the Stochastic Gradient Descent (SGD) method, which consists in an algorithm which iteratively updates the neural network parameters and it converges to an optimum point (as long as it is correctly configured). This algorithm makes use of the following update equation.

$$w < -w - \eta \frac{\partial MSE}{\partial w}$$

where $\eta > 0$ is a learning rate which can control the size of the modification of the parameter. From the above equation it can be seen that: + If the derivative is positive, then the parameter decreases; + If the derivative is negative, then the parameter increases;

The above equation is applied at each iteration for all the weights and bias of the neural network. It should reduce the loss at each iteration (monotonically or not, i.e., oscillating) until it reaches a minimum, improving the network prediction capability.

The SGD algorithm is Stochastic since the optimization will be done by selecting one sample at a time, randomly. The complete algorithm for optimization consists in:

- Selecting one sample from the dataset at a time;

- Calculate all the partial derivatives of the neural network with respect to the loss function;
- Update the parameters using the update equation;
- Go back to step 1.

10.8. IMPLEMENTATION OF A COMPLETE NEURAL NETWORK WITH TRAINING METHOD

To test the algorithm, consider the weight and height/ gender table already mentioned in this chapter. We use a neural network with 2 neurons in hidden layer to predict the gender based on both the weight and height.

The complete code for the class TrainableNet is shown below. It consists in an adaptation of the code presented by Zhou, 2019 at <https://victorzhou.com/blog/intro-to-neural-networks/>

```
def deriv_sigmoid(x):
    # Derivative of sigmoid: f'(x) = f(x) * (1 - f(x))
    fx = sigmoid(x)
    return fx * (1 - fx)
```

```
class TrainableNet:
```

```
    ""
```

TrainableNet is a neural network class limited to:

- 2 inputs
- 1 hidden layer with 2 neurons ($h1, h2$)
- 1 output layer with 1 neuron ($o1$)

```
    ""
```

```
def __init__(self):
    # Weights
    self.w = np.ones((6,))
    # Biases
    self.b = np.ones((3,))
```

```
def predict(self, x):
    # x is a numpy array with 2 elements.
    h1 = sigmoid(self.w[0] * x[0] + self.w[1] * x[1] + self.b[0])
    h2 = sigmoid(self.w[2] * x[0] + self.w[3] * x[1] + self.b[1])
    o1 = sigmoid(self.w[4] * h1 + self.w[5] * h2 + self.b[2])
    return o1

def train(self, X, y_true_all):
    """
    - data is a (n x 2) numpy array, n = # of samples in the dataset.
    - all_y_trues is a numpy array with n elements.
    Elements in all_y_trues correspond to those in data.
    """

    learn_rate = 0.2
    epochs = 1000 # number of iterations used

    for epoch in range(epochs):
        for x, y_true in zip(X, y_true_all):
            # --- Calculate acutal outputs
            sum_h1 = self.w[0] * x[0] + self.w[1] * x[1] + self.b[0]
            h1 = sigmoid(sum_h1)

            sum_h2 = self.w[2] * x[0] + self.w[3] * x[1] + self.b[1]
            h2 = sigmoid(sum_h2)

            sum_o1 = self.w[4] * h1 + self.w[5] * h2 + self.b[2]
            o1 = sigmoid(sum_o1)
            y_pred = o1

            # --- Calculate partial derivatives.
```

```

# --- Naming: d_L_d_w1 represents "partial L / partial w1"
dL_dyPred = -2 * (y_true - y_pred)

# Neuron o1
dyPred_dw5 = h1 * deriv_sigmoid(sum_o1)
dyPred_dw6 = h2 * deriv_sigmoid(sum_o1)
dyPred_db3 = deriv_sigmoid(sum_o1)

dyPred_dh1 = self.w[4] * deriv_sigmoid(sum_o1)
dyPred_dh2 = self.w[5] * deriv_sigmoid(sum_o1)

# Neuron h1
dh1_dw1 = x[0] * deriv_sigmoid(sum_h1)
dh1_dw2 = x[1] * deriv_sigmoid(sum_h1)
dh1_db1 = deriv_sigmoid(sum_h1)

# Neuron h2
dh2_dw3 = x[0] * deriv_sigmoid(sum_h2)
dh2_dw4 = x[1] * deriv_sigmoid(sum_h2)
dh2_db2 = deriv_sigmoid(sum_h2)

# --- Update weights and biases
# Neuron h1
self.w[0] -= learn_rate * dL_dyPred * dyPred_dh1 * dh1_dw1
self.w[1] -= learn_rate * dL_dyPred * dyPred_dh1 * dh1_dw2
self.b[0] -= learn_rate * dL_dyPred * dyPred_dh1 * dh1_db1

# Neuron h2
self.w[2] -= learn_rate * dL_dyPred * dyPred_dh2 * dh2_dw3
self.w[3] -= learn_rate * dL_dyPred * dyPred_dh2 * dh2_dw4
self.b[1] -= learn_rate * dL_dyPred * dyPred_dh2 * dh2_db2

```

```
# Neuron o1
self.w[4] -= learn_rate * dL_dyPred * dyPred_dw5
self.w[5] -= learn_rate * dL_dyPred * dyPred_dw6
self.b[2] -= learn_rate * dL_dyPred * dyPred_db3

# --- Calculate total loss at the end of each epoch
if epoch % 10 == 0:
    y_preds = np.apply_along_axis(self.predict, 1, X)
    loss = loss(y_true_all, y_preds)
    print("Epoch %d loss: %.3f" % (epoch, loss))
```

In the next step, we build a neural network and test its initial predictions in the dataset provided. Then this network is trained and retested to see how its performance improved.

```
# Define dataset
```

```
X = np.array([[0, 0],
[0.5161, 0.607],
[1, 1 ],
[0.193, 0]])
```

```
y_true_all = np.array([
1,
0,
0,
1,
])
```

```
# Test the neural network
```

```
mynetwork = TrainableNet()
print([mynetwork.predict(x) for x in X]) # [0.92, 0.94, 0.95, 0.93]
```

Using the current set of parameters, the neural network predictions are indicating that all the records are highly probable (with more than 90%) females, though it is known that there is half of females and half of males

in the dataset. The next code is used to train the network and check the prediction again.

Train the neural network

```
mynetwork.train(X, y_true_all)  
print([mynetwork.predict(x) for x in X]) # [0.96, 0.06, 0.013, 0.94]
```

Notice how the network performance improved. It predicted with more than 90% of confidence the two records which are real females and the two which are males.

In this chapter, the following concepts were presented:

- what is a neuron and how it works;
- the architecture of neural networks;
- sigmoid function;
- loss function, specifically the Mean Squared Error (MSE);
- How training works and that training is loss minimization;
- Use of backpropagation to calculate partial derivatives;
- How to use stochastic gradient descent (SGD) method to train a neural network.

11

Introduction to Decision Trees and Random Forest

CONTENTS

11.1. Decision Trees	196
11.2. Random Forest.....	205
11.3. Programming Exercise – Decision Tree From Scratch With Python.....	207

In this chapter, the concept of decision trees is presented assuming no prior knowledge of machine learning and the models available. From the decision trees, it is derived the concept of random forest and how they work with simple examples.

11.1. DECISION TREES

A tree can be seen essentially as a main branch with smaller branches connected to it. It has some essential parts as well, such as its roots and its treetop. Decision trees are similar to the physical trees, in the sense they are composed by branches (decisions) and main parts (nodes).

To better clarify, consider the following plotted data, where circles represent one label and crosses are another label.

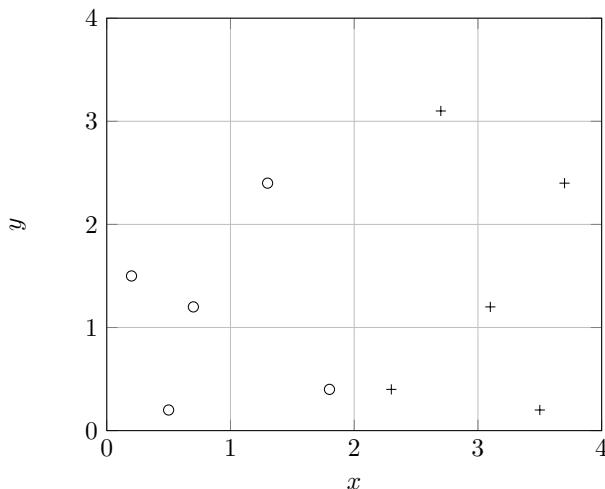


Figure 11.1: Dataset example.

Consider that one should predict was it the label for a new point to be added on Figure 11.1, which falls in $x = 1$, naturally it would be labeled as circles. What was done to do such labeling is a natural decision process of visualizing the plot and noticing that, since all the circles are contained in the region where $x < 2$ and that all the crosses are in the region $x > 2$, then it is natural that a point falling in $x = 1$ would fall in the first category.

Decision trees work in a similar fashion, by performing decisions or applying certain conditions to data and when such conditions are satisfied, then the data can be labeled.

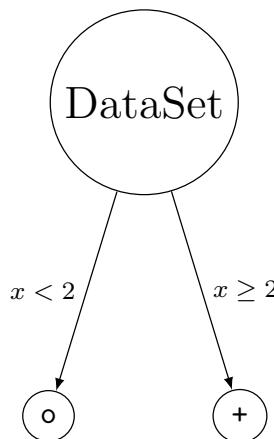


Figure 11.2: Decision tree for dataset in Figure 11.1.

The structure above is one of the simplest decision trees possible, with a single decision node which evaluates $x < 2$. If the test passes, then data is assigned to left branch. Otherwise, it is given to the right branch.

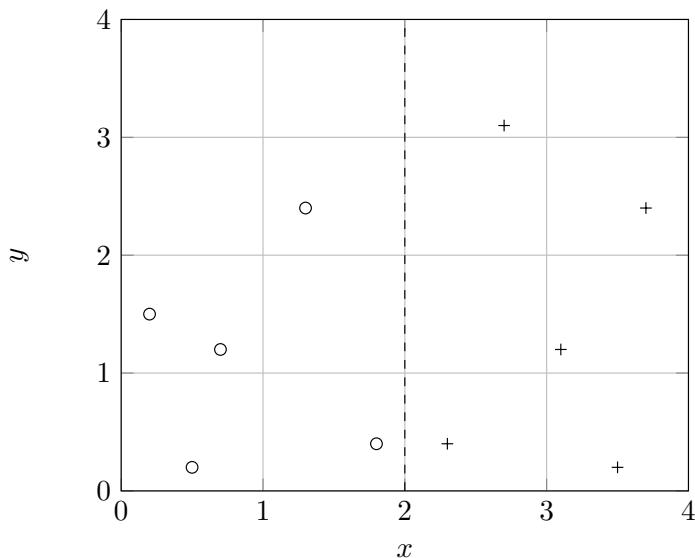


Figure 11.3: Decision Tree Boundary at $x = 2$.

Therefore, the task given to a decision tree consists in, given a sample data, with unknown label, how to classify new samples. Consider now another example, where data contains three labels.

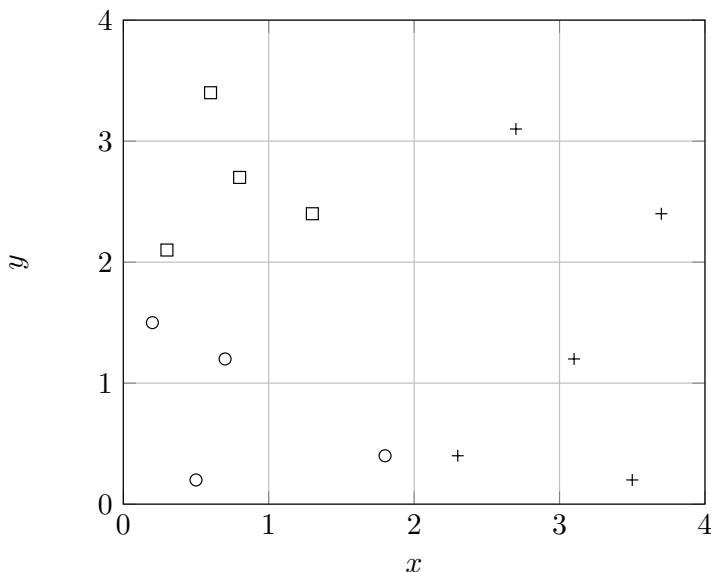


Figure 11.4: Dataset Example with three labels.

Notice that the addition of a new class makes the use of the decision tree built previously not applicable, since it won't recognize the third class.

Rather, given a point (x, y) :

- It checks if $x < 2$, then classify as circle, but it could be circle or square;
- It checks if $x \geq 2$, then classify as cross.

This shows that the decision tree needs to grow in order to incorporate the possibility of classifying the new label. One possible solution is the one shown in the following Figure.

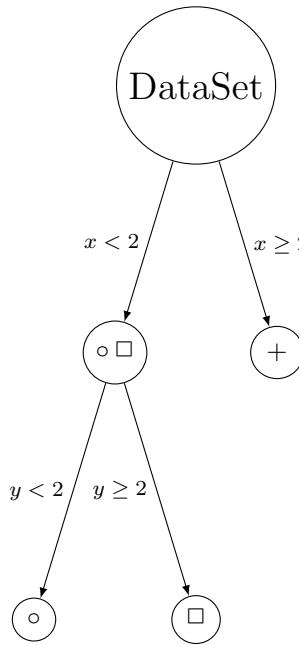


Figure 11.5: Decision tree with three labels.

In this sense, it becomes simple to understand the concept behind decision trees. Nodes hold the labeled data while the branches make the decision or act like gates, where data which satisfies the condition flows through the branch and data which does not satisfy is not passed to the following node.

11.1.1. Decision Tree Training

To understand how the training of decision trees work, it is important to understand that it is necessary to have a criteria which defines how “good” a certain decision (choice of feature and threshold) is. One possible criteria is the Gini impurity.

11.1.2. The Gini Impurity

Consider the 2-class labeled data presented in Figure 11.1. One picks a random point from this dataset and wants to classify it as circle or cross. Since there is a total of 10 data points and 5 points of each, randomly attributing labels means one would classify a point 5/10 times as circle and 5/10 as cross. Gini Impurity can measure the probability of classifying a data point incorrectly.

To understand it, first create a table to measure the probability of classifying a point correctly or incorrectly (Table 11.1).

Table 11.1: Probability in Binary Classification

Event	Probability
Pick circle, Classify circle	25%
Pick circle, Classify cross	25%
Pick cross, Classify circle	25%
Pick cross, Classify cross	25%

One can notice that, from the fours events mentioned above, two of them consists in a wrong classification, summing to 50% of probability of misclassifying a datapoint. Therefore, Gini Impurity is 0.5.

Consider there are C total classes and $P(i)$ is the probability of selecting one datapoint with class i . Gini Impurity formula then consists in,

$$G = \sum_{i=1}^C p(i) * (1 - p(i))$$

In the example above mentioned, there are two classes ($C = 2$), and the probability of picking a cross is the same as picking a circle thus $p(i) = 0.5$. The Gini Impurity is therefore,

$$G = p(1) \times (1 - p(1)) + p(2) \times (1 - p(2))$$

$$G = 0.5 \times (0.5) + 0.5 \times (0.5) = 0.5$$

Consider now the perfect split of data points as shown in Figure 12.3, i.e., the decision boundary is placed at $x = 2$, so everything to the left is one class (circle) and everything to the right is another class (square). On the left side, Gini Impurity is,

$$G = 1 \times (1 - 1) + 0 \times (1 - 0) = 0$$

Similarly, the Gini Impurity on the right side is,

$$G = 1 \times (1 - 1) + 0 \times (1 - 0) = 0$$

In both cases, Gini Impurity is exactly 0, which indicates a perfect split. In summary, the dataset which had initially 0.5 impurity was splitted into 2 branches with 0 impurity.

The above example illustrates that, when the data is perfectly split, Gini impurity is 0 (zero), the lowest and best possible value. This can only be achieved when every data point is correctly labeled at every class.

Consider now an example of inaccurate data classification. Suppose the threshold at x is slightly displaced to the left side, thus leaving one data point which is supposed to be a circle together with the crosses. In such case, at the left side there is still only circle points, thus $G_l = 0$ where the index l indicates left side. For the right (r) side,

$$G_r = \frac{1}{6} \times \left(1 - \frac{1}{6}\right) + \frac{5}{6} \times \left(1 - \frac{5}{6}\right) = 0.278$$

The splitting quality is measured by weighting Gini Impurity of each side (branch) by the quantity of points in it. In the above example, the left branch contains 4 points while right contains 6. Thus,

$$G = 0.4 \times 0 + 0.6 \times 0.278 = 0.167$$

The total amount of impurity removed can be calculated by the difference between the original impurity and the new one.

$$0.5 - 0.167 = 0.33$$

The value above, referred to as Gini Gain, defines the best split to be chosen. In summary, the higher the Gini Gain, better is the split considered. For instance, it can be noticed that the Gini Gain for the perfect split is 0.5 which is higher than 0.33, thus showing that the perfect split is better than this inaccurate one.

11.1.3. The Root Node

Consider the 3-class labeled data presented in Figure 11.4.

The first step of training is to configure the decision of the root node in the tree. It is necessary to define which feature (x or y) needs to be evaluated and what is the threshold value. In the previous example, the root node used feature x with a threshold in value 2.

Intuitively, it can be seen that a decision node which performs good decisions is the one able to separate the classes as much as possible. In such case, the initial decision of doing a separation on $x = 2$ is considered to be the best one since it isolates one class perfectly. In the right side there is only crosses, and there is no crosses on the left side.

However, for the sake of understanding how training works, consider that the best decision is not known and it is necessary to find it mathematically. One possible way (through not the best or most efficient one) is to check every possible split and pick up the best one, i.e., the one that produces the highest Gina Gain (which is the one with the lowest Impurity).

To test every split means:

- To evaluate all the features (x and y in this case);
- All thresholds that produce different splits, i.e., after each coordinate of feature containing a point.

Figure 11.6 shows all the possible splits done in the feature x for the 3-class labeled data presented in Figure 11.4.

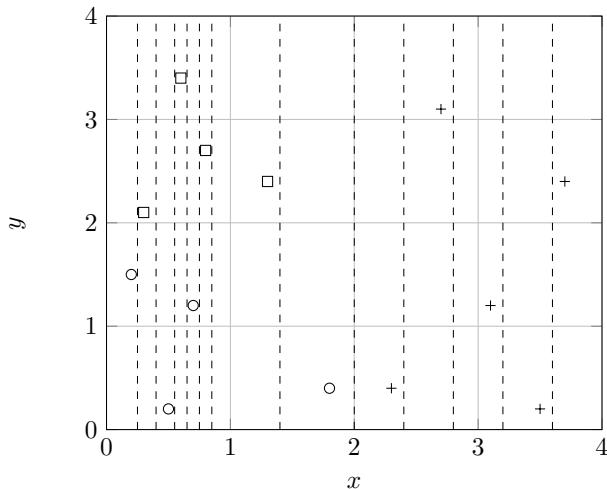


Figure 11.6: All possible splits in x feature.

A total of two splits are done, considering all possible thresholds that produces unique splits. As an example consider the split done at $x = 0.55$ (the third split from the left to right) and obtain the Gini impurity for that split. In such case, the left side (branch) contains 2 circles and 1 square, while the right branch contains 3 squares, 2 circles and all the 5 crosses.

Table 11.2: Example Splitting

Decision	Left Branch	Right Branch
$x = 0.55$	2 circles; 1 square	3 squares, 2 circles, 5 crosses

First obtain the Gini Impurity for the initial arrangement, i.e., with no split.

$$G_{init} = \sum p(i)(1-p(i)) = \frac{4}{13} \frac{9}{13} + \frac{4}{13} \frac{9}{13} + \frac{5}{13} \frac{8}{13}$$

$$G_{init} = 0.663$$

Now obtain the Gini Impurity for the split shown above.

$$G_l^{x=0.55} = 2/3 * 1/3 + 1/3 * 2/3 = 0.44$$

$$G_r^{x=0.55} = 3/10 * 7/10 + 2/10 * 8/10 + 5/10 * 5/10 = 0.62$$

$$G^{x=0.55} = \frac{3}{13} G_l^{x=0.55} + \frac{10}{13} G_r^{x=0.55} = 0.58$$

Naturally, any split of the data reduces the Gini Impurity, as it occurred above. The Gini Gain is:

$$\Delta G = 0.663 - 0.48 = 0.183$$

Therefore, the Gini Gain with the condition set on $x = 0.55$ is 0.183. To actually evaluate which is the best decision, the same evaluation is performed over all the other splits and the one with the lowest Gini Impurity is assumed as the best decision possible. The following table summarizes such possibilities evaluated at both features x and y .

Table 11.3: Summary of All Possible Splits and Their Gini Impurity

Decision	Left (circles, squares, crosses)	Right (circles, squares, crosses)	G_l	G_r	Gin Impurity	
$x = 0.25$	(1, 0, 0)	(3, 4, 5)	0.000	0.652	0.602	
$x = 0.40$	(1, 1, 0)	(3, 3, 5)	0.500	0.645	0.622	
$x = 0.55$	(2, 1, 0)	(2, 3, 5)	0.440	0.620	0.579	
$x = 0.65$	(2, 2, 0)	(2, 2, 5)	0.500	0.592	0.564	
$x = 0.75$	(3, 2, 0)	(1, 2, 5)	0.480	0.531	0.511	

$x = 0.85$	(3, 3, 0)	(1, 1, 5)	0.500	0.449	0.472	
$x = 1.40$	(3, 4, 0)	(1, 0, 5)	0.489	0.278	0.392	
$x = 2.00$	(4, 4, 0)	(0, 0, 5)	0.500	0.000	0.308	
$x = 2.40$	(4, 4, 1)	(0, 0, 4)	0.592	0.000	0.410	
$x = 2.80$	(4, 4, 2)	(0, 0, 3)	0.640	0.000	0.492	
$x = 3.20$	(4, 4, 3)	(0, 0, 2)	0.661	0.000	0.559	
$x = 3.60$	(4, 4, 4)	(0, 0, 1)	0.667	0.000	0.615	
$y = 0.30$	(1, 0, 1)	(3, 4, 4)	0.500	0.661	0.634	
$y = 0.50$	(2, 0, 2)	(2, 4, 3)	0.500	0.642	0.598	
$y = 1.30$	(3, 0, 3)	(1, 4, 2)	0.500	0.571	0.538	
$y = 1.60$	(4, 0, 3)	(0, 4, 2)	0.489	0.444	0.469	
$y = 2.20$	(4, 1, 3)	(0, 3, 2)	0.594	0.480	0.550	
$y = 2.50$	(4, 2, 4)	(0, 2, 1)	0.640	0.444	0.595	
$y = 2.80$	(4, 3, 4)	(0, 1, 1)	0.661	0.500	0.636	
$y = 3.20$	(4, 3, 5)	(0, 1, 0)	0.653	0.000	0.602	

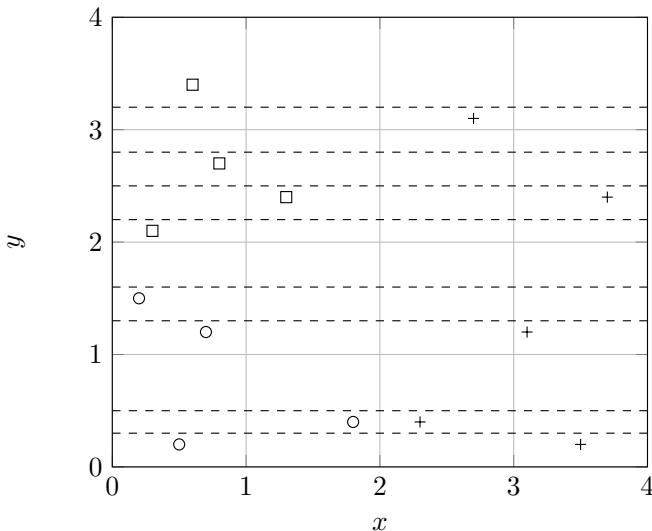


Figure 11.7: All possible splits in the y feature.

Notice that, as already expected, the lowest Gin is at $x = 2$ ($\text{Gin} = 0.308$). Another observation is that, starting at this point and for higher values of x , the Gin Impurity of right branch are all zero. That is because there is only one label retained at such branch (crosses) thus showing no mixture of classes.

Still there is mixed data. So one way of dealing with such is to add another decision node at the tree, at the branch consists of mixed data. By doing the same operation done above, one can find the best feature and condition to be used in the new node at the tree.

Adding a node on the right branch would be possible but it does not make sense since this branch already have only one label as data. A stop mode such as the left one is called a leaf node and no further branches can be added to it since it would separate data with the same label.

Once evaluated each branch and leaf branch of the decision tree, the training is done.

11.2. RANDOM FOREST

A random forest can be considered a collection of decision trees operating together. Though this concept is correct, it simplifies a lot what really is a

random forest. Therefore, for a better understanding consider the process of **bagging**.

Bagging, also called Bootstrap Aggregating can be used to train some decision trees with a set of n datapoints. The algorithm to perform such training consists in:

- Sample, with replacement, n samples from the data (bootstrap sample);
- Train a decision tree on the selected n samples;
- Repeat the above t times.

A prediction done using the t decision trees is done through aggregation of the results of each individual tree using one of the following methods:

- Take the majority vote, i.e., the most repeated label for classification problem;
- Take the average output, for regression problems.

Table 11.4: Example of Bagging (Cross Repeats So It Is Considered the Major Vote)

Decision Tree 1	Decision Tree 2	Decision Tree 3	Bagging Result
Cross	Cross	Circle	Cross

For a bagging to be considered a random forest, it requires a second parameter (besides the number of trees t), which controls how many features to try when finding the best split. This may not make much sense for the example shown because there were only two features, but most real-world problems have much more than this, even hundreds or thousands.

For instance, let a dataset with p features. Instead of evaluating all of them when adjusting a decision tree, as it was done in Table 11.3, a subset of

p is selected, in most cases with size \sqrt{p} or $\frac{p}{3}$. In this way, randomness is added to the problem, avoiding correlation between trees, which tends to improve forecast performance. The selection of features is also called *Feature Bagging*.

11.3. PROGRAMMING EXERCISE – DECISION TREE FROM SCRATCH WITH PYTHON

In this section, a decision tree algorithm is implemented from scratch (without using already implemented versions such as Sklearn). The algorithm is evaluated using the Iris Dataset from Sklearn package. This code is a modification/ adaptation of the one developed by jarvvis (<https://www.kaggle.com/jarvvis>).

11.3.1. The Iris Dataset

The data consists of three iris species with 50 samples each (150 samples total). The data was first presented by R.A. Fisher's classic 1936 paper, "The Use of Multiple Measurements in Taxonomic Problems." Alternatively, it can be found on the UCI Machine Learning Repository.

Besides the iris species (labels) the dataset includes properties of each flower. What makes this dataset interesting for academic purposes is that, while one dataset is linearly separable from the other, the third one cannot be linearly separated.

The flower properties present in the dataset as columns are:

- Id
- Sepal length (cm)
- Sepal width (cm)
- Petal length (cm)
- Petal width (cm)
- Species

11.3.2. The Algorithm

The first step consists in loading the necessary libraries. It will be used:

- pandas – for data processing;
- numpy – efficient numerical computation; and
- sklearn.datasets – to obtain the dataset.

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
import seaborn as sns
```

```
# load the data
```

```
dataset = load_iris()
```

Retrieve the labels y and features X (to denote a matrix).

```
X = dataset.data
```

```
y = dataset.target
```

#concatenate features matrix and labels into an array to be transformed into a dataframe

```
data = np.c_[X,y]
```

create column names

```
cols=['Sepal length', 'Sepal width', 'Petal length', 'Petal Width', 'Species']
```

create the dataframe for data processing

```
iris_df = pd.DataFrame(data = data, columns = cols)
```

```
iris_df.head(5)
```

S.No.	Sepal length	Sepal width	Petal length	Petal Width	Species
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0

for better understanding, change the labels from numeric to strings representing real names.

```
translation_dict = {0.0: 'iris sesota', 1.0: 'iris versicolor', 2.0: 'iris virginica'}
```

```
for label in translation_dict:
```

```
    iris_df['Species'] = iris_df['Species'].replace(label, translation_dict[label])
```

```
iris_df.head()
```

S.No.	Sepal length	Sepal width	Petal length	Petal Width	Species
0	5.1	3.5	1.4	0.2	iris sesota
1	4.9	3.0	1.4	0.2	iris sesota

2	4.7	3.2	1.3	0.2	iris sesota
3	4.6	3.1	1.5	0.2	iris sesota
4	5.0	3.6	1.4	0.2	iris sesota

11.3.3. Analyzing the Data

Use the command ‘.shape’ to verify the number of features ($n - 1$) and the number of samples of the dataset,

```
shape = iris_df.shape
```

```
print('Number of samples = ', shape[0])
```

```
print('Number of features = ', shape[1] - 1)
```

```
Number of samples = 150
```

```
Number of features = 4
```

11.3.4. Correlation Among Features

#the correlation matrix heatmap for analyzing the correlation among features

```
corr_martix=iris_df[cols].corr()
```

to make hard copy of pictures

```
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt
plt.tight_layout()
```

```
with PdfPages('heatmap.pdf') as pdf:
```

```
    sns.heatmap(corr_martix,cbar=True,annot=True,fmt='%.1f',cmap='coolwarm');
```

```
    pdf.savefig(bbox_inches='tight') # saves the current figure into a pdf page
```

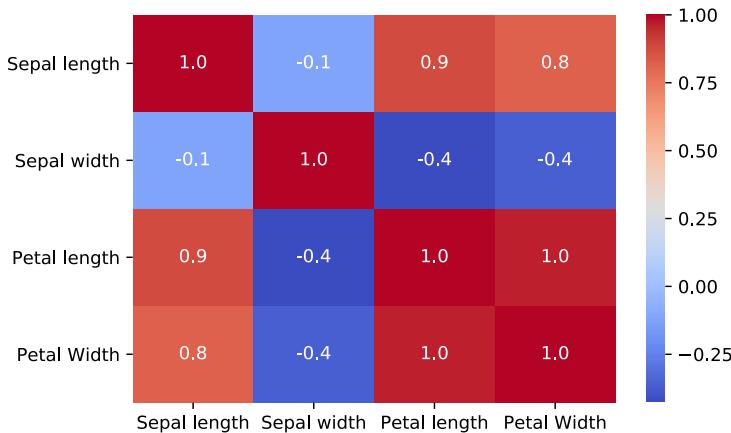


Figure 12.8: Heatmap with cross correlation matrix between features.

It can be noticed that petal length and petal width present high correlation, with similar values for sepal length and sepal width.

11.3.5. Decision Tree Implementation

As already mentioned, the concept behind decision trees is to select values of features (thresholds) that can best split the data according a criteria. Branches are added and the same is repeated until everything left are similar types of labels in a branch or there are no additional features to be splitted.

class Question:

```
def __init__(self,column,value):
```

```
    ""
```

initialization of column and value variables->

Example-> if (sepal length >= 2cm) ->

sepal_length == col and 1cm = value

```
    ""
```

```
    self.column = column
```

```
    self.value = value
```

```
def match(self,data):
```

“”

check if the data satisfy the criteria

returns true or false

“”

value = data[self.column]

return value >= self.value

def __repr__(self):

“”

Auxiliary method to print formatted question.

“”

condition = “>=”

return f’Is {cols[self.column]} {condition} {str(self.value)}?’

print(‘Testing the class Question...’)

create a question that takes column as 0 and value 5

q=Question(0,5)

checks if the feature at column 0 of the record #4 is greater or equal to 5

q.match(X[3])

Testing the class Question...

False

def count_values(rows):

“”

create a dictionary with the unique labels

dictionary key is specie and values are the frequencies

“”

count = dict()

for row **in** rows:

```
label=row[-1] # label is at the last column in a row
```

```
if label not in count:
```

```
    count[label] = 0
```

```
    count[label] += 1
```

```
return count
```

```
print('Testing count function...')
```

```
count_values(data)
```

```
Testing count function...
```

```
{0.0: 50, 1.0: 50, 2.0: 50}
```

```
def partition(rows,question):
```

```
    """
```

```
    split data based on Boolean value (true or false)
```

```
    """
```

```
# lists initialization
```

```
true_row, false_row = [], []
```

```
for row in rows:
```

```
    if question.match(row):
```

```
        #if question.match is true then value is stored as true,
```

```
        # otherwise append on false
```

```
        true_row.append(row)
```

```
    else:
```

```
        false_row.append(row)
```

```
return true_row,false_row
print('Testing partition function...')

true_rows, false_rows = partition(data,Question(0,5))

#thus true_rows contains only sepal length values greater than 5cm
len(true_rows)
Testing partition function...
```

128

11.3.6. Gini Impurity

As already mentioned, Gini impurity can quantify the right and wrong questions. So a function is implemented to evaluate Gini impurity. If there is no misclassification, then Gini impurity is equal null (0).

```
def gini(rows):
    """
    Checks gini impurity of a dataset.
    """
    count = count_values(rows)

    # impurity initialization
    impurity = 1
    for label in count:
        #probability of a unique label
        probab_of_label=count[label]/float(len(rows))

        impurity-=probab_of_label**2

    return impurity
print('Testing Gini Impurity...')
```

```
print('Gini Impurity for the whole dataset')
```

```
gini(data)
```

```
Testing Gini Impurity...
```

```
Gini Impurity for the whole dataset
```

```
0.6666666666666665
```

Another possible measure is the entropy, defined as a measure of chaos/randomness on a dataset. Mathematically, it can be expressed as,

$$S = \sum_x -p(x) \log(p(x))$$

```
def entropy(rows):
```

```
    entropy = 0
```

```
    count=count_values(rows)
```

```
    for label in count:
```

```
        p=count[label]/float(len(rows))
```

```
        entropy-=p*np.log2(p)
```

```
    return entropy
```

```
print('Testing Entropy of whole dataset....')
```

```
entropy(data)
```

```
Testing Entropy of whole dataset....
```

```
1.584962500721156
```

Define a function to obtain the information gain. This is a way of quantifying accuracy through data splitting and measuring the amount of information gained.

```
def info_gain_gini(current, left_branch, right_branch):
```

```
size_left = len(left_branch)
size_right = len(right_branch)
p = size_left/size_left + size_right

return current - p*gini(left_branch) - (1 - p)*gini(right_branch)
#weighted info gain
def info_gain_entropy(current, left_branch, right_branch):

    size_left = len(left_branch)
    size_right = len(right_branch)
    p = size_left/size_left + size_right
    return current - p*entropy(left_branch) - (1 - p)*entropy(right_branch)
```

11.3.7. Best Split

Define a function to evaluate the best split possible. This makes actually the core of the decision tree training, since it performs the decision regarding the best feature and value to split upon. For example, best split help to decide if the dataset should be splitted on petal length and at which value (i.e., 6.9 cm).

```
def best_split(rows):

    best_gain = 0
    best_question = None

    # obtain current gain
    current = gini(rows)

    # number of features
    features = len(rows[0]) - 1

    for col in range(features):
        # unique classes of feature
        values=set([row[col] for row in rows])
```

```
for val in values:
```

```
    question=Question(col, val)
    # divide data based on the question
    true_rows,false_rows = partition(rows, question)

    if len(true_rows)== 0 or len(false_rows) == 0:
        # check if the splitting produces no separation and ignore it.
        continue
```

```
# Gini Gain
```

```
    gain = info_gain_gini(current, true_rows, false_rows)
```

```
#if this gain is better than the actual best one then replace it.
```

```
    if gain >= best_gain:
```

```
        best_gain, best_question = gain, question
```

```
#iterate through each unique class of each feature and return the best
gain and best question
```

```
return best_gain, best_question
```

```
print('Testing best split...')
```

```
a, b = best_split(data)
```

```
print(b)
```

```
print(a)
```

```
Testing best split...
```

```
Is Petal length >= 6.9?
```

```
99.99552572706934
```

Up to this point it was adapted a top-down approach to solve each part of the decision tree algorithm separately. The next step consists in gathering together all this small parts and compile them together in a Decision Tree class which will be used to perform the classification on the data.

```
class DecisionNode:  
    # Class to contain the nodes in a tree  
    def __init__(self, question, true_branch, false_branch):  
        # question contains the column and values of the question attributed  
        # to node  
        self.question = question  
  
        self.true_branch = true_branch  
        self.false_branch = false_branch  
  
class Leaf:  
    # Each leaf is a node of a tree.  
    def __init__(self, rows):  
        self.predictions = count_values(rows)  
    The following function is used to recursively build a tree.  
    def build_tree(rows):  
  
        gain, question = best_split(rows)  
  
        # check if gain = 0, i.e., best split already satisfied  
        if gain == 0:  
            # return a leaf object  
            return Leaf(rows)  
        # Otherwise a useful feature and value can be used to partition.  
        true_rows, false_rows = partition(rows, question)  
  
        # Recursion part – build the true branch  
        true_branch = build_tree(true_rows)  
  
        # Recursion part – build the false branch  
        false_branch = build_tree(false_rows)  
  
        # return the root node with branches as well as the question
```

```
return DecisionNode(question, true_branch, false_branch)
```

```
print('Testing to build the tree...')
```

```
tree = build_tree(data)
```

```
Testing to build the tree...
```

An important observation is that, since there is no control over the depth (amount of leaf nodes) of the decision tree, it may be highly overfitting and may perform poorly on new dataset.

12

Principal Component Analysis

CONTENTS

12.1. Introduction.....	220
12.2. Mathematical Concepts	220
12.3. Principal Component Analysis Using Python	227

12.1. INTRODUCTION

This chapter presents the very basic concepts behind Principal Component Analysis (PCA). This statistical tool is widely used in problems that require data compression due to high dimensionality of it, such as image processing. It is also used when it is necessary to find patterns of high dimension data.

In the first part of this chapter, some basic mathematical concepts that are important to understand before using PCA. The following concepts will be presented:

- standard deviation;
- covariance;
- eigenvectors;
- eigenvalues.

Some examples will be used to illustrate how PCA works. The examples are a modification of the one presented by Smith (2002).

12.2. MATHEMATICAL CONCEPTS

12.2.1. Standard Deviation

Standard deviation is an important concept in statistics, which helps to understand how a population (of anything not only people) is distributed. Such statistical analysis is usually based on obtaining a sample from the population. It is essential that this sample from the population is a representative one from the whole dataset, i.e., it can be said that any measurement taken into this sample produces the same metrics as if it was done on the whole population. This is usually guaranteed by taking a significant amount of records from the original sample, or by multiple sampling. Consider the following sample extracted from a certain population.

$x = [1 \ 3 \ 5 \ 7 \ 11 \ 17 \ 22 \ 41 \ 68 \ 67 \ 62 \ 84]$

Referring to the whole sample can be done by calling it as x . If it is necessary to refer to one of the records in the sample, one can make use of subscripts, such as x_3 refers to the third element in the sample x , or the number 5.

The mean of the sample above can be obtained by summing up all the values and dividing by the number of records, as shown in the following formula.

$$x_m = \frac{\sum_{i=1}^n x_i}{n}$$

Where n is the number of samples. Unfortunately, the mean only does not provide enough information to understand the dataset. To understand this, consider the following two samples y and z , both having the same mean (2.5) with the exactly same amount of records, but being clearly different.

$$y = [1, 2, 3, 4]$$

$$z = [0.5, 1, 1.5, 7]$$

The sample z is clearly more spread (going from 0.5 to 7) than the sample y , though both show the same mean. The standard deviation provides a way of measuring such spread, thus helping to better understand a sample.

The standard deviation can be formally defined as “The average distance from the data mean to any point.”

The formula to compute the standard deviation consists in subtract each point from the mean (distance), square it, sum all and divide by $n-1$, taking the positive square root. Thus,

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x - x_m)^2}{n-1}}$$

Applying the above formula on the samples y and z , one can obtain the standard deviation of such samples.

y	$(y - y_m)$	$(y - y_m)^2$
1	-1.5	2.25
2	0.5	0.25
3	0.5	0.25
4	1.5	2.25

Total		5.00
Total/(n-1)		1.67
Square root		1.29
z	$(z - z_m)$	$(z - z_m)^2$
0.5	-2	4
1	-1.5	2.25
1.5	1	1.25
7	4.5	20.25
Total		27.25
Total/(n-1)		9.17
Square root		3.03

Notice how the standard deviation of sample z (3.03) is higher than of sample y (1.29), which indicates greater deviation from the mean, i.e., more spreading. Compare now both results with the following sample, also with mean equal to 2.5.

[2.5, 2.5, 2.5, 2.5]

In this case, the standard deviation is equal to 0.0, which indicates that there is absolute no deviation from the mean.

12.2.2. Variance

Another common measure for sample spreading is the variance, which consists mathematically in the squared standard deviation.

$$\sigma^2 = \frac{\sum_{i=1}^n (x - x_m)^2}{n - 1}$$

Exercises

Determine the mean, standard deviation and variance of the following samples.

[12 21 32 45 54 66 78 86 91]

[12 14 16 19 81 86 88 90 92]

[12 15 16 17 21 23 90 91 92]

12.2.3. Covariance

The statistical measurements mentioned above (mean, standard deviation and variance) are related to a single sample, and can be calculated independently. They are 1-dimensional values which can describe datasets such as house sizes, product prices, exam scores, etc.

However, many datasets have more than one dimension, and one may want to determine if there is any relation between the dimensions of the dataset. As an example, consider a dataset composed by the weight of students and their exam scores. One could aim at investigating if the weight of a student is somehow related with his exam scores.

Covariance metrics serves such purpose. It is always used between two dimensions, or samples, to quantify the relationship between two samples. If the covariance is applied on a dataset against itself, then it produces the variance. As an example, for a dataset composed by three samples (x , y and z), then covariance can be determined between x and y , x and z , and between y and z . Measuring the covariance of x and x would produce the variance of x .

Using the variance formula, expand the squared difference between samples and mean to obtain,

$$\sigma^2 = \frac{\sum_{i=1}^n (x - x_m)(x - x_m)}{n-1}$$

Covariance can be mathematically expressed in a similar formula, where one of the differences of mean and sample comes from the second dataset, thus,

$$cov(x, y) = \frac{\sum_{i=1}^n (x - x_m)(y - y_m)}{n-1}$$

Putting up in words, covariance can be expressed as, “For each point, multiply the difference between point x and the average of x with the difference of y and the average of y . Add all these products and divide by the dataset size minus 1.”

Suppose the following example. Students were interviewed regarding their weight and their last exam scores. Therefore, the dataset consists in 2 dimensions, the weight w of each student and the exam scores m .

Let $\text{cov}(w, m)$ be the covariance between the weight and the exam scores. This value can be interpreted first regarding its sign (positive, negative or zero). If the covariance value is positive, it indicates a positive correlation between the samples, i.e., as the student weight increases, his score also increases. On the other hand, if the covariance is negative, it indicates that if the weight increases, then the score decreases, and vice-versa. A covariance equals to zero indicates no relation between the variables.

Additionally, it is important to notice that $\text{cov}(x \ y)$ is exactly the same as $\text{cov}(y, x)$, since the only change is the commutation of x and y in the formula, which does not change its value.

Sample	Weight (kg)	Exam Scores (0–1)
1	81.83	0.8
2	88.11	1.0
3	63.57	0.0
4	86.88	0.9
5	73.64	0.5
6	73.18	0.4
7	75.96	0.5
8	73.95	0.4
9	84.64	0.9
10	62.32	0.0
Totals	764.07	5.4
Average	76.4	0.54

Covariance:

Sample	w	m	$(w - w_m)$	$(m - m_m)$	$(m - m_m)(w - w_m)$
1	81.83	0.80	5.42	0.26	1.41
2	88.11	1.00	11.70	0.46	5.38
3	63.57	0.00	-12.84	-0.54	6.93
4	86.88	0.90	10.47	0.36	3.77
5	73.64	0.50	-2.77	-0.04	0.11
6	73.18	0.40	-3.22	-0.14	0.45
7	75.96	0.50	-0.45	-0.04	0.02

8	73.95	0.40	-2.46	-0.14	0.34
9	84.64	0.90	8.23	0.36	2.96
10	62.32	0.00	-14.09	-0.54	7.61
Totals					28.99
Average					2.889

When it becomes necessary to consider the covariance between more than two dimensions, then a set of covariance measurements are collected. As in an example already mentioned, if one need to obtain the covariances on the dataset (x, y, z), then the covariances obtained are $cov(x, y)$, $cov(x, z)$ and $cov(y, z)$. For a n dimensional dataset, the number of covariances possible is,

$$\frac{n!}{(n-2)! \times 2}$$

A useful way to show all the covariances in a dataset is to put them in matrix. Remember that $cov(x, y) = cov(y, x)$. The covariance matrix can be stated as,

$$C^{n \times n} = (c_{i,j})$$

where $c_{i,j} = cov(dim_i, dim_j)$. The covariance matrix contains n rows and n columns. For example, building the covariance matrix for the dataset (x, y, z) gives.

$$C = \begin{bmatrix} cov(x, x) & cov(x, y) & cov(x, z) \\ cov(y, x) & cov(y, y) & cov(y, z) \\ cov(z, x) & cov(z, y) & cov(z, z) \end{bmatrix}$$

It is important to mention some properties of the matrix above. At the main diagonal, the covariances are $cov(x, x)$, $cov(y, y)$ and $cov(z, z)$, which returns the covariance of the variable with itself. Also, the matrix is symmetrical bout the main diagonal, since $cov(a, b) = cov(b, a)$.

Exercises

Obtain the covariance between the sample x and y below. Mention what this value indicates about the data.

Record	x	y
1	-2.85	10.18
2	-0.65	3.84
3	-0.11	11.63
4	-0.74	16.78
5	-2.89	4.03
6	-0.76	5.92
7	-1.62	13.81
8	0.53	8.67
9	-0.72	4.26

Obtain the covariance matrix for the following dataset composed by three samples x , y and z .

Record	x	y	z
1	9	2	7
2	-1	0	0
3	2	15	-1
4	-11	16	-2
5	1	12	1
6	2	-4	-1
7	-12	14	-3
8	-5	2	6
9	7	17	1

12.2.4. Eigenvectors and Eigenvalues

As already shown in the linear algebra chapter, matrix multiplication can be only done with the dimensions of the matrices are compatible. As an example, consider the following matrix multiplications.

Multiplication of matrix with a non-eigenvector.

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 11 \\ 5 \end{bmatrix}$$

Multiplication of a matrix with one eigenvector

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 8 \end{bmatrix} = 4 \times \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

In the first example (multiplication with non-eigenvector), the resulting vector cannot be transformed by an integer back into the original vector used in the multiplication. However, on the second case, the resulting vector is exactly a multiple of the vector used originally in the multiplication. This occurs because the vector represents an arrow that starts in the origin and points towards the coordinates (3,2). The multiplication of the vector with the matrix consists in a transformation of it, changing its original position.

Therefore, by definition an eigenvector of a linear transformation is a vector that is just modified by a scalar value when the transformation is performed on it. In a formal sense, consider T as a linear transformation from a vector space V on a field F into itself and v is a vector in V (not null), so v is an eigenvector of T if $T(v)$ is a scalar multiple of v .

$$T(v) = \lambda v$$

where λ is a scalar value, also called eigenvalue or characteristic value.

12.3. PRINCIPAL COMPONENT ANALYSIS USING PYTHON

Principal Component Analysis consists in a statistical tool to identify patterns in a dataset and reconstruct it in a way that information such as similarity can be easily retrieved. For instance, consider a dataset of higher dimensions such as 5 or more. In such case, graphical representations are not possible, and it is specially in those situations that PCA comes more at hand.

A main advantage of PCA over some other techniques is that, it is not only able of compressing data by dimensionality reduction. It also loses minimal information from the original data. That is one reason it has found so much popularity in image compression.

The following sections will guide the reader on how to apply PCA on a simple dataset.

12.3.1. Libraries Used

Numpy: for linear algebra computation; Matplotlib: for visualization

```
import numpy as np
```

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

```
from matplotlib.backends.backend_pdf import PdfPages
```

12.3.2. The Data

For the sake of simplicity, we generate the data to be used. It consists in a 2-dimensional set. Using this number of dimensions is convenient for graphical visualization whenever necessary.

The following code is used to generate the data, which is shown in the table following the code.

```
x = np.random.normal(2, 3, 10)
y = x + np.random.normal(-1, 2, 10)
print('| x | y |')
for i in range(len(x)):
    print(f'| {x[i]:.2f} | {y[i]:.2f} |')
print('\n'*2)
```

<i>x</i>	<i>y</i>
-2.16	-3.64
4.28	0.37
6.91	5.02
4.76	0.95
6.44	3.02
-0.20	-0.80
2.95	4.19
0.46	1.20
3.45	5.61
2.62	2.89

Figure 12.1 shows a plot of the original data.

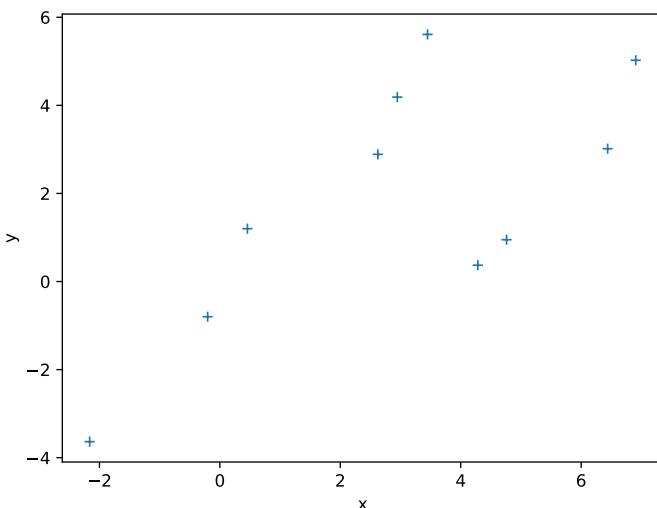


Figure 12.1: Plot of original dataset.

The first step after obtaining the data consists in removing the mean of each dimension. Therefore, each the average x_m is subtracted from each point x , while the average y_m is subtracted from each y . This produces a zero-mean dataset.

```
xs = x - np.mean(x)
```

```
ys = y - np.mean(y)
```

```
print('| x | y |')
for i in range(len(x)):
    print(f'| {xs[i]:.2f} | {ys[i]:.2f} |')
```

```
print('\n'*2)
```

<i>x</i>	<i>y</i>
-5.11	-5.52
1.33	-1.51
3.96	3.14

1.81	-0.93
3.49	1.13
-3.15	-2.68
-0.00	2.31
-2.49	-0.68
0.50	3.73
-0.33	1.01

Following, calculate the covariance matrix. Its dimension is 2x2 because the dataset has 2 dimensions.

```
cov_mat = np.cov(x,y)  
print(cov_mat)  
[[8.39489351 5.84121688] [5.84121688 8.07036548]]
```

As expected, the values outside of the main diagonal are positive, which indicates that when x increases, y also increases, as it can be seen in Figure 13.1.

Obtain the eigenvectors and eigenvalues of the covariance matrix. This is one of the most important steps, since they provide relevant information about the data.

```
eigenval, eigvec = np.linalg.eig(cov_mat)  
print('eigenvec')  
print(eigvec)  
  
print('eigenval')  
print(eigenval)  
eigvec [[ 0.71685718 -0.69722004] [ 0.69722004 0.71685718]]  
eigenval [14.07609971 2.38915927]
```

Since the dataset is two-dimensional, there are two eigenvalues and two eigenvectors, each one two-dimensional. To choose the principal, most important vector one has to look at the values of the eigenvalues. The highest eigenvalue is the most important one. If the first eigenvalue is the highest one, then the first eigenvector is also the most relevant.

We select the best eigenvalue using argsort function in Python, which retrieves the indexes which would sort the values ascending. Then the last value from the reordered list of indexes is used to isolate the best eigenvector.

```
best_idx = np.argsort(eigenval)
main_eigvec = eigvec[:, best_idx[-1]]
chosen vec [0.71685718 0.69722004]
```

To obtain the simplified, compressed dataset using PCA, use the chosen eigenvector(s) to be kept, and multiply it by the transposed adjusted dataset, i.e., the one with zero-mean. Mathematically,

$$X_{PCA} = E \times X_s^T$$

where X_{PCA} is the dataset after PCA, E is the set of eigenvector(s) to be kept (in the present case, 1) and X_s is the dataset with subtracted mean (zero-mean). Here X means both dimensions of the dataset x and y .

```
PCA      = main_eigvec.dot(np.concatenate((xs.reshape(-1,      1),
                                         ys.reshape(-1, 1)), axis = 1).T)
print('PCA')
print(PCA)
PCA [-7.51198525 -0.09689408  5.02743813  0.64876661  3.29294885
-4.12900809 1.60511379 -2.26171932 2.95747785 0.46786151]
```

This one-dimensional dataset consists in the compressed information contained in the original dataset, with minor information loss.

To retrieve the original dataset, one can perform the inverse operation, assuming that the inverse of the eigenvector is the same as the transposed version of it.

$$X_s = E^T \times X_r$$

This will return the adjusted dataset (with zero mean). To obtain the dataset with the original mean, then it is necessary to sum the mean values of each dimension.

$$X = E^T \times X_r + X_m$$

where X is the original 2-dimensional dataset and X_m are the mean values. Naturally, if not all eigenvectors are used to recover the original data, then the obtained one is not exactly the same as the one initially obtained.

```
original = main_eigvec.reshape(-1, 1).dot(PCA.reshape(-1, 1).T) +
np.array([[np.mean(x)], [np.mean(y)]])
print('original')
print(original)
original [[-2.43449761 2.88106374 6.55447808 3.41559596 5.31109699
-0.009386144.1011603 1.32919322 5.07061219 3.28591284][-3.35673729
1.81321288 5.38599998 2.33310245 4.1766793 -0.99805781 2.99988687
0.30385333 3.94278219 2.20697179]]
```

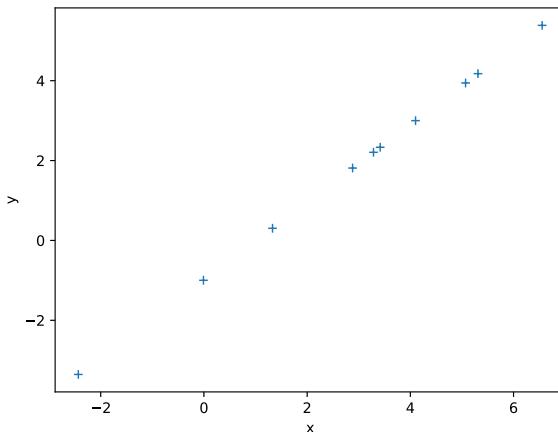


Figure 12.2: Reconstructed dataset.

Observe in the Figure above that the variations on one of the dimensions, considered less significant, is completely removed. This is due the elimination of one of the eigenvectors before reconstructing the dataset.

Exercises

1. What is a covariance matrix?
2. What is the main step to compress the data in PCA?

13

Classification Using k -Nearest Neighbor Algorithm

CONTENTS

13.1. Introduction.....	234
13.2. Principles and Definition of KNN	234
13.3. Algorithm	235
13.4. Example and Python Solution	237

13.1. INTRODUCTION

k-Nearest Neighbor (kNN) is a relatively simple algorithm, easy to implement and still a powerful tool to understand and solve classification problems. In this chapter the reader will develop an understanding of the principles used to develop the kNN algorithm, when it can be used. The latter part of the chapter shows a simple example and how to implement it with Python.

13.2. PRINCIPLES AND DEFINITION OF KNN

Consider that four groups of people are talking in a room. What would be most expected: that people talking in the same group have more affinity between each other or less affinity? Naturally, one would expect that people “grouped” together have more affinity between each other than with people from other groups. Extensively, people in the same group probably share common traits, which attracts one to the other.

kNN is based on this same principle. It assumes that data “grouped” together are more similar to their neighbors and thus may be classified with the same label as its neighbor(s). To illustrate, consider two neighborhoods. One has many high-class houses, with expensive cars and where very conservative people live. The second neighborhood is less fancy, with lower quality houses. For a person which has a well-paid job and an expensive car, conservative, would it probably live on the first or on the second neighborhood? kNN works by fitting such person on the expected neighborhood it may live (though it may sometimes fall wrong). The term “ k ” indicates the number of nearest neighbors being considered when classifying an unknown element. For instance, $k = 1$ uses only the one nearest neighbor as an indicator. For $k = 3$, it uses the vote of the majority, i.e., to which class most of the neighbors belong? Nearest or Farthest is measured by the distance between the point to be classified and the neighbor. There are different possible metrics, being **Euclidean distance** the most common one.

Neighbors-based method are referred to as non-generalizing machine learning models, because they work only by “remembering” the training data. Therefore, it does not contain any intricate calculation, only the measurement of the distance(s) and the choice of the label. An element is classified according the majority vote of the neighbors being considered.

13.3. ALGORITHM

Initially, a set with known labels is presented in the kNN model, such as the one shown in the Figure 13.1.

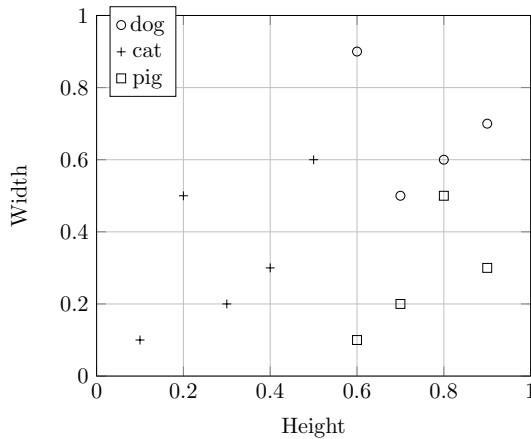


Figure 13.1: Example dataset containing data with three labels.

kNN model stores this dataset in its inner structure so as to “remember” the labels and the position of each data. Let a new point p be presented to kNN. The location of this point in relation to the whole dataset is presented in Figure 13.2.

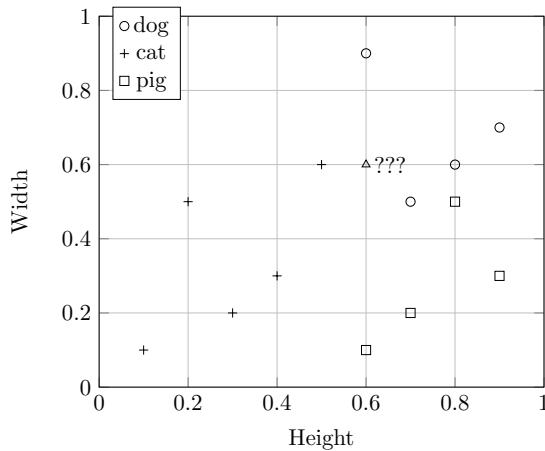


Figure 13.2: Point with unknown class plotted with the kNN dataset.

A number of neighbors must be chosen before kNN can classify the new point. For instance, if $k = 1$, then kNN will check the class of the nearest point as shown in the Figure below. The distance is measured using Euclidean distance in this case.

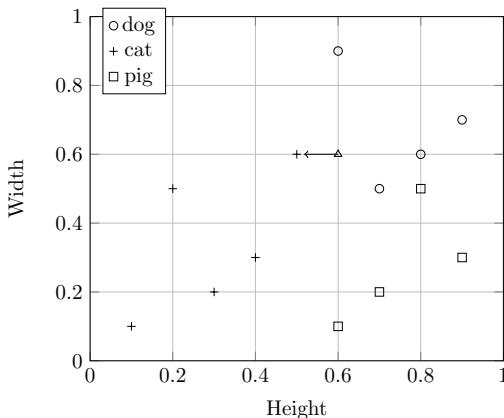


Figure 13.3: Point and distance to nearest neighbor ($k = 1$).

Since the class of the nearest neighbor is “cat,” then the point is also classified as “cat,” due to similarity. On the other hand, suppose that $k = 3$, as in Figure 14.4.

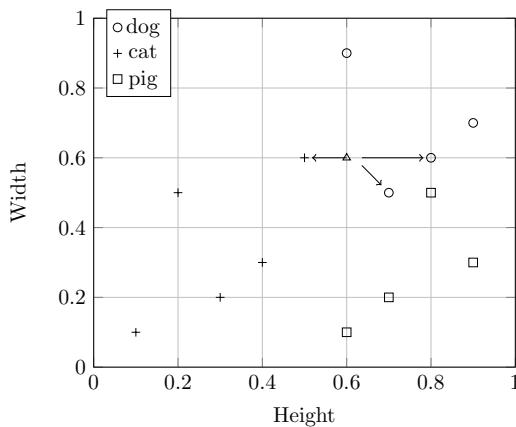


Figure 13.4: Point and distance to nearest neighbors ($k = 3$).

Since two neighbors belong to class “dog” and one neighbor belongs to class “cat,” per majority of vote the unknown point is classified as “dog.” This example illustrates that the choice on the number of neighbors and the presence of outliers can interfere with the classification process in kNN.

13.4. EXAMPLE AND PYTHON SOLUTION

13.4.1. Example

In the example above, it was illustrated, without direct calculation but only graphically, how kNN algorithm works. In this part, let's use the same example but using the relatively simple mathematics behind kNN to perform the classification. The dataset, as illustrated in Figure 13.1, is the one shown in Table 13.1. This is an illustrative example and the values given of height and width are to be considered processed values, i.e., not real-world ones.

Table 13.1: Dataset for kNN Example – Animal Classification

Height	Width	Class
0.1	0.1	cat
0.2	0.5	cat
0.3	0.2	cat
0.4	0.3	cat
0.5	0.6	cat
0.6	0.1	pig
0.7	0.2	pig
0.8	0.5	pig
0.9	0.3	pig
0.6	0.9	dog
0.7	0.5	dog
0.8	0.6	dog
0.9	0.7	dog

Initially, this dataset is given to the kNN model with all the values of the features and the classes. As already mentioned, kNN will classify new data based on this one, i.e., by measuring the distances between the new point and the available data.

Let a new animal, with unknown class (either cat, dog or pig), with height 0.6 and width 0.6.

Table 13.2: Unknown Animal To Be Classified

Height	Width	Class
0.6	0.6	???

The task of the kNN algorithm is to attribute a class to such animal. To do so, it measures the distances between each point already known and the new one. A common metrics for this is the Euclidean distance, which uses the formula,

$$dist = \sqrt{\sum(d_i - d_j)^2}$$

where d_i is the value of point A at dimension d , which could be x , y , etc. d_j is the value for the second point B at the same dimension. For a 2-dimensional problem such as the current one, the distance formula can be written as,

$$dist = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

where i is the point already known and j is the point with unknown class. Table 13.3 summarizes the calculated distances for all the dataset.

x_i	x_j	Class of i	Dist
(0.1, 0.1)	(0.6, 0.6)	cat	0.71
(0.2, 0.5)	(0.6, 0.6)	cat	0.41
(0.3, 0.2)	(0.6, 0.6)	cat	0.50
(0.4, 0.3)	(0.6, 0.6)	cat	0.36
(0.5, 0.6)	(0.6, 0.6)	cat	0.10
(0.6, 0.1)	(0.6, 0.6)	pig	0.50
(0.7, 0.2)	(0.6, 0.6)	pig	0.41
(0.8, 0.5)	(0.6, 0.6)	pig	0.22
(0.9, 0.3)	(0.6, 0.6)	pig	0.42
(0.6, 0.9)	(0.6, 0.6)	dog	0.30
(0.7, 0.5)	(0.6, 0.6)	dog	0.14
(0.8, 0.6)	(0.6, 0.6)	dog	0.20
(0.9, 0.7)	(0.6, 0.6)	dog	0.32

The following step consists in reordering the distances from the lowest to the highest. Lower distances means near neighbors, while higher distances are more far neighbors.

Class of i
cat
dog
dog
pig
dog
dog
cat
cat
pig
pig
cat
pig
cat

The above steps are the same independent of the k value chosen. However, from here on, one needs to set the value of k so the algorithm chooses the number of neighbors that can vote. The class is attributed according vote majority. As an example, consider that we choose $k = 3$, then only the three neighbors with the least distance to be point can vote.

Class of i
cat
dog
dog

Since the above set is constituted by one cat and two dogs, the attributed class to the unknown point is “dog.” Notice that one should be able to validate the experimental data and double check if the algorithm performs reasonable classifications. This is because the point could easily be a cat, since this class has the least distance (it would be classified as cat if $k = 1$).

Another important point to mention is, though we have selected $k = 3$, it is usually best practice not to choose even values or multiples of the number of classes to k . That can help to avoid ties.

13.4.2. Implementation with Python

In this section, we implement the kNN algorithm with Python to perform classification on data based on a given dataset. Consider the example already

given on Table 4.1 as our training data, i.e., the dataset which will guide the kNN to perform the classification.

Define the dataset as a list of tuples in Python. Each tuple contains the height, width and class of each animal.

```
data = [(0.1, 0.1, 'cat'),  
        (0.2, 0.5, 'cat'),  
        (0.3, 0.2, 'cat'),  
        (0.4, 0.3, 'cat'),  
        (0.5, 0.6, 'cat'),  
        (0.6, 0.1, 'pig'),  
        (0.7, 0.2, 'pig'),  
        (0.8, 0.5, 'pig'),  
        (0.9, 0.3, 'pig'),  
        (0.6, 0.9, 'dog'),  
        (0.7, 0.5, 'dog'),  
        (0.8, 0.6, 'dog'),  
        (0.9, 0.7, 'dog')]
```

Separate the data into the input features (X_{train}) and the labels (y_{train}). The input features are the first two elements of each tuple, and the label is the last element.

```
X_train = [(x[0], x[1]) for x in data]  
y_train = [x[2] for x in data]
```

Define the testing set, i.e., the point with the unknown label as shown in Table 14.2.

```
x_test = (0.6, 0.6)
```

To evaluate the distance between each point, create a function `euclidean` which, by receiving as input two points, measure the euclidean distance between each other.

```
def euclidean(xi, xj):  
    dist = ((xi[0] - xj[0])**2 + (xi[1] - xj[1])**2)**0.5  
    return dist
```

The following step consists in obtaining the distances between each point in X_train and the point x_test. These distances need to be sorted ascending, and the labels in y_train have to be sorted using the same order as the distances vector.

obtain the distances between x_test and each point of X_train

```
distances = [euclidean(xi,x_test) for xi in X_train]
```

```
[print(f'{distance:.2f}') for distance in distances]
```

retrieve the order of the indexes that would sort the distances

```
sort_idx = sorted(range(len(distances)), key=lambda k: distances[k])
```

sort the distances

```
distances.sort()
```

use sort_idx to sort the labels in the same order as distances

```
y_train = [y_train[i] for i in sort_idx]
```

```
[print(f'{y}|{distance:.2f}') for y,distance in zip(y_train,distances)]
```

Once the distances are known and sorted with the labels, it is time to choose the number of neighbors k that can vote. Let it be k = 3, then create a vector votes which gets from y_train only the k nearest neighbors, and evaluate which labels appear most in this set.

set the number of nearest neighbors

```
k = 3
```

from y_train get only the nearest neighbors

```
votes = y_train[:k]
```

```
count_votes = dict()
```

initialize the majority votes to zero

```
best_vote = 0
```

```
# initialize the winner to empty string
winner = ''
for vote in votes: # evaluate each possible vote
    if vote not in count_votes: # add the label to count_votes if is not already
        there
            count_votes[vote] = 0
            count_votes[vote] += 1 # add a vote to the corresponding label

    if count_votes[vote] > best_vote: # check if the label has the majority and
        attribute it
        best_vote = count_votes[vote]
        winner = vote

ypred = winner # set the final winner according the majority of votes

print(ypred) # dog
```

The final value of ypred for this example is “dog,” which confirms that the code is correct and the algorithm works properly. The following code shows a compilation of the complete exercise (except for the data) above using functional programming approach.

```
def sorted_distances(X_train, y_train, x_test):

    distances = [euclidean(xi, x_test) for xi in X_train]

    sort_idx = sorted(range(len(distances)), key = lambda k: distances[k])

    distances.sort()

    y_train = [y_train[i] for i in sort_idx]

    return distances, y_train
```

```
def predict_class(y_train, k):
```

```
    votes = y_train[:k]
```

```
    count_votes = dict()
```

```
    best_vote = 0
```

```
    winner = ''
```

```
    for vote in votes:
```

```
        if vote not in count_votes:
```

```
            count_votes[vote] = 0
```

```
            count_votes[vote] += 1
```

```
        if count_votes[vote] > best_vote:
```

```
            best_vote = count_votes[vote]
```

```
            winner = vote
```

```
    ypred = winner
```

```
    return ypred
```

```
def kNN(X_train, y_train, x_test, k = 3):
```

```
    distances, y_train = sorted_distances(X_train, y_train, x_test)
```

```
    ypred = predict_class(y_train, k)
```

```
    return ypred
```

```
print('Function test')
```

```
print(kNN(X_train, y_train, x_test, k = 3))
```

13.4.3. Using SkLearn Library

Sklearn has developed a library neighbors, which incorporates various functionalities for nearest neighbors classifiers in both unsupervised and supervised learning.

Both scipy sparse matrices and numpy arrays can be used with this library. A variety of distances metrics can be used for dense matrices, while for sparse ones the Minkowski distance metrics is the one available.

There are two types of nearest neighbor classifiers implemented in sklearn:

KNeighborsClassifier Uses the k nearest neighbors for classification. This is the most popular classifier between the two mentioned here. RadiusNeighborsClassifier chooses the number of neighbors within a radius r of the dataset. This classifier is seldom used.

What follows is an example code of Python to implement kNN using sklearn.neighbors library.

```
from sklearn.neighbors import KNeighborsClassifier
```

```
knn = KNeighborsClassifier(n_neighbors = 5,
```

```
    weights = 'uniform'
```

```
    algorithm = 'auto',
```

```
    leaf_size = 30,
```

```
    p = 2,
```

```
    metric = 'minkowski',
```

```
    metric_params = None,
```

```
    n_jobs = None)
```

```
knn.fit(features, labels)
```

```
print("How to do predictions")
```

```
print(knn.predict(features_test))
```

```
print("Target")
```

```
print(labels_test)
```

14

Introduction to KMeans Clustering

CONTENTS

14.1. How Kmeans Works?	246
14.2. Kmeans Algorithm	247

In most of the examples showed along this book the discussion focused on supervised learning algorithms. In such cases, the model knows the labels (correct values) that it should predict, and it tries to reproduce such values to its best.

On the other side there are unsupervised learning algorithms. They do not use a set of known labels to “classify” the data. Rather it uses different assumptions about the data to find patterns on it and create clusters, i.e., groups of data. Kmeans is one of such algorithms.

14.1. HOW KMEANS WORKS?

A number of K clusters is initially set manually. Through iteration, the algorithm assigns each datapoint to one of the clusters based on their position in n-dimensional space (1D, 2D, etc.). The clustering is performed according data similarity, i.e., similar data is joined under the same cluster. Similarity may be measured by different criteria, depending on the problem. After the iteration, Kmeans produce:

- The centroid of the K clusters, used to label the data and new data;
- Labels for the training data.

It is important to notice that each point can only be assigned to a single cluster. The final results enables one to analyze the data and determine patterns on it. Qualitative examination of each clusters assists one in understanding what kind of group each cluster represents.

KMeans clustering (and clustering algorithms in general), is commonly used in a variety of fields (Trevino (2016)):

- Behavioral segmentation:
 - Purchase history segmentation;
 - Activity Segmentation (application, website, or platform);
 - Definition of personas based on interests;
 - Creation of profiles through activity monitoring.
- Inventory categorization:
 - Group inventory by sales activity;
 - Group inventory by manufacturing metrics.
- Sorting sensor measurements:

- Detect activity types in motion sensors;
- Group images;
- Separate audio;
- Identify groups in health monitoring.
- Detecting bots or anomalies:
 - Separate valid activity groups from bots;
 - Group valid activity to clean up outlier detection.

14.2. KMEANS ALGORITHM

Step 1:

Initially, a number K of centroids c is placed in the n -dimensional space containing the datasets. Ideally, the clusters should be widely spread over the space, not dislocated or grouped together.

Then each data point is assigned to the cluster nearest to it. One way of defining the distance is by using the Euclidean distance metrics. Other options are the Cosine and the Manhattan distances. Mathematically, each point x is assigned to the cluster based on,

$$\text{argmin} \text{dist}(c_i, x)$$

where $\text{dist}()$ is the Euclidean distance between the centroid c_i and the point x . Let the sample attributed to cluster c_i to be S

Step 2:

The position of each centroid is updated, by dislocating it to the central position among S . This is done by taking the average value of each dimension and attributing it to the cluster c_i as follows.

$$c_i = \frac{1}{S} \sum S_i$$

Then go back to step 1.

The iteration is performed until a stopping criteria is met. This may be:

- datapoints do not change clusters;
- maximum number of iterations;

- position of clusters does not change significantly (above a certain threshold).

One great advantage of KMeans is that it ways converges, though not guaranteed to be a global optimum (possibly a local optimum). This can be overcome by running the algorithm multiple times with random initializations and picking up the best one.

14.2.1. Choice of K

In many situations, the best way of choosing the amount of centroids K is by trial and error. Using this methodology, one evaluates the algorithm with a variety of K values and compares the results to which one makes the most sense. Still, there are some techniques that may be used to estimate the value of K.

A popular metrics to choose the value of K is the mean distance between data points and cluster centroids. With 1 cluster the mean distance is the greatest, with continuous decrease until it reaches zero when the number of centroids is the same as the number of points. This shows that this method cannot be used isolated. Rather, it is coupled with other metrics such as the elbow point, which shows when the rate of mean distance sharply decreases. This point can be used to roughly estimate K.

Besides this one, techniques for validating K are usable, such as cross-validation, the silhouette method, information criteria, the information theoretic jump method and G-means algorithm.

14.2.2. Example of KMeans Using Python

In the following section, we consider an example of clustering a dataset with two features (for better visualization). The data derives from sklearn library, make blobs. The code below shows how to import the data and the plot shows how the data is spread along the 2-dimensional space.

```
from sklearn.datasets import make_blobs
```

```
n_samples = 1500
```

```
random_state = 999
```

```
X,y = make_blobs(n_samples = n_samples, random_state = random_state)
```

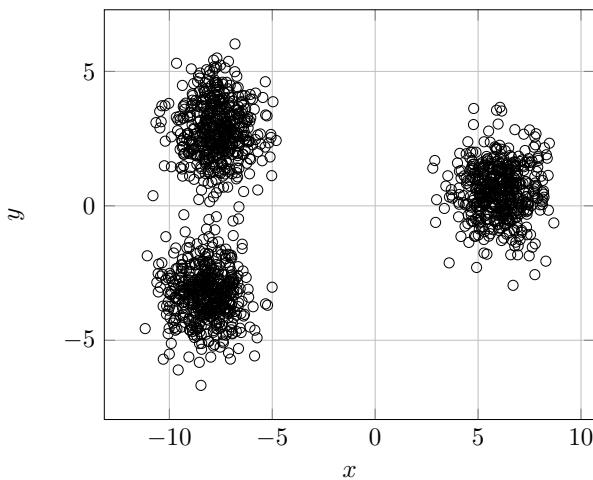


Figure 14.1: Dataset to be clustered.

In this academic example, it is clearly visible that there are three “groups” of data. However, for the sake of demonstration, let the number of clusters K to be equal 2 initially. Use Sklearn function Kmeans to evaluate the algorithm over this dataset.

$k = 2$

```
y_pred = KMeans(n_clusters = 2, random_state = random_state).fit_predict(X) (kMeans2.pdf)
```

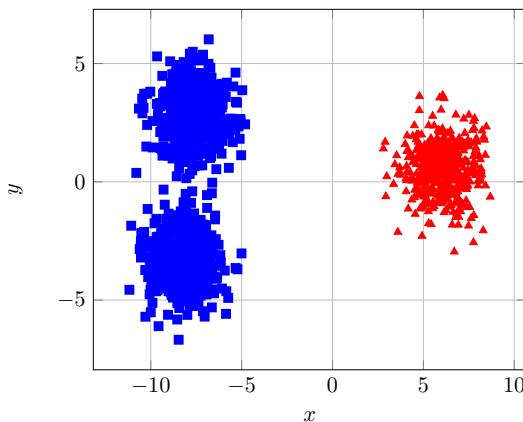


Figure 14.2: Clustering dataset with $k = 2$.

The following shows a comparison of the results using $k = 3$ and $k = 4$. Notice how the groups becomes splitted differently.

$k = 3$

```
ypred2 = KMeans(n_clusters = k, random_state = random_state).fit_
predict(X).astype(np.str)
```

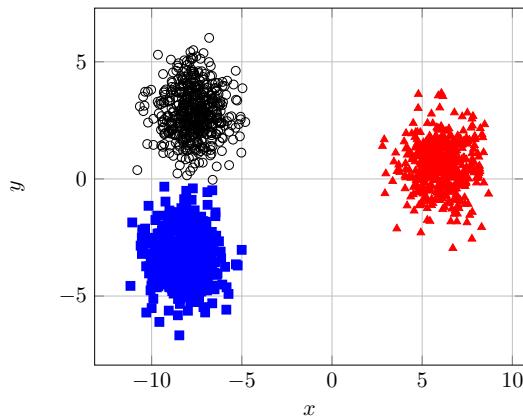


Figure 14.3: Clustering dataset with $k = 3$.

$k = 4$

```
ypred2 = KMeans(n_clusters = k, random_state = random_state).fit_
predict(X).astype(np.str)
```

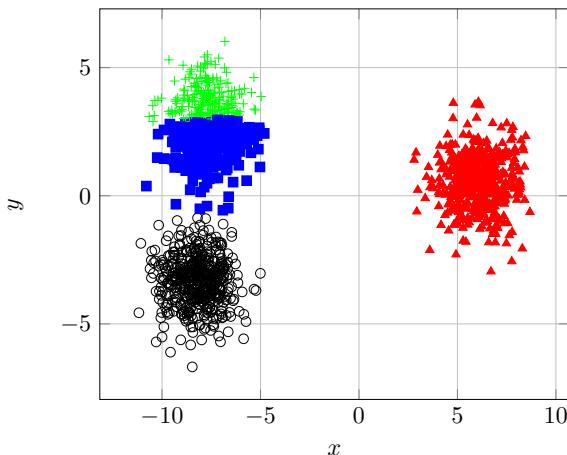


Figure 14.4: Clustering dataset with $k = 4$.

From the results above, it is clear that $k = 3$ is the ideal number of clusters, with a correct separation of the groups. Using a smaller number of k causes different groups to be joined together by similarity. Since the two groups at the right of the plot are clearly nearer, those are the ones joined. On the other hand, using a higher number of clusters causes group splitting.

The correct clustering of a dataset depends not only on the choice of K but also in the correct choice of dimensions (features) of the dataset. These dimensions should capture the variability of the data, which is an essential part to find all naturally occurring groups (Trevino, 2016).

15

Computing with TensorFlow: Introduction and Basics

CONTENTS

15.1. Installing Tensorflow Library	254
15.2. Tensors.....	255
15.3. Computational Graph and Session.....	255
15.4. Operating With Matrices.....	256
15.5. Variables.....	260
15.6. Placeholders.....	260
15.7. Ways Of Creating Tensors	262
15.8. Summary	265

Up to this point of the book, we have seen basic examples of machine learning models, developed with Python and with the libraries Numpy, Scipy among other auxiliary modules. For more advanced models, there are Python libraries that incorporates specific functionalities that makes it easier and more straightforward the development of algorithms without the need of “reinventing” things.

In this regard, TensorFlow is a library or framework developed specifically for deep learning. It was released in 2015 by Google and has been since then used by this company in several of its products, such as Google Search, e-mail filtering, speech recognition, Google Assistant, Google Now, and Google Photos.

This chapter provides a basic introduction to TensorFlow framework. Its structure is based on constructing partial subgraph computations, allowing distributed training (data parallelism) by conceptualizing every model as neural networks (model parallelism). There are different Application Programming Interface (APIs) associated with TensorFlow, but using core TensorFlow is the best way of having full programming control.

TensorFlow performs computations by describing them as a graph, where nodes are operations. These graphs are executed inside the context of a session. Therefore, computations are performed once a graph is launched in a session. Such session executes operations into CPU or GPU.

The following concepts are discussed along this chapter:

- Tensors
- Computational Graph and Session
- Constants, Placeholders and Variables
- How to create tensors
- Operating with matrices
- Operation with tensors

15.1. INSTALLING TENSORFLOW LIBRARY

To install the TensorFlow library, use the pip command. According its documentation, TensorFlow is currently tested and supported on the following 64-bit systems:

- Ubuntu 16.04 or later
- Windows 7 or later

- macOS 10.12.6 (Sierra) or later (no GPU support)
- Raspbian 9.0 or later

`pip install tensorflow`

Another option is to use TensorFlow directly in the browser with Colaboratory, a Google research project developed to assist in educating about machine learning and research. Basically it consists in a Jupyter Notebook with additional features. Colaboratory can be accessed through the URL <https://colab.research.google.com/notebooks/welcome.ipynb>.

A third option is to use it Repl.it, an online Python IDE that runs directly from the browser. Differently from Colaboratory, Repl.it is not a Jupyter Notebook but a complete IDE, containing text editor functionality with Python syntax highlighting and an integrated terminal to run the code online. Repl.it can be accessed though <https://repl.it>.

15.2. TENSORS

A tensor consists in the basic unit defined in TensorFlow. It is a generalization of scalars, vectors and matrices. In this sense, a scalar is a tensor of rank zero (0), while a vector is a tensor with rank 1. By analogy, it can be deduced that a matrix is a tensor with rank 2. In general, a tensor is by definition a n-dimensional array, where n is any integer value starting 0.

Consider the following examples of tensors:

- 10 – a tensor of rank 0, the shape of it is [];
- [10, 20] – a tensor of rank 1, the shape of it is [2].
- [[10, 20, 30], [100, 200, 300]] – tensor with rank 2. Its shape is [2,3];
- [[[10, 20, 30]], [[1, 2, 3]]] – tensor with rank 3, and shape [2,1,3].

15.3. COMPUTATIONAL GRAPH AND SESSION

TensorFlow core is based on two main actions:

- To construct the computational graph in the building phase;
- To run the graph at execution phase.

These actions are taken according the following steps, running inside TensorFlow core:

- A graph is defined by its nodes (operations) and edges (tensors);
- The building phase assembles the graph;
- In execution phase, a session runs the nodes (operations) in the graph.

For better understanding, the simplest operation that can be performed in TensorFlow is the attribution of a constant, taking no inputs but passing outputs to nodes performing certain operations (multiplication, addition, subtraction, etc.).

A session instance is generated by calling the `tf.Session()` constructor.

```
sess = tf.Session()
```

Once created, the session allocated the required resources (CPU and GPU) to execute the required operations. It also holds any intermediate results and variables. These variables and intermediate results are only valid inside the session object, so it cannot be retrieved in the global namespace or in another session.

In the following example, TensorFlow is used to simply store a variable. This illustrates the basic syntax of TensorFlow library.

```
import tensorflow as tf  
var = tf.constant(10)  
sess = tf.Session()  
print(sess.run(var))  
10
```

A tensor is composed by the following structure:

- **Rank** – Number of dimensions of the tensor.
- **Shape** – Number of rows, columns and additional dimensions defines the shape of a tensor.
- **Type** – Data type assigned to the tensor.

15.4. OPERATING WITH MATRICES

As already mentioned, TensorFlow graph consists of nodes, where operations are performed, and edges, which are tensors. Any algebraic operation can be performed using TensorFlow core, such as matrix addition, subtraction, multiplication, etc.

Consider the problem of determining the result of the following matrix multiplication.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

It can be performed using TensorFlow module through the following steps.

- Import TensorFlow module

```
import tensorflow as tf
```

- Define two edges of the graph by using `tf.constant()` instances.

```
matrix1 = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
matrix2 = tf.constant([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
```

- Define the node `tf.matmul()` which will perform the matrix multiplication.

```
matrix_prod = tf.matmul(matrix1, matrix2)
```

- Obtain the result by running the node in a TensorFlow session

```
with tf.Session() as sess:
```

```
    result1 = sess.run(matrix_prod)
```

```
    print(result1)
```

Gathering all together,

```
import tensorflow as tf
```

```
matrix1 = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
matrix2 = tf.constant([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
```

```
matrix_prod = tf.matmul(matrix1, matrix2)
```

```
with tf.Session() as sess:
```

```
    result1 = sess.run(matrix_prod)
```

```
    print(result1)
```

```
[[ 14  32  50]]
```

```
[ 32  77 122]
```

[50 122 194]]

In the following example, `tf.add()` node is used to perform matrix addition matrices defined above.

```
import tensorflow as tf
```

```
matrix1 = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
matrix2 = tf.constant([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
```

```
matrix_add = tf.add(matrix1, matrix2)
```

```
with tf.Session() as sess:
```

```
    result1 = sess.run(matrix_add)
```

```
print(result1)
```

```
[[ 2  6 10]
```

```
[ 6 10 14]
```

```
[10 14 18]]
```

Nodes can also be connected sequentially. For instance consider the following problem which consists in obtaining the solution of,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \\ -10 & -20 & -30 \\ 10 & 20 & 30 \end{bmatrix}$$

The above problem can be split in two parts:

- Perform the matrix multiplication;
- Perform the addition of the matrix multiplication result with the last matrix.

This can be done via TensorFlow by concatenating the nodes sequentially as demonstrated in the following code.

```
import tensorflow as tf
```

```
matrix1 = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
matrix2 = tf.constant([[1, 4, 7], [2, 5, 8], [3, 6, 9]])
```

```
matrix3 = tf.constant([[10, 20, 30], [-10, -20, -30], [10, 20, 30]])  
matrix_prod = tf.matmul(matrix1, matrix2)  
matrix_add = tf.add(matrix_prod, matrix3)  
with tf.Session() as sess:  
    result1 = sess.run(matrix_add)  
  
print(result1)  
[[24 52 80]  
 [22 57 92]  
 [60 142 224]]
```

The dimension of the result in a TensorFlow operation may be different from the dimension of the inputs. For example, the node `tf.matrix_determinant()` can be used to calculate the determinant of a matrix (tensor rank 2), generating a scalar (tensor rank 0), as shown in the following example.

```
import tensorflow as tf  
matrix1 = tf.constant([[1., 2., 3.], [-2., -5., -6.], [3., 2., 1.]])  
matrix_det = tf.matrix_determinant(matrix1)  
with tf.Session() as sess:  
    result1 = sess.run(matrix_det)  
  
print(result1)  
8.000002
```

TensorFlow contains all basic math operators. The following list shows some of the most common operators, which may require one or more arguments.

- `tf.add(a, b)`
- `tf.subtract(a, b)`
- `tf.multiply(a, b)`
- `tf.div(a, b)`
- `tf.pow(a, b)`
- `tf.exp(a)`

- `tf.sqrt(a)`

15.5. VARIABLES

A Variable in TensorFlow contains data which may change along processes. Instead of being an edge (as constants are), variables are stored as nodes.

A way to create a variable is to use the `get_variable` method.

```
tf.get_variable(name = “,”values,dtype,initializer)
```

Inputs:

- name(str) – Name of the variable
- values(list/array) – Dimension of the tensor
- dtype – Type of data. Optional
- initializer – (Optional)How to initialize the tensor

When an initializer is set, there is no need of setting the values since the dimensions are directly set from the initializer. In the following code, a two dimensional variable is created. The default value of a variable is set at random by TensorFlow.

```
# Create a variable  
var = tf.get_variable(“var,” [1, 3])  
print(var)  
<tf.Variable ‘var:0’ shape=(1, 3) dtype=float32_ref>
```

In the example above, the variable var is a vector of one row and 3 columns. To fix the initial values of the variable to zero, one can use the `zeros_initializer` function.

```
# Create an initializer to fix variables to zero  
init = tf.zeros_initializer  
  
# Create a variable  
var = tf.get_variable(“var,” [1, 3], initializer = init)  
print(var)  
<tf.Variable ‘var:0’ shape=(1, 3) dtype=float32_ref>
```

15.6. PLACEHOLDERS

A placeholder is a node used to feed a tensor. Its purpose consists in

initializing data which will be fed into tensor. The placeholder is fed using the feed_dict instruction when running a session. Therefore, a placeholder can only be fed withing a session.

The following syntax is used to create a placeholder.

```
tf.placeholder(dtype, shape = None, name = None )
```

Inputs:

- dtype – Data type
- shape – Dimension of the placeholder. Optional.
- name – Name of the placeholder. Optional

The following example shows the use of a placeholder to feed input data into a linear equation.

```
x = tf.placeholder(float)
y = 2*x + 1
sess = tf.Session()
result = sess.run(y, feed_dict = {x:[0, 5, 10, 15, 20]})
```

array([1., 11., 21., 31., 41.], dtype=float32)

A placeholder can hold tensors of any shape. For instance, a 2D tensor can be fed in place of the 1D array shown previously.

```
x = tf.placeholder(float,[None,3])
y = 2*x + 1
data = [[1, 2, 3], [10, 20, 30]]
sess = tf.Session()
result = sess.run(y, feed_dict = {x:data})
array([[ 3.,  5.,  7.],
       [21., 41., 61.]], dtype=float32)
```

The output is a 2x3 matrix, so the None used in the definition of the placeholder, could be safely replaced by 2 in the code above, yielding the same result.

```
x = tf.placeholder(float,[2,3])
y = 2*x + 1
data = [[1, 2, 3], [10, 20, 30]]
sess = tf.Session()
```

```
result = sess.run(y, feed_dict = {x: data})
array([[ 3.,  5.,  7.],
       [21., 41., 61.]], dtype=float32)
```

However, the previous syntax is preferred if the first dimension of the variable changes at different circumstances. For example, when using the syntax,

```
x = tf.placeholder(float,[None,3])
```

Matrices with dimension 1x3, 2x3, 3x3, 4x3, etc can be input to the placeholder with no error. However, by fixing the two dimensions, as in,

```
x = tf.placeholder(float,[2,3])
```

The placeholder will only accept a matrix with dimension 2x3. Therefore it gives less flexibility to the placeholder. However this may be necessary if one wants to avoid certain errors that may occur by leaving the first dimension open.

15.7. WAYS OF CREATING TENSORS

Tensors can be generated using different functions in TensorFlow. Each function generates a tensor with certain characteristics, such as random numbers, set of zeros, ones, repeating sequence, etc.

Consider the case of creating a tensor from an image. In essence, an image consists in 3 matrices, also called RGB (red, green and blue channels). Therefore, images can be conceptualized as tensors of rank 3, where the first two dimensions are the number of rows and columns of each matrix (number of pixels at each direction) and the last dimension is the number of channels. For example, an image with dimension,

```
(100,80,3)
```

Consists in a picture 100 pixels tall, 80 pixels large and with three channels (always). An image can be read into tensor by using the method `tf.image.decode_jpeg` (when the image consists in a jpeg file).

```
image = tf.image.decode_jpeg(tf.read_file("cat.jpg"), channels = 3)
sess = tf.Session()
sess.run(image.shape)
[210,150,3]
```

A tensor can be generated as a set of zeros by using the `tf.zeros(size)` function.

```
x = tf.zeros([3, 2])
sess = tf.Session()
sess.run(x)
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

It may be a set of ones by using the `tf.ones(size)` function.

```
x = tf.ones([3, 2])
sess = tf.Session()
sess.run(x)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

A tensor may be filled with a unique number through the `tf.fill(size,value)` function, where value is the number (or object) used to fill each element of the tensor.

```
x = tf.fill([3, 2], 5)
sess = tf.Session()
sess.run(x)
[[5 5]
 [5 5]
 [5 5]]
```

The method `tf.diag(elements)` creates a matrix with diagonal elements equal to elements.

```
x = tf.diag([1, 2, 3])
sess = tf.Session()
sess.run(x)
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

To create a tensor consists in a sequence of numbers, use the `tf.range(start,limit,delta)` function, where start is the initial value, limit is the

last value (exclusive) and delta is the step size, i.e., the difference between each value.

```
x = tf.range(start = 3, limit = 30, delta = 3)
sess = tf.Session()
sess.run(x)
array([ 3,  6,  9, 12, 15, 18, 21, 24, 27], dtype=int32)
```

For a sequence of evenly spaced values, use `tf.linspace(start,stop,n)` where n is the number of points to be added in the sequence.

```
h = tf.linspace(3, 30, 5)
sess = tf.Session()
sess.run(h)
array([ 3. ,  9.75, 16.5, 23.25, 30. ], dtype=float32)
```

A random sequence of values according the uniform distribution can be constructed using the `tf.random_uniform(value,minval,maxval)`, where minval is the minimum value of the distribution and maxval is the maximum value. The dimension is given in value.

```
r = tf.random_uniform([3,2], minval = 0, maxval = 3)
sess = tf.Session()
sess.run(r)
array([[1.0970614, 1.5497539 ],
       [2.7323077, 0.7938584 ],
       [0.24447012, 0.46585643]], dtype=float32)
```

To obtain a sequence of normally distributed values, using the `tf.random_normal(value,mean,stddev)`, where the dimension is given by value, the mean and the standard deviation are the given by mean and stddev respectively.

```
r = tf.random_normal([3,2], mean = 2, stddev = 1)
sess = tf.Session()
sess.run(r)
array([[1.1963735, 1.0302888],
       [1.3054788, 1.1521074],
       [1.1329939, 2.3905213]], dtype=float32)
```

15.8. SUMMARY

The structure of TensorFlow library is based on three entities:

- *Graph*: Environment encapsulating operations (nodes) and tensors (edges)
- *Tensors*: Data (or value) stored in the edge of the graph. It flows along the graph passing through operations.
- *Sessions*: Environment that executes the operations.

Table 15.1 shows example on how to create constant tensors.

Table 15.1: Constant Tensors

Dimension	Example
0	<code>tf.constant(1, tf.int16)</code>
1	<code>tf.constant([1, 2, 3], tf.int16)</code>
2	<code>tf.constant([[1, 2, 3], [4, 5, 6]], tf.int16)</code>
3	<code>tf.constant([[[1, 2], [3, 4], [4, 5]]], tf.int16)</code>

Table 15.2: TensorFlow Operators

Operation	Example
$a + b$	<code>tf.add(a, b)</code>
$a - b$	<code>tf.subtract(a, b)</code>
$a \times b$	<code>tf.multiply(a, b)</code>
$\frac{a}{b}$	<code>tf.div(a, b)</code>
a^b	<code>tf.pow(a, b)</code>
e^a	<code>tf.exp(a)</code>
\sqrt{a}	<code>tf.sqrt(a)</code>

Table 15.3: TensorFlow Variables

Random initial value	<code>tf.get_variable("var", [2, 3])</code>
Initialized value	<code>tf.get_variable("var", [2, 3], dtype = tf.int32, initializer = tf.zeros([2,3]))</code>

Table 15.4: Operations on a Session

Description	Code
Create a session	<code>tf.Session()</code>
Run session	<code>tf.Session().run()</code>
Evaluate	<code>variable_name.eval()</code>
Close session	<code>tf.Session().close()</code>
Session in block	<code>with tf.Session() as sess:</code>

16

TensorFlow: Activation Functions and Optimization

CONTENTS

16.1. Activation Functions	268
16.2. Loss Functions	273
16.3. Optimizers.....	274
16.4. Metrics	277

In the last chapter, the basics of TensorFlow elements were presented, and many examples showed how TensorFlow works. In this chapter, we will discuss Activation Functions, what are them, and which types are already implemented in TensorFlow. We will also discuss Loss Function, Optimizers, and Metrics. By the end of this chapter, all these concepts should be clear to the reader.

In summary, the following concepts are discussed along this chapter:

- Activation functions;
- Loss functions;
- Optimizers;
- Metrics.

16.1. ACTIVATION FUNCTIONS

Activation function derives from the concept of how human brain neurons work. When a chemical sign arrives at the neuron, it may become active beyond a certain threshold, which is also called the activation potential. By becoming active it propagates a signal to other neurons, which have a similar behavior. In mathematical neural networks, there are different activation functions:

- Sigmoid;
- Hyperbolic tangent (\tanh);
- ReLU;
- Softplus; +
- Variants of ReLU.

These are the most popular ones.

16.1.1. Hyperbolic Tangent Activation Function

As the name indicates, this activation function consists in the transcendent function $\tanh(x)$. A visual output of this function can be seen in Figure 16.1.

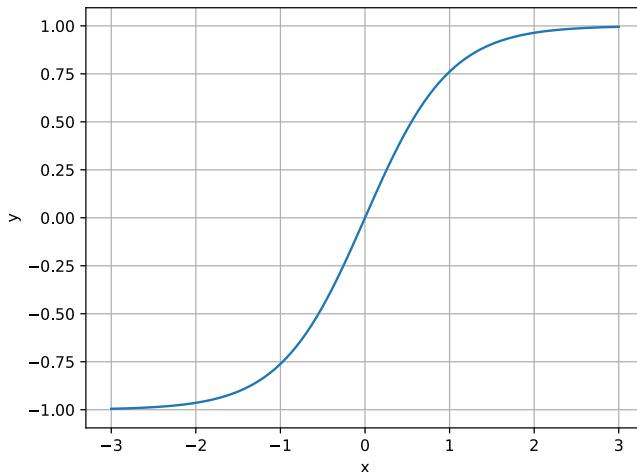


Figure 16.1: Hyperbolic tangent activation function.

Note that its output varies from -1 to $+1$ any value of x (input).

Using the activation function in TensorFlow is done through the function `tf.tanh()` as demonstrated in the following code.

```
tanh = tf.nn.tanh(0.5)  
sess = tf.Session.run(tanh)  
0.46211717
```

16.1.2. Sigmoid Activation Function

The sigmoid function consists in the following mathematical expression.

$$y = \frac{1}{1 + e^{-x}}$$

A plot of the above function can be seen in Figure 16.2.

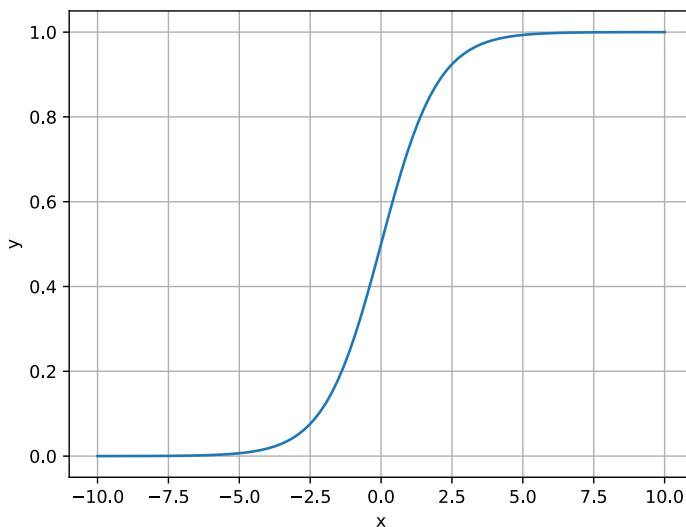


Figure 16.2: Sigmoid activation function.

The sigmoid function is implemented in TensorFlow with the function `tf.nn.sigmoid()`. An example code is demonstrated below.

```
tanh = tf.nn.sigmoid(0.5)  
sess = tf.Session.run(tanh)  
0.62245935
```

16.1.3. ReLU Activation Function

ReLU stands for Rectifier Linear Unit, defined as the positive part of its argument.

$$f(x) = x^+ = \max(0, x)$$

where x is an input and $f(x)$ is an output. This function is also known as **ramp**. A visual output of this function is shown in Figure 16.3.

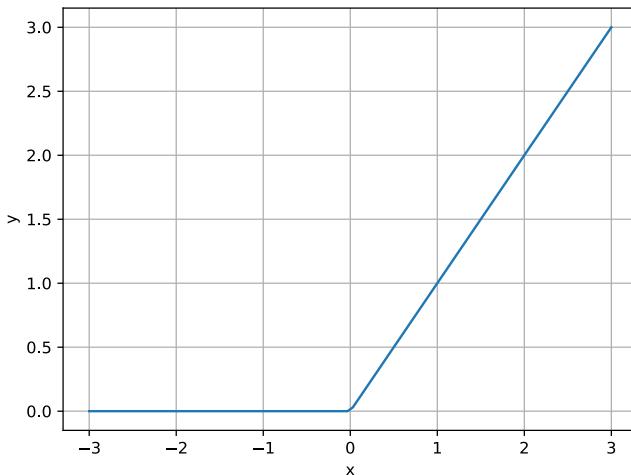


Figure 16.3: ReLU activation function.

As it can be seen above, it consists in a horizontal line where $y = 0$ when $x \leq 0$, and a linear function with slope equals 1 when $x > 0$.

This function is implemented in TensorFlow using the `tf.nn.relu()` function.

```
relu = tf.nn.relu(0.5)
```

```
sess = tf.Session()
```

```
sess.run(relu)
```

```
0.5
```

16.1.4. Softplus Activation Function

Softplus consists in a smooth approximation of ReLU.

$$f(x) = \log(1 + e^x)$$

As may be noted in Figure 16.4, the difference is that the hard edge in point $(0,0)$ is smoothed, thus been differentiable at this point (ReLU is not).

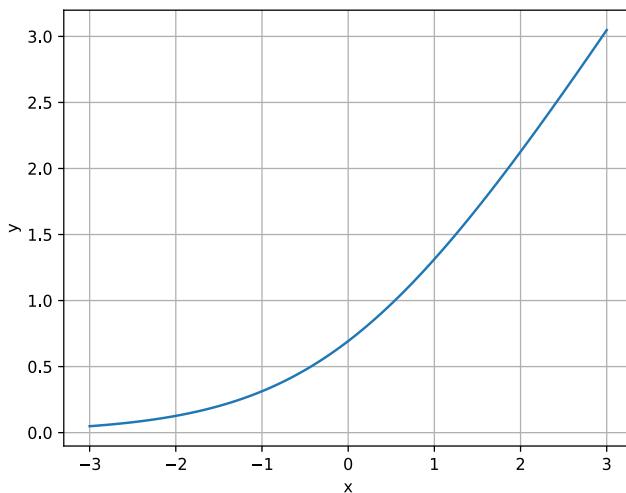


Figure 16.4: Softplus activation function.

The following code shows how to use the Softplus function in TensorFlow.

```
softplus = tf.nn.softplus(0.5)
sess = tf.Session()
sess.run(softplus)
0.974077
```

16.1.5. ReLU Variants

ELU stands for Exponential linear units. It has an advantage over RELU since it tries to make the mean activations closer to zero which accelerates the learning process.

If $x > 0$,

$$f(x) = x$$

Otherwise,

$$f(x) = a(e^x - 1)$$

where a is a tunable hyperparameter, constrained at $a \geq 0$

```

vec1 = tf.constant([-3., -1., 0., 1., 3.])
elu = tf.nn.elu(vec1)
sess = tf.Session()
sess.run(elu)
array([-0.95021296, -0.63212055, 0. , 1. , 3. ], dtype=float32)

```

Leaky ReLU

A small modification of ReLU, allowing a small gradient when $x < 0$, i.e., when the neuron is inactive.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

```

vec1 = tf.constant([-3., -1., 0., 1., 3.])
elu = tf.nn.leaky_relu(vec1)
sess = tf.Session()
sess.run(elu)
array([-0.03, -0.01, 0. , 1. , 3. ], dtype=float32)

```

16.2. LOSS FUNCTIONS

A loss function is a way of measuring how good (or bad) is a model by giving it a cost. Therefore, the loss function is also called a cost function. When calibrating a model to obtain the best set of parameters according a dataset, the cost function is the value to be minimized. For example, in linear regression, the loss function is usually the sum of squared residuals/errors (SSE) or the mean squared errors (MSE). In essence, minimization of the loss function is the driving force to obtain the optimum set of parameters of a certain model. For regression, L1 (sum of absolute residuals) or L2 (sum of squared residuals) is usually a good choice, while for classification softmax or sigmoid cross entropy is usually a valid choice.

Some common loss functions used in TensorFlow are:

- `tf.contrib.losses.absolute_difference`
- `tf.contrib.losses.add_loss`
- `tf.contrib.losses.hinge_loss`

- `tf.contrib.losses.compute_weighted_loss`
- `tf.contrib.losses.cosine_distance`
- `tf.contrib.losses.get_losses`
- `tf.contrib.losses.get_regularization_losses`
- `tf.contrib.losses.get_total_loss`
- `tf.contrib.losses.log_loss`
- `tf.contrib.losses.mean_pairwise_squared_error`
- `tf.contrib.losses.mean_squared_error`
- `tf.contrib.losses.sigmoid_cross_entropy`
- `tf.contrib.losses.softmax_cross_entropy`
- `tf.contrib.losses.sparse_softmax_cross_entropy`
- `tf.contrib.losses.log`

16.3. OPTIMIZERS

Once the loss function is known, it is necessary to use an algorithm to find the value of the parameters of the model which minimizes it. This algorithm is the optimizer. In most algorithms, an initial guess on the parameters are given, and the optimizer evaluates at each iteration the direction to change each parameter (increasing or decreasing) so as to reduce the loss function. After many iterations, the values of the parameters start to stabilize around the optimum values.

As it occurs with other deep learning frameworks, TensorFlow provides a variety of optimizers that changes the values of the parameters in a model to minimize the loss function. Actually the purpose of the optimizer is to find the direction to which change the parameters so to become nearer to the minimum. Choosing the best optimizer and the amount of change in the variables is not direct and may rely on heuristic rules.

According Manaswi (2018), adaptive techniques such as adadelta and adagrad are able to perform faster convergence of complex neural networks. In general, Adam is the best optimizer, which motivates it to be the default one in certain frameworks, such as keras. This optimizer outperforms other adaptive techniques though being computationally costly. When the dataset is big, containing many sparse matrices, SGD, NAG and momentum may not be the best options. In such cases, the adaptive learning rate methods are the best choice.

As an example, consider the task of finding the solution to the system of linear equations shown below.

$$3x_1 + 2x_2 - x_3 = 1$$

$$2x_1 - 2x_2 + 4x_3 = -2$$

$$-x_1 + 1/2x_2 - x_3 = 0$$

The following code demonstrates how to solve this system using TensorFlow and Adam optimizer.

First define the matrix A containing all the coefficients of the system of equations above.

```
A = tf.constant([[3, 2, -1], [2, -2, 4], [-1, 0.5, -1]])
```

Then create the TensorFlow variable x which will be the one to be determined through optimization.

```
x = tf.Variable(initial_value = tf.zeros([3, 1]), name = "x," dtype = tf.float32 )
```

Define the matrix multiplication Ax by creating a node on the TensorFlow graph which perform such operation.

```
model = tf.matmul(A, x)
```

Create the result vector, consisting of the values in the right-hand side of the system of equations.

```
result = tf.constant([[1.], [-2.], [0.]])
```

Once the result is defined, implement the loss function, which calculates the mean squared difference between the expected result and the one obtained through the matrix multiplication.

```
loss = tf.losses.mean_squared_error(result, model)
```

Create an instance of the optimizer using Adam algorithm (AdamOptimizer) with a learning rate of 1.5.

```
optimizer = tf.train.AdamOptimizer(1.5)
```

Tell the optimizer that the task is to minimize the loss variable.

```
train = optimizer.minimize(loss)
```

In the rest of the code, create the session and run it a couple of iterations until convergence of x is observed. The complete code of the above example is shown below.

```
A = tf.constant([[3, 2, -1], [2, -2, 4], [-1, 0.5, -1]])
x = tf.Variable(initial_value = tf.zeros([3, 1]), name = "x," dtype = tf.float32)
model = tf.matmul(A, x)
result = tf.constant([[1.], [-2.], [0.]])
loss = tf.losses.mean_squared_error(result, model)

optimizer = tf.train.AdamOptimizer(1.5)
train = optimizer.minimize(loss)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print('Initial value of x:', sess.run(x), ' loss:', sess.run(loss))
for step in range(1000):
    sess.run(train)
    if step % 100 == 0:
        xval = sess.run(x).flatten()
        print('step ', step, 'x1: ', round(xval[0], 3), 'x2: ', round(xval[1], 3),
              'x3: ', round(xval[2], 3), ' loss: ', sess.run(loss))
step 0 x1: -1.5 x2: 1.5 x3: -1.5 loss: 38.354153
step 100 x1: 0.375 x2: -0.492 x3: -0.915 loss: 0.03166012
step 200 x1: 0.741 x2: -1.38 x3: -1.551 loss: 0.0053522396
step 300 x1: 0.923 x2: -1.816 x3: -1.867 loss: 0.00047268326
step 400 x1: 0.983 x2: -1.96 x3: -1.971 loss: 2.2427732e-05
step 500 x1: 0.997 x2: -1.994 x3: -1.995 loss: 5.682331e-07
step 600 x1: 1.0 x2: -1.999 x3: -1.999 loss: 7.4286013e-09
step 700 x1: 1.0 x2: -2.0 x3: -2.0 loss: 4.7748472e-11
step 800 x1: 1.0 x2: -2.0 x3: -2.0 loss: 7.058058e-13
step 900 x1: 1.0 x2: -2.0 x3: -2.0 loss: 1.3310834e-12
```

Some common optimizers in TensorFlow are (all from `tf.train`):

- GradientDescentOptimizer
- AdadeltaOptimizer
- AdagradOptimizer

- AdagradDAOptimizer
- MomentumOptimizer
- AdamOptimizer
- FtrlOptimizer
- ProximalGradientDescentOptimizer
- ProximalAdagradOptimizer
- RMSPropoptimizer

16.4. METRICS

Metrics provides a mathematical value to measure model accuracy. TensorFlow contains different metrics, such as accuracy, logarithmic loss, area under curve (AUC), ROC, among others.

Some common metrics available in TensorFlow are (from `tf.contrib.metrics`):

- streaming_root_mean_squared_error
- streaming_covariance
- streaming_person_correlation
- streaming_mean_cosine_distance
- streaming_percentage_less
- streaming_false_negatives

17

Introduction to Natural Language Processing

CONTENTS

17.1. Definition Of Natural Language Processing	280
17.2. Usage Of NLP.....	280
17.3. Obstacles In NLP	281
17.4. Techniques Used In NLP	281
17.5. NLP Libraries	282
17.6. Programming Exercise: Subject/Topic Extraction Using NLP.....	282
17.7. Text Tokenize Using NLTK.....	286
17.8. Synonyms From Wordnet.....	288
17.9. Stemming Words With NLTK.....	290
17.10. Lemmatization Using NLTK	291

In essence, Natural Language Processing (NLP) is a technology used to transform human's natural language into something understandable for machines. Though it may appear simple, it is actually a complex task, when one thinks in terms of different languages worldwide, sentence structures, grammar, etc. This chapter provides a simple introduction to NLP and how it works.

17.1. DEFINITION OF NATURAL LANGUAGE PROCESSING

A field of artificial intelligence, NLP provides algorithms to aid the interaction between computers and humans using natural language. The objective of the algorithms is to read, decipher, understand and extract sense in a valuable manner.

The majority of techniques used in NLP derive from machine learning.

The following steps illustrate a general human-machine interaction using natural language.

Step 1: A human talks to the machine;

Step 2. The audio is captured by the machine;

Step 3. Machine converts the audio to text;

Step 4. The text is processed;

Step 5: The machine output data is converted to audio;

Step 6. Audio output simulates that machine is talking.

17.2. USAGE OF NLP

The following are some common applications of NLP:

- Language translation services (Google Translate, Bing Translate);
- Grammar checking and correction in word processors (Microsoft Word, Grammarly);
- Interactive Voice Response (IVR) services (e.g., call centers);
- Personal Assistant applications (OK Google, Siri, Cortana)

17.3. OBSTACLES IN NLP

The nature of human language presents a natural difficulty for Natural Language Processing. This is due the fact that human language is composed by complex structures. It may contain emotional traits, such as sarcasm to pass certain information. Besides that, there are literally thousands of human languages worldwide, and each language requires a different treatment.

The rules used to pass information to machines may have different level of difficulties and abstraction. For instance, a sarcastic remark to be understandable requires a high-level rule with elevate abstraction. On the other hand, the meaning of item plurality can be easily interpreted in many languages by the addition of an “s” at the end of the word items.

According Garbade (2018), a comprehensive understanding of the human language is only possible by understanding the linking of words and concepts, and how such links are used to deliver the message.

17.4. TECHNIQUES USED IN NLP

The two main techniques used in NLP are **syntactic** and **semantic** analysis.

Syntactic analysis refers to the way words are arranged in a sentence so that it is grammatically correct and it makes sense. In NLP, this concept is used to retrieve the agreement of natural language with grammatical rules. To do so, algorithms apply grammatical rules to group of words and obtain meaning from them.

The following is a list of syntax techniques that are currently used:

- **Lemmatization:** Reduction of different word inflections to a unique form for easy analysis.
- **Morphological segmentation:** Division of words into individual units (morphemes).
- **Word segmentation:** Division of large continuous text into smaller units.
- **Part-of-speech (POS) tagging:** Identification of a tag for each word (such as verb, adjective, etc.).
- **Parsing:** Sentence grammatical analysis.
- **Sentence boundary disambiguation:** Also known as Sentence breaking, consists in detecting boundaries on text.
- **Stemming:** Reduction of inflected words to root form.

Semantic analysis is used to retrieve the meaning of a text. Being one of the most difficult aspects of NLP, it still has not been fully resolved. In machine learning, it involved using algorithms that extract meaning of words and the structure of sentences.

The following is a list of semantic techniques that are currently used:

- **Named entity recognition (NER)**: identification and grouping of words according its meaning. Examples are names of people and names of places.
- **Word sense disambiguation**: Retrieve meaning of word based on the context. One example in English is the word “spirit,” which may indicate alcoholic drink, or supernatural entity.
- **Natural language generation**: Express semantic intentions into human language. Requires a database to store all the relevant semantic intentions that the machine may express.

17.5. NLP LIBRARIES

There is currently a great variety of Natural Language Processing libraries. Some of the most popular are (Ebrahim (2017)):

- Natural language toolkit (NLTK) in Python;
- Apache OpenNLP;
- Stanford NLP suite;
- Gate NLP library.

From the ones above, NLTK is by far the most popular one. According Ebrahim (2017), it is easy to learn and use, being probably the easiest NLP library.

17.6. PROGRAMMING EXERCISE: SUBJECT/TOPIC EXTRACTION USING NLP

In this exercise, NLTK library is used to retrieve the topic/subject of a certain text. In summary this simple algorithm consists in, after preprocessing the text, extract the words with highest frequency and assumed these words refer to the main topic of the page. For instance, for a text about World War II, we would expect to retrieve the words “war,” “conflict” or something similar which means that the text talks about war.

To start, import nltk in Python terminal.

```
import nltk
```

Step 1: Tokenize text

The topic of a Wikipedia article will be used to show how this algorithm works. It is chosen one from Linux (<https://en.wikipedia.org/wiki/Linux>), so it is expected the algorithm will return as a topic the word Linux or at least something relatable.

The module `urllib.request` is used to read the web page and obtain the pure html content.

```
import urllib.request
```

```
response = urllib.request.urlopen('https://en.wikipedia.org/wiki/Linux')
```

```
html = response.read()
```

```
print(html)
```

```
b'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n<meta charset="UTF-8"/>\n<title>Linux – Wikipedia</title>\n...'
```

Notice the `b` character at the beginning of the html output. It indicates that this is a binary text. To clean HTML tags and to a general cleaning of the raw text we use BeautifulSoup library, as shown in the code below.

```
from bs4 import BeautifulSoup\nsoup = BeautifulSoup(html, "html5lib")\ntext = soup.get_text(strip = True)\nprint (text[:100])
```

```
Linux – Wikipediadocument.documentElement.className=document.documentElement.className.replace(/(^|\r\n|^ |^ )/g, '')
```

Convert the text into tokens by using `split()` method in Python, which splits a string into whitespaces. Notice that up to this point `nltk` is not being used.

```
tokens = [t for t in text.split()]\nprint (tokens)
```

```
[‘Linux’, ‘-’, ‘Wikipediadocument.documentElement.className=document.documentElement.className.replace(/(^|\s)client-nojs(\s|$)/,”$1client-js$2”);...]
```

With the words already separated, one can use the FreqDist() function from nltk library to count the frequency of words in a list.

```
freq = nltk.FreqDist(tokens)
```

We create a table using pandas library to visualize the frequency of words. For better readability, we can sort the frequencies descending (highest to lowest) using sort_values(..., ascending = False) functionality from pandas Dataframe. The code is used to print the 20 words with highest frequencies in the text.

```
import pandas as pd  
df = pd.DataFrame.from_dict(freq,orient = ‘index’)  
df.sort_values(by = 0,ascending = False).head(20)
```

the	465
of	269
and	219
Linux	197
on	193
to	166
a	159
for	113
original	110
in	100
is	99
with	71
as	66
operating	57
from	57
system	52
software	51
that	51
distributions	50
also	48

Notice that, understandably, words which are not very meaningful but important to create links in sentences are the ones with highest frequencies. It would be necessary to look on the above table to understand further in its topic due to the presence of many so-called “stop words,” i.e., words that are important to create links and bring sense, but not relevant to extract the topic of the text. Fortunately, nltk library has a functionality to remove such “stop words.” To use it, first it is necessary to download the set of “stop words” from the language under interest (English in this case).

```
from nltk.corpus import stopwords
nltk.download('stopwords')
stopwords.words('english')[:10]
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

The list above shows only the first 10 stop words stored in the nltk stopwords functionality. We use this to remove such words from the original text, so they won’t appear in the frequency count table.

```
clean_tokens = tokens[:]
```

for token **in** tokens:

```
if token in stopwords.words('english'):
```

```
    clean_tokens.remove(token)
```

With the new list of tokens without stop words (clean_tokens), build again the frequency table and visualize the words which appear with highest frequency in the text.

```
freq = nltk.FreqDist(clean_tokens)
df = pd.DataFrame.from_dict(freq,orient='index')
df.sort_values(by=0,ascending=False).head(20)
```

Linux	197
original	110

operating	57
system	52
software	51
distributions	50
also	48
fromthe	42
Archived	41
originalon	40
use	32
used	31
kernel	29
July	27
RetrievedJune	26
desktop	25
GNU	24
December	24
distribution	24
September	23

Notice now how the word “Linux” has appeared at first place. Also, the next words are “original,” “operating” and “system” which tells us some information about what is the article about. That is a very simple, if not the simplest NLP algorithm one can use.

17.7. TEXT TOKENIZE USING NLTK

Tokenization means splitting a text or a sentence into smaller parts. A text cannot be processed without this step. In the exercise above the text was divided into tokens using the split built-in Python function. In this part you will see how NLTK can be used for that task.

Paragraphs can be tokenize to sentences and sentences tokenize into words depending on the problem. In any case, NLTK contains both functionalities (sentence tokenizer and word tokenizer).

Let a text to be analyzed as below.

“Good morning, Alex, how are you? I hope everything is well. Tomorrow will be a nice day, see you buddy.”

The above text can be tokenized into sentences using sentence_tokenizer functionality from nltk.

```
import nltk  
nltk.download('punkt')  
  
from nltk.tokenize import sent_tokenize
```

mytext = “Good morning, Alex, how are you? I hope everything is well. Tomorrow will be a nice day, see you buddy.”

```
print(sent_tokenize(mytext))  
[‘Good morning, Alex, how are you?’, ‘I hope everything is well.’,  
 ‘Tomorrow will be a nice day, see you buddy.’]
```

To use this algorithm, it is necessary to download the PunktSentenceTokenizer which is part of the nltk.tokenize.punkt module. This is done through the command: nltk.download('punkt').

The whole text, as a single string, is divided into sentences by a recognition of the dots in nltk. Notice how the following text, which incorporates the tile “Mr.” (with a dot) is still correctly divided.

```
import nltk  
nltk.download('punkt')  
  
from nltk.tokenize import sent_tokenize
```

mytext = “Good morning, Mr. Alex, how are you? I hope everything is well. Tomorrow will be a nice day.”

```
print(sent_tokenize(mytext))  
[‘Good morning, Mr. Alex, how are you?’, ‘I hope everything is well.’,  
 ‘Tomorrow will be a nice day.’]
```

Similarly, words can be tokenized by using the word_tokenize functionality from nltk library. Notice the result as this is applied to the sentence presented above.

```
import nlt
```

```
nltk.download('punkt')

from nltk.tokenize import word_tokenize
```

mytext = "Good morning, Mr. Alex, how are you? I hope everything is well. Tomorrow will be a nice day."

```
print(word_tokenize(mytext))
['Good', 'morning', ',', 'Mr.', 'Alex', ',', 'how', 'are', 'you', '?', 'I', 'hope',
 'everything', 'is', 'well', '.', 'Tomorrow', 'will', 'be', 'a', 'nice', 'day', '.']
```

Notice how this algorithm recognizes that the word "Mr." contains the dot at the end, thus not removing it.

NLTK library does not work only with English natural language. In the following example, sentence tokenizer is used to split a sentence in Portuguese.

```
import nltk
nltk.download('punkt')
```

```
from nltk.tokenize import sent_tokenize
```

mytext = "Bom dia, Sr. Alex, como o senhor está? Espero que esteja bem. Amanhã será um ótimo dia."

```
print(sent_tokenize(mytext))
['Bom dia, Sr. Alex, como o senhor está?', 'Espero que esteja bem.', 'Amanhã será um ótimo dia.']}
```

NLTK is able to automatically recognize the language being used in the example above. This is evident since it does not split at the word "Sr." (meaning Mr. in English).

17.8. SYNONYMS FROM WORDNET

According Ebrahim (2017), WordNet consists in a database created and maintained exclusively for natural language processing. It contains sets of

synonyms with their definitions. The following code shows an example of extracting synonyms of a word using WordNet.

```
from nltk.corpus import wordnet
```

```
syn = wordnet.synsets("patient")
```

```
print(syn[0].definition())
```

```
print(syn[0].examples())
```

The output of the above code is,

a person who requires medical care

[‘the number of emergency patients has grown rapidly’]

To obtain synonyms from a words using WordNet, one can use the synsets(word_to_be_inquired) function from wordnet module and obtain each lemma, as shown in the code below.

```
from nltk.corpus import wordnet
```

```
synonyms = []
```

```
for syn in wordnet.synsets('Car'):
```

```
    for lemma in syn.lemmas():
```

```
        synonyms.append(lemma.name())
```

```
print(synonyms)
```

[‘car’, ‘auto’, ‘automobile’, ‘machine’, ‘motorcar’, ‘car’, ‘railcar’, ‘railway_car’, ‘railroad_car’, ‘car’, ‘gondola’, ‘car’, ‘elevator_car’, ‘cable_car’, ‘car’]

In a similar way, antonyms can be retrieved by doing a slight modification of the code above.

```
from nltk.corpus import wordnet
```

```
antonyms = []
```

```
for syn in wordnet.synsets("beautiful"):
```

```
    for l in syn.lemmas():
```

```
        if l.antonyms():
```

```
            antonyms.append(l.antonyms()[0].name())
```

```
print(antonyms)
```

```
['ugly']
```

17.9. STEMMING WORDS WITH NLTK

Stemming consists in obtaining the root from a word containing affixes. For example, the stem of building is build. This is a common technique used by search engines when indexing pages. In this way, even when people write different versions of the same word, all of them are stemmed, thus converging to the same root word.

Among the different algorithms available for stemming, Porter stemming algorithm is one of the most used. NLTK incorporates such algorithm in the class PorterStemmer, as shown in the code below.

```
from nltk.stem import PorterStemmer
```

```
print(PorterStemmer().stem('building'))
```

The result is,

build

Other stemming algorithm worth mentioning is the Lancaster stemming algorithm. There are slightly different results from both algorithms in different words.

NLTK also supports stemming of languages other than English, using the SnowballStemmer class. The supported languages can be visualized by checking the SnowballStemmer.languages property.

```
from nltk.stem import SnowballStemmer
```

```
print(SnowballStemmer.languages)
('arabic', 'danish', 'dutch', 'english', 'finnish', 'french', 'german',
'hungarian', 'italian', 'norwegian', 'porter', 'portuguese', 'romanian',
'russian', 'spanish', 'swedish')
```

The code below shows an example usage of SnowballStemmer to stem word from a non-English language, for example Portuguese.

```
from nltk.stem import SnowballStemmer
portuguese_stemmer = SnowballStemmer('portuguese')

print(portuguese_stemmer.stem("trabalhando"))
trabalh
```

17.10. LEMMATIZATION USING NLTK

Similar to stemming, lemmatization returns the root of a word. However, the difference is, while stemming may return not always return a real word as root, lemmatization returns always a real word. The default word is noun, but different part of speeches can be returned, by using the pos argument when using the lemmatize function of WordNetLemmatizer as shown in the following example.

```
import nltk
nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize('panting', pos = "v")) # return a verb
paint
```

Notice that a word with 7 letters is reduced to 5 letters, a compression of almost 30%. In many cases, the overall text compression can reach 50 to 60 % by using lemmatization.

Other than a verb (pos = v), the lemmatization result can be noun (pos = n), adjective (pos = a) or adverb (pos = r)

18

Project: Recognize Handwritten Digits Using Neural Networks

CONTENTS

18.1. Introduction.....	294
18.2. Project Setup	294
18.3. The Data	294
18.4. The Algorithm	294

This example is a modified version of the one presented by Birbeck (2019).

18.1. INTRODUCTION

Developments in the field of machine learning have allowed models to exceed capabilities that are commonly attributed only to living-beings or even humans. One of such capabilities is object recognition. Though still inferior to human vision, machines are able to recognize objects, faces text, and even emotions through neural network frameworks.

In this chapter, we implement a simple digit recognition algorithm using TensorFlow library and Python programming language. The algorithm is able to recognize hand-draw images from 0 (zero) to 9 (nine) through classification of the hand-written digit.

18.2. PROJECT SETUP

Before developing the algorithm, there are some remarks of the project structure which are relevant.

The complete code can be written in a Jupyter Notebook or using Google Collab Notebook. The second one requires a Google account but no installation of Python in the local machine. The first one (Jupyter) requires a complete installation of python, jupyter and the libraries numpy and TensorFlow.

A better understanding of this chapter can be reached by first going through the chapters describing Neural Networks and TensorFlow along this book.

18.3. THE DATA

The dataset consists in a set of images of handwritten digits with size 28x28 pixels. This dataset is known as the MNIST dataset, and is a classical one of machine learning community.

18.4. THE ALGORITHM

The algorithm to deal with this set is implemented in a Jupyter Notebook. First it is necessary to import the TensorFlow library, as well as the dataset

from mnist package in TensorFlow.

```
import tensorflow as tf
```

```
mnist = tf.keras.datasets.mnist
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>

```
11493376/11490434 [=====] - 0s 0us/step
```

Verify some dimensions of the data for a better understanding of it.

```
n_train = train_images.shape[0] # 60,000
```

```
n_test = test_images.shape[0] # 10,000
```

```
print('Size of images: ', (train_images.shape[1], train_images.shape[2]))
```

```
print('Number of training samples: ', n_train)
```

```
print('Number of testing samples: ', n_test)
```

```
print('Example of labels: ', train_labels[:10])
```

Size of images: (28, 28)

Number of training samples: 60000

Number of testing samples: 10000

Example of labels: [5 0 4 1 9 2 1 3 1 4]

The data imported consists in the images interpreted as matrices (28x28) in the variable train_images and test_images, as well as the digit values in train_labels and test_labels

Before any modeling, it is important to preprocess the data in a way that it can be used by the neural network that will be developed along this chapter.

The neural network accepts each pixel of the image as an input. An image with 28x28 pixels contains 784 pixels in total. Therefore, the number of inputs that the neural network accepts should be 784. That also means that the image matrix must be flatten into a vector, by concatenating each subsequent row at the end of the previous one.

Also, the values stored in the input matrices ranges from 0 to 255, meaning color intensity. For better performance of the neural network, it is necessary to perform normalization of such values so they fit in the range 0–1.

The following code is used to do matrix flattening (matrix to vector) and normalization of the values in the input matrices.

```
# Matrix -> vector transformation
```

```
train_images = train_images.reshape(train_images.shape[0], -1)
```

```
# Matrix normalization
```

```
train_images = (train_images - np.min(train_images))/(np.max(train_images) - np.min(train_images))
```

Regarding the labels (expected output of the network), each digit is a label or class. To perform classification using the neural network, each output neuron should produce a value between 0 or 1, indicating the probability of that image being a certain digit. The following illustrates a pseudo calculation of probabilities from the neural network.

Image 1 → probability of “0” = 0.4 Image 1 → probability of “1” = 0.6
 Image 1 → probability of “2” = 0.55

Where each probability is given by an output neuron. Therefore, the general structure of the neural network is according Figure 18.1.

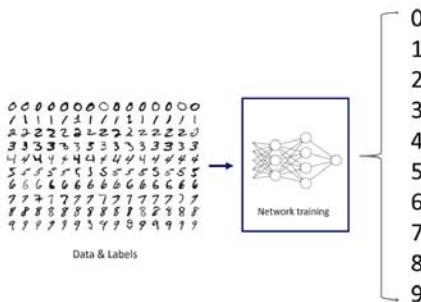


Figure 18.1: General neural network structure for MNIST dataset problem.

To transform the labels stored in `train_labels` and `test_labels` into probabilities, one can one-hot encode it, as it is done in the following code.

```
import numpy as np
```

```
train_labels_encode = np.zeros((train_labels.shape[0], len(set(train_labels))))
```

```
train_labels_encode[np.arange(train_labels.shape[0]), train_labels] = 1
test_labels_encode = np.zeros((test_labels.shape[0], len(set(test_labels))))
test_labels_encode[np.arange(test_labels.shape[0]), test_labels] = 1
```

Define some constant to build the neural network.

```
n_input = 784 # input layer (28x28 pixels)
n_hidden1 = 128 # 1st hidden layer
n_hidden2 = 64 # 2nd hidden layer
n_hidden3 = 32 # 3rd hidden layer
n_output = 10 # output layer (0–9 digits)
```

The first hidden layer contains 128 neurons, with subsequent layers containing 64, 32, and 10 neurons respectively. Notice that, from the above values, only n_input and n_output are fixed for this problem. The others can be “freely” modified, though excessive neurons may cause overfitting and too few neurons may lead to underfitting.

Besides the structure of the network, define the parameters for the training process. These are the learning rate, the number of iterations, the batch size, and dropout value.

```
learning_rate = 1e-3
n_iterations = 1000
batch_size = 128
dropout = 0.5
```

The structure of the neural network is built using placeholders for the input and output vectors, and variables for the weights and biases.

```
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_output])
keep_prob = tf.placeholder(tf.float32)
```

```
weights = [tf.Variable(tf.truncated_normal([n_input, n_hidden1],
                                         stddev=0.1)),
           tf.Variable(tf.truncated_normal([n_hidden1, n_hidden2], stddev=0.1)),
           tf.Variable(tf.truncated_normal([n_hidden2, n_hidden3], stddev=0.1)),
           tf.Variable(tf.truncated_normal([n_hidden3, n_output], stddev=0.1))]
```

```
biases = [tf.Variable(tf.constant(0.1, shape=[n_hidden1])),  
          tf.Variable(tf.constant(0.1, shape=[n_hidden2])),  
          tf.Variable(tf.constant(0.1, shape=[n_hidden3])),  
          tf.Variable(tf.constant(0.1, shape=[n_output]))]
```

Each hidden layer neuron in the neural network uses the sigmoid activation function. A drop layer is added before the output layer, which indicates a chance dropout of a value being eliminated by chance before being sent to the output layer. This helps to avoid overfitting. The code to structure the network is as follows.

```
layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(X, weights[0]), biases[0]))  
layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights[1]), biases[1]))  
layer_3 = tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights[2]), biases[2]))  
layer_drop = tf.nn.dropout(layer_3, rate = 1 - keep_prob)  
output_layer = tf.add(tf.matmul(layer_3, weights[3]), biases[3])
```

The training is done using the gradient descent method. Among the several optimizers available for this algorithm, it is used Adam, which is also the default one. One advantage of such algorithm is that it can use momentum to accelerate the training process by computing an exponentially weighted average of the gradients (Birbeck (2019)).

```
cross_entropy = tf.reduce_mean(  
    tf.nn.softmax_cross_entropy_with_logits_v2(  
        labels=Y, logits = output_layer  
    ))  
train_step      =      tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

Each training step, the network should evaluate how “good” it is. This value also helps to see when the training should be stopped due to the convergence of the performance. This is done by evaluating the accuracy as shown in the following code.

```
correct_pred = tf.equal(tf.argmax(output_layer, 1), tf.argmax(Y, 1))  
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

With everything configured, initialize the session and perform the training of the neural network. Due to the number of data points, at each iteration a subset of it is chosen and the optimization is processed. After modification of the parameters, then check the accuracy and print it to screen to observe how the training process evolves.

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

complete_idx = np.arange(train_images.shape[0])
# train on mini batches
for i in range(n_iterations):

    batch_idx = np.random.choice(complete_idx,batch_size,replace = False)

    batch_x, batch_y = train_images[batch_idx],train_labels_encode[batch_idx]

    sess.run(train_step, feed_dict={
        X: batch_x, Y: batch_y, keep_prob: dropout
    })

    # print loss and accuracy (per minibatch)
    if i % 100 == 0:

        minibatch_loss, minibatch_accuracy = sess.run(
            [cross_entropy, accuracy],
            feed_dict={X: batch_x, Y: batch_y, keep_prob: 1.0}
        )

        print("Iteration,"
```

```
    str(i),  
    "\t| Loss =,"  
    str(minibatch_loss),  
    "\t| Accuracy =,"  
    str(minibatch_accuracy)  
)
```

```
Iteration 0 | Loss = 2.3467183 | Accuracy = 0.109375  
Iteration 100 | Loss = 1.8820465 | Accuracy = 0.375  
Iteration 200 | Loss = 1.2935321 | Accuracy = 0.671875  
Iteration 300 | Loss = 1.0126756 | Accuracy = 0.6953125  
Iteration 400 | Loss = 0.7320422 | Accuracy = 0.84375  
Iteration 500 | Loss = 0.69052655 | Accuracy = 0.859375  
Iteration 600 | Loss = 0.5260591 | Accuracy = 0.8671875  
Iteration 700 | Loss = 0.3844887 | Accuracy = 0.9296875  
Iteration 800 | Loss = 0.4345752 | Accuracy = 0.90625  
Iteration 900 | Loss = 0.46503332 | Accuracy = 0.890625
```

The accuracy should increase with optimization progress, as it can be seen above. A bad choice of the learning rate may cause it to oscillate and even diverge. The loss decreases, with perfect loss being equal to 0, while perfect accuracy is 1.

With the trained model, one can evaluate the accuracy of it on the test dataset.

```
test_accuracy = sess.run(accuracy, feed_dict={X: test_images.reshape(-1,n_input), Y: test_labels_encode, keep_prob: 1.0})  
print("\nAccuracy on test set:", test_accuracy)  
0.9284
```

Notice that an accuracy of almost 93% is achieved using the configuration above. Actually, state-of-the-art algorithms have already achieved approximately 98% accuracy for this dataset. Still being able to correctly predict 9 out of 10 images can be considered a high performance, especially given the relatively simple neural network structure used in this example.

Appendix A

A.1. Housing Price Dataset

Square size (ft)	Price (USD)
648	582
548	425
1069	825
681	488
912	631
425	364
675	441
615	589
417	492
1147	711
728	542
829	590
976	647
848	
376	388
781	633
778	639
211	276
211	348
349	386
1183	740

428	316
962	730
686	617
506	492
518	518
527	423
1070	746
1113	777
902	737
685	540
315	290
558	478
755	512
744	538
817	582
844	541
725	625
499	361
284	374
272	365
1199	705
950	740
483	480
1146	752
223	324
636	420
284	308
845	605
499	437
1049	704
1191	871
964	671
1150	776
1138	813
416	331
513	372

889	613
243	410
457	426
434	411
406	423
654	433
971	721
571	488
1168	709
1076	739
721	638
278	386
962	586
563	395
216	275
647	576
839	608
1145	809
1026	754
743	502
502	357
252	287
916	696
214	350
692	518
248	332
462	372
972	630
799	631
1035	738
636	482
1159	748
481	369
435	—
319	450
1139	821

1190	707
867	682
222	387
572	562
278	290
751	608
580	519

A.2. Linear Regression using LLS (Linear Least Squares) Method

Perform Linear Regression with Native Python and Linear Least Squares Method

```
x = [] # Square size
y = [] # Housing Price
with open('housing_data.csv', 'r') as file:
    # read header
    file.readline()

    # loop over the file, filling the lists with x and y vectors
    for line in file:
        row = line.split(',')
        x.append(float(row[0]))
        y.append(float(row[1]))

def average(x):
    return sum(x) / len(x)

avg_x = average(x)
avg_y = average(y)

a1_num = sum([(xi-avg_x)*(yi-avg_y) for (xi,yi) in zip(x,y)])
a1_den = sum([(xi-avg_x)**2 for xi in x])
a1 = a1_num/a1_den

a0 = avg_y - a1*avg_x
```

```
print('Parameter a0 = ',a0)
```

```
print('Parameter a1 = ',a1)
```

```
""
```

Determine R2 (Coefficient of determination).

A value between 0 and 1, where 1 means perfect model, and 0 is a model with low accuracy.

```
""
```

```
sqres = sum([(yi - (a1*xi+a0))**2 for (xi,yi) in zip(x,y)])
```

```
sqtot = sum([(yi - avg_y)**2 for yi in y])
```

```
R2 = 1 - sqres/sqtot
```

```
print('Coefficient of Determination R2 = ',R2)
```

```
""
```

The final Cost value is the Sum of Squared Residues, obtained in variable sqres

```
""
```

```
J = sqres
```

```
print('Final Cost J = {:.3e}'.format(J))
```

A.3. Programming Exercise: Linear Regression with Single Input and Multiple Inputs

example1.py

Chapter 13 Programming Exercise: Linear Regression with Single Input

#%#-----Part 0: Importing Libraries-----

```
import numpy as np
```

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

```
from matplotlib.backends.backend_pdf import PdfPages
```

```
#%% -----Part 1: Data Generation and Plotting-----
np.random.seed(999)
X = np.stack((np.ones((100,)),np.random.exponential(5,100))).T
y = X.dot([-4,1.2])+ np.random.rand(100)*10
with PdfPages('example1_fig1.pdf') as pdf:
    plt.figure()
    plt.plot(X[:,1],y,'b.')
    plt.xlabel('population (Thousands)')
    plt.ylabel('profit (1000 USD)')
    plt.grid()
    pdf.savefig()
    plt.show()

#%% -----Part 2: Cost Function-----
def computeCost(X,y,a):
    """
    J = computeCost(X,y,theta):
        Computes the cost for linear regression using the parameters a
        to fit points in X and y dataset
    """
    J = 1/(2*len(y))*np.sum((X.dot(a) - y)**2)

    return J

#%% -----Part 3: Gradient Descent-----
def gradientDescent(X,y,a,alpha,MAX_ITER,verbose=False):
    """
    gradientDescent(X,y,theta,alpha,MAX_ITER)
        Computer the optimum a values (parameters) using GD algorithm.
    Inputs:
    
```

X – input vector (1D numpy array)

y – output vector (1D numpy array)

a – initial guess of parameters (2x1 numpy array)

alpha – learning rate (float)

MAX_ITER – maximum number of iterations (int)

verbose – True to print the iteration process, False otherwise

Returns:

a – optimum parameters

..

iter_ = 0 # initial iteration step

J_history = [] # a list to store the cost at each iteration

J_history.append(computeCost(X,y,a)) # store the cost at each iteration

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4s}\t{a[0]:4s}\t{a[1]:4s}\t{J_history[:4s]}'")

while iter_ < MAX_ITER:

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4d}\t{a[0]:4.2f}\t{a[1]:4.2f}\t{J_history[-1]:4.2f}')

*dJda0 = np.sum((a[0] + a[1]*X[:,1]-y)) # partial derivative of J / a0*

*dJda1 = np.sum((a[0] + a[1]*X[:,1]-y)*X[:,1]) # partial derivative of J / a1*

J += 1

update the-parameters

*a[0] = a[0] – alpha*dJda0*

*a[1] = a[1] – alpha*dJda1*

```
    iter_ += 1
    J_history.append(computeCost(X,y,a)) # store the cost at each iteration

return a,J_history
```

#%% -----Part 4: Solution using GD -----

```
a = np.zeros((2,)) # initial gues on parameters
```

```
# GD parameters
MAX_ITER = 1500
ALPHA = 0.0002
```

```
# run GD
a, J_history = gradientDescent(X,y,a,ALPHA,MAX_ITER,True)
```

```
# print final result to screen
print('Final parameters')
print(f'a0 = {a[0]:.2f}')
print(f'a1 = {a[1]:.2f}')
```

```
# plot the data and the linear fit
with PdfPages('example1_fig2.pdf') as pdf:
    plt.figure()
    plt.plot(X[:,1],y,'b.')
    plt.plot(X[:,1],X.dot(a),'r')
    plt.xlabel('population (Thousands)')
    plt.ylabel('profit (1000 USD)')
    plt.legend(['Observed','Predicted'])
    plt.grid()
    pdf.savefig()
```

```
plt.show()
```

#%# -----Part 5: Perform predictions -----

```
# predict the profit for a population of 5000  
yhat = np.array([1,5]).dot(a)
```

```
print(f'For a population of 5000, the predicted profit is {yhat:.2f} Thousands  
USD.)'
```

example1_multi.py

*## Chapter 13 Programming Exercise: Linear Regression with Multiple
Input*

#%#-----Part 0: Importing Libraries-----

```
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend_pdf import PdfPages
```

#%# -----Part 1: Data Generation and Plotting-----

```
np.random.seed(999)  
X = np.stack((np.ones((100,)),np.random.exponential(5,100),np.random.  
exponential(30,100))).T  
y = X.dot([-7,-1.2,3])+ np.random.rand(100)*100
```

```
with PdfPages('example1_multi_X1.pdf') as pdf:  
    plt.figure()  
    plt.hist(X[:,1])  
    plt.xlabel('population (1000)')  
    plt.ylabel('Frequency')  
    plt.grid()  
    pdf.savefig()
```

```
plt.show()
```

```
with PdfPages('example1_multi_X2.pdf') as pdf:  
    plt.figure()  
    plt.hist(X[:,2])  
    plt.xlabel('Income per capita')  
    plt.ylabel('Frequency')  
    plt.grid()  
    pdf.savefig()  
    plt.show()
```

```
with PdfPages('example1_multi_fig1.pdf') as pdf:  
    plt.figure()  
    plt.plot(np.sort(y), 'b.')  
    plt.xlabel('data point')  
    plt.ylabel('profit (1000 USD)')  
    plt.grid()  
    pdf.savefig()  
    plt.show()
```

#%% -----Part 2: Cost Function-----

```
def computeCost(X,y,a):  
    """
```

J = computeCost(X,y,theta):

*Computes the cost for linear regression using the parameters a
to fit points in X and y dataset*
""

```
J = 1/(2*len(y))*np.sum((X.dot(a) - y)**2)
```

```
return J
```

#%% -----Part 3: Gradient Descent-----

```
def gradientDescent(X,y,a,alpha,MAX_ITER,verbose=False):
```

```
""
```

gradientDescent(X,y,theta,alpha,MAX_ITER)

Computer the optimum a values (parameters) using GD algorithm.

Inputs:

X – input vector (1D numpy array)

y – output vector (1D numpy array)

a – initial guess of parameters (2x1 numpy array)

alpha – learning rate (float)

MAX_ITER – maximum number of iterations (int)

verbose – True to print the iteration process, False otherwise

Returns:

a – optimum parameters

```
""
```

iter_ = 0 # initial iteration step

J_history = [] # a list to store the cost at each iteration

J_history.append(computeCost(X,y,a)) # store the cost at each iteration

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4s}\t\t{J_:4s}')

*J_old = computeCost(X,y,a)*1000*

while iter_ < MAX_ITER **and** abs(J_old-J_history[-1])>1e-5:

print the iteration process if verbose is True

if verbose == True:

print(f'{iter_:4d}\t\t{J_history[-1]:4.2f}')

```
dJda = np.zeros((X.shape[1],))  
for i in range(X.shape[1]):  
    dJda[i] = np.sum((X.dot(a)-y)*X[:,i])
```

update the-parameters

```
a -= alpha*dJda
```

```
iter_ += 1
```

```
J_old = J_history[-1]
```

J_history.append(computeCost(X,y,a)) # store the cost at each iteration

```
return a,J_history
```

%%% -----Part 4: Solution using GD -----

```
a = np.zeros((X.shape[1],)) # initial guess on parameters
```

GD parameters

```
MAX_ITER = 1500
```

```
ALPHA = 0.002
```

feature normalization

```
X[:,1:] = (X[:,1:]-np.min(X[:,1:],axis=0))/(np.max(X[:,1:],axis=0)-np.  
min(X[:,1:],axis=0))
```

run GD

```
a, J_history = gradientDescent(X,y,a,ALPHA,MAX_ITER,True)
```

print final result to screen

```
print('Final parameters')
print('a = ',a)

# plot the data and the linear fit
with PdfPages('example1_multi_fig2.pdf') as pdf:
    plt.figure()
    plt.plot(np.sort(y), 'b.')
    plt.plot(np.sort(X.dot(a)), 'r')
    plt.xlabel('data point')
    plt.ylabel('profit (1000 USD)')
    plt.legend(['Observed', 'Predicted'])
    plt.grid()
    pdf.savefig()
    plt.show()
```

#%#% -----Part 5: Perform predictions -----

```
# predict the profit for a normalized input vector x = [1,0.9,0.4]
yhat = np.array([1,0.9,0.4]).dot(a)
```

```
print(f'For a population of 5000, the predicted profit is {yhat:.2f} Thousands
      USD.')
```

#%#% -----Part 6: Normal Equation -----

```
a = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
print('Parameters obtained using Normal Equation')
print('a = ',a)
```

A.4. Programming Exercise: Logistic Regression in Python

logistic_simple.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from time import time

class LogisticRegression:
    def __init__(self, learning_rate=0.02, max_iterations=1000, fit_intercept=True):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.fit_intercept = fit_intercept

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def __loss(self, h, y):
        return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

    def fit(self, X, y, verbose=False):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        # weights initialization
        self.theta = np.zeros(X.shape[1])
```

```
i = 0
while i < self.max_iterations:
    z = np.dot(X, self.theta)
    h = self.__sigmoid(z)
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.learning_rate * gradient

    if(verbose == True and i % 100 == 0):
        z = np.dot(X, self.theta)
        h = self.__sigmoid(z)
        print(f'loss: {self.__loss(h, y)} \t')
    i += 1

def predict_prob(self, X):
    if self.fit_intercept:
        X = self.__add_intercept(X)

    return self.__sigmoid(np.dot(X, self.theta))

def predict(self, X, threshold):

    return self.predict_prob(X) >= threshold

if __name__ == '__main__':
    # hours of study
    X = np.array([0.45, 0.75, 1.00, 1.25, 1.75, 1.50, 1.30, 2.00, 2.30, 2.50,
2.80, 3.00, 3.30,
3.50, 4.00, 4.30, 4.50, 4.80, 5.00, 6.00]).reshape(-1, 1)
```

```
# results (0 – fail; 1 – pass)
y = np.array([0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1])

with PdfPages('logistic_simple_data.pdf') as pdf:
    plt.scatter(X[y > 0.5], y[y > 0.5], color = 'blue', label = 'Positive')
    plt.scatter(X[y < 0.5], y[y < 0.5], color = 'red', label = 'Negative')
    plt.xlabel('Hours of study')
    plt.ylabel('Result')
    plt.title('Plot of dataset')
    plt.legend()
    pdf.savefig()
    plt.show()

model = LogisticRegression(learning_rate = 0.1, max_iterations =
1000, verbose = True)

start = time()
model.fit(X, y, verbose = True)
end = time()
print('time taken = ' + str(round(end - start, 2)) + ' s')

preds = model.predict(X, 0.51)
# accuracy
print('(preds == y).mean() = ', (preds == y).mean())
print('theta = ', model.theta)

with PdfPages('logistic_simple_pred.pdf') as pdf:
    plt.scatter(X[y > 0.5], y[y > 0.5], color = 'blue', label = 'Positive')
    plt.scatter(X[y < 0.5], y[y < 0.5], color = 'red', label = 'Negative')
    plt.plot(X, preds, '-k', label = 'Predicted')
    plt.xlabel('Hours of study')
```

```
plt.ylabel('Result')
plt.title('Plot of dataset')
plt.legend()
pdf.savefig()
plt.show()
```

```
exit = input('Press enter to exit...')
```

logistic_keras.py

```
#-----Library Import -----
```

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

```
#-----Data Import/Generate/Process -----
```

```
# hours of study
```

```
X = np.array([0.45, 0.75, 1.00, 1.25, 1.75, 1.50, 1.30, 2.00, 2.30, 2.50, 2.80,
3.00, 3.30,
3.50, 4.00, 4.30, 4.50, 4.80, 5.00, 6.00]).reshape(-1, 1)
```

```
# results (0 – fail; 1 – pass)
```

```
y = np.array([0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1]).reshape([-1,
1])
```

```
#-----Model Configuration -----
```

```
model = keras.models.Sequential()
```

```
model.add(keras.layers.Dense(units=1, input_dim=1, activation='sigmoid'))
```

```
sgd = keras.optimizers.SGD(lr = 0.5)
```

```
model.compile(optimizer = sgd, loss = 'binary_crossentropy', metrics =
['accuracy'])
```

#----*Model Training* -----

```
history = model.fit(X, y, epochs = 1000, verbose = 0)
```

#----*Model Prediction* -----

```
ypred = model.predict(X).reshape(-1, 1)
```

```
plt.plot(X, ypred)
```

```
plt.plot(X, y, 'o')
```

Bibliography

1. Birbeck, Ellie. (2019). “How To Build a Neural Network to Recognize Handwritten Digits with TensorFlow | DigitalOcean.” <https://www.digitalocean.com/community/tutorials/how-to-build-a-neural-network-to-recognize-handwritten-digits-with-tensorflow>.
2. Ebrahim, Mokhtar. (2017). “NLP Tutorial Using Python NLTK (Simple Examples) – Like Geeks.” <https://likegeeks.com/nlp-tutorial-using-python-nltk/>.
3. Garbade, Michael J. (2018). “A Simple Introduction to Natural Language Processing.” <https://becominghuman.ai/a-simple-introduction-to-natural-language-processing-ea66a1747b32>.
4. Getzmann, S., Jasny, J., Falkenstein, M. (2016). “Switching of auditory attention in “cocktail-party” listening: ERP evidence of cueing effects in younger and older adults.” *Brain and Cognition.*, no. 111:1–12.
5. Harnad, Stevan. (2008). “The Annotation Game: On Turing 1950 on Computing, Machinery, and Intelligence.” *The Turing Test Sourcebook: Philosophical and Methodological Issues in the Quest for the Thinking Computer*.
6. Ioffe, Sergey, & Christian Szegedy. (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” February. <http://arxiv.org/abs/1502.03167>.
7. Johansson, R. (2014). “Introduction to scientific computing in Python.” [github. com/jrjohansson/scientific-python-lectures](https://github.com/jrjohansson/scientific-python-lectures).
8. Langtangen, Hans Peter. (2011). *A Primer on Scientific Computing with Python*.
9. Manaswi, Navin Kumar. (2018). *Deep Learning with Applications Using Python*. <https://doi.org/10.1007/978-1-4842-3516-4>.

10. Mitchell, T. (1997). *Machine Learning*. McGraw Hill.
11. Patel, P. (2018). “Why Python is the most popular language used for Machine Learning.” <https://medium.com/@UdacityINDIA/why-use-python-for-machine-learning-e4b0b4457a77>.
12. PennState. (n.d.). “Simple Linear Regression Examples.” <https://newonlinecourses.science.psu.edu/stat462/node/101/>.
13. Simpson, Andrew, J. R., Gerard Roma, & Mark D. Plumbley. (2015). “Deep karaoke: Extracting vocals from musical mixtures using a convolutional deep neural network.” In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-22482-4_50.
14. Smith, Lindsay A. (2002). “A Tutorial on Principal Components Analysis.” http://www.cs.otago.ac.nz/cosc453/student{_}tutorials/principal{_}components.pdf.
15. Susan Li. (2017). “Building A Logistic Regression in Python, Step by Step.” <https://towardsdatascience.com/building-a-logistic-regression-in-python-step-by-step-becd4d56c9c8>.
16. The Jupyter Notebook. (2019). “What Is the Jupyter Notebook? — Jupyter/IPython Notebook Quick Start Guide 0.1 documentation.” https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what{_}is{_}jupyter.html.
17. The MathWorks, Inc. (2019). “Machine Learning with MATLAB.” <https://www.mathworks.com/solutions/machine-learning.html>.
18. Trevino, Andrea. (2016). “Introduction to K-Means Clustering.” <https://www.datascience.com/blog/k-means-clustering>.
19. Yegulalp, Serdar. (2019). “11 Open-Source Tools to Make the Most of Machine Learning.” <https://www.infoworld.com/article/2853707/machine-learning/11-open-source-tools-machine-learning.html{\#}slide3>.

INDEX

A

Absolute minimum 85
Accuracy metrics 146
Activation function 174, 175, 176,
 178, 268
Activation Function 267, 268
Activation potential 268
Activity Segmentation 246
Algebraic manipulation 157
Algorithm 58, 60
Algorithm implementation 105
Algorithm progress 142
Algorithm recognizes 288
Algorithms apply grammatical 281
Application Programming Interface
 (APIs) 254
Appropriate value 94, 106
Approximate minimum 84, 85
Approximation 82, 83, 91

Artificial Intelligence 4
Assumption 156
Automatic hyperparameter 5
Automatic memory management
 system 3
Auxiliary variable 41
Avoid confusion 156

C

Calibration 134, 185
Client request submission 54
Coefficient 157, 159, 170
Column 132, 133, 134
Column vector 115, 128
Community integrates 5
Complex cancer analysis 70
Complexity 174
Complex matrix operation 114
Comprehension 89, 91

Computational capacity 78

Computer memory 3

Computer program 62

Constructing 65

Consultancy 52

Cost function 84, 85, 87, 90, 91, 92, 93, 96, 99, 100, 101, 102, 103, 104, 108, 109, 110, 111, 135, 138, 142, 143, 150

Covariance equal 224

Covariance matrix 225, 226, 230, 232

Covariance value 224

D

Data Mining 63

Data-structure 28

Deallocate memory 3

Decision tree 196

Didactic algorithm 78

Dimension 223, 225, 226, 227, 228, 230, 231, 232

Dimensional variable 260

Dynamically-type language 2

E

Eigenvector 220, 227, 230, 231, 232

Element 25, 26, 27, 28, 30, 31, 32, 35, 40, 46

English natural language 288

Entropy 214, 215

Environment manager 9, 10

Experimental data 239

F

Function 26, 30, 32, 44, 45, 46, 47, 48, 49

Functional programming approach 242

G

Generic line 66

Generic neural network 180

Global namespace 48, 256

Google Assistant 254

Google research 255

Gradient descent algorithm 143, 145, 170, 171, 172

Gradient Descent (GD) 91

Graphical procedure 91

Graphical representations 227

Graphical visualization 228

Group inventory 246

H

Horizontal concatenation operator 122

Human brain neurons work 268

Human language 281, 282

Human neural system 174

Human voice recognition 73

Hypothetical database 78

Hypothetical dataset 140

I

Implementation 166

Independent variable 66

Initialization function 176

Initial values 260

Inner dimension 118, 119, 121, 123

Inner structure 235

Instruction 47, 53

Intensive computational task 74

Interactive Voice Response (IVR) 280

International community 3, 4

Internet documentation 59
 Inverse operation 231
 Iteration process 143, 144, 145,
 150, 151
 Iterative approach 137
 Iterative process 150

K

k-Nearest Neighbor (kNN) 234

L

Languages worldwide 280, 281
 Language translation services 280
 Library ecosystem 4
 Linear algebra 114, 226, 228
 Linear least squares (LLS) 68
 Linear regression 78, 79, 80, 82, 85,
 88, 90, 92, 94, 96, 97, 98, 99,
 108, 109, 111, 112
 Linear regression model 132, 134,
 135, 138, 140, 141
 Linear regression problem 141
 Linear relationship 156
 Linear system 68
 Linear transformation 174
 Logic requires comparison 36
 Logistic regression 156, 158, 159,
 160, 161, 162, 165, 172
 Loss function 273, 274, 275

M

Machine Learning 4, 5, 6, 7, 10, 12,
 15, 17, 51, 52, 56, 58, 60
 Machine learning algorithm 7, 8
 Machine learning computation 6
 Machine learning system 68
 Magnitude 161
 Manual selection 170

Mapping associate 28
 Mathematical neural network 268
 Matrix 28, 114, 115, 116, 117, 118,
 119, 120, 121, 122, 123, 124,
 125, 126, 128, 129

Matrix element 115

Matrix inversion 153

Matrix multiplication 138, 149

Maximum value 136

Mean squared errors (MSE) 273

Minimum value 264

Min-max normalization 135, 136

Min-max scaling method 135

Misclassification 213

Mixed data 205

Model generalization 169

Model selection algorithm 170

Model structure 80

Modular approach 166

Momentum 274

Morphological segmentation 281

Multiple Measurement 207

Multiple variable 132

Multiplication 117, 118, 120, 121,
 122, 123, 125, 129

N

Named entity recognition (NER)
 282

Namespace collision 15

Natural decision process 196

Natural Language Processing 279,
 280, 281, 282

Negligible 161

Neighbors-based method 234

Neural network 8, 9, 74

Normal Equation approach 171

Normal equation method 137

Numerical Python 4
Numerous implementation 160

O

Object-oriented approach 176
Object-Oriented Programming approach 162
One-dimensional dataset 231
Optimal value 80, 87
Optimization process 99, 109, 110
Optimum found 83
Optimum neural network 185
Optimum parameter 137, 144, 150
Optimum values 274
Optional implementation 180
Original data 227, 228, 231
Original dataset 229, 231
Original impurity 201

P

Parameter 81, 82, 83, 84, 88, 89, 92, 93, 96, 100, 101, 102, 104, 105
Parameter reduction 170
Partygoer 73
Pass information 281
Personal development system 4
Popular function 175
Portuguese 288, 291
Positive correlation 224
Possible measure 214
Predictive system 75
Presumption 86
Principal Component Analysis 219, 220, 227
Probability 156, 157, 158, 159, 161, 162
Procedural programming 166
Produces unique splits 202

Programming community 5
Programming language 2, 3, 4, 6, 7, 11, 16, 53, 57
Prototyping 6, 7
Python 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19
Python object 26, 28
Python programming language 22, 96, 115
Python syntax 57

Q

Quadratic 168, 169, 170
Qualitative examination 246
Quantifying accuracy 214

R

Random forest 196, 205, 206
Random value 161
Real minimum 82, 85
Rectangular 114
Regression 66, 70, 71, 75, 76
Regularization 170, 171, 172
Regularized linear regression 171
Regular task 73
Reinforcement learning 63
Replacement 206

S

Scalar multiplication 117
Scientific computing 114
Semantic analysis 281, 282
Sigmoid cross entropy 273
Sigmoid function 175, 176, 178, 186, 193
Sigmoid transformation 176
Simple algorithm 234
Single column 115

Single element 133
 Single-Value Decomposition (SVD) 72
 Smooth approximation 271
 Social interchange 5
 Solution 134, 137, 138, 139, 152, 153
 Spectrogram 73, 74
 Splitting quality 201
 Squared Method 88
 Squared residuals/errors (SSE) 273
 Square root 221
 Standard deviation 220
 Statistical analysis 220
 Statistical measurement 223
 Stochastic Gradient Descent (SGD) 188
 Sum of squared errors (SSE) 100
 Symmetrical 225
 Synonym 4
 Syntax techniques 281

T

Target variable 80
 Tensor 255, 256, 259, 260, 261, 262, 263
 Tokenization 286
 Training data 60, 170
 Transcendent function 268
 Transformation 227
 Transpose operation 133

U

Unique instrument 73
 Unit variance 136

V

Variable 132, 133, 135, 136, 140
 Vector 114, 118, 122
 Vector operation 114, 121
 Verbose argument 145
 Visualization 97, 101

W

WinPython Package Manager (WPPM) 11

Fundamentals of Machine Learning using Python

Machine learning is a field of study that has drastically grown in the last decades. Whenever one enters the Internet and uses a search engine or an automatic translation service, machine learning is present. Modern surveillance systems are equipped with systems, which can detect and classify objects using machine-learning techniques. When predicting stock prices or general market features, economists use this same technique to have accurate results. These are just to mention a few applications that machine learning has found in different areas. This book is intended for under-graduate students and enthusiasts who are interested in starting to develop their own machine learning models and analysis. It provides a simple but concise introduction to the main knowledge streams that are necessary to start into this area, even entering a bit beyond the basics with an introduction of TensorFlow, a state-of-the-art framework to develop machine learning models.

Since most applications are currently being developed in Python, the authors chose to focus on this programming language. The first two chapters provide a general introduction to machine learning and Python programming language. Chapter 3 introduces how a machine-learning model can be developed from zero using a story-based description.

On Chapter 4, main concepts that appear when talking about machine learning are described. The following chapters go deeper into each modeling technique. Chapters 5, 6, and 7 deals with the linear regression model and the concepts on linear algebra to understand more complex techniques. Chapter 8 describes the logistic regression model, the equivalent of linear regression for classification. Chapter 9 introduces the concept of regularization to later present neural networks in Chapters 10 and 11. How Decision Trees works with examples, as well as the development of Random Forests are mentioned in Chapter 12.

Chapter 13 shows how Principal Component Analysis (PCA) can be used to simplify a dataset, reducing its dimensions and enabling one to retrieve information of it. Chapter 14 deals with classification problems by showing how k Nearest Neighbors model works with implementation in Python as the previous models.

Chapters 15 and 16 presents to the reader the state-of-the-art framework TensorFlow, a collection of functionalities which highly enhances the development of machine learning models, with the incorporation of efficient computations and intuitive programming.



Euan Russano was born in Minas Gerais, Brazil. He is a Chemical Engineer since 2012 by the Rural University of Rio de Janeiro (UFRRJ). He obtained his Msc. in 2014 at UFRRJ, in the area of Chemical Engineering, with specialization in Process Control. In 2014, Russano began to develop his PhD at the University Duisburg-Essen (UDE) in the field of Water Science. His career was initiated in a Petrobras project in the Polymers Laboratory, at UFRRJ. From 2012 to 2014 he worked in the Fluids Flow Laboratory (Petrobras/ UFRRJ) with oil well pressure control. Since 2014 he works as an research assistant at the University of Duisburg-Essen, with water systems identification and control.



Elaine Ferreira Avelino was born in Rio de Janeiro, Brazil. She obtained her Bsc in Forestry Engineering at the Rural University of Rio de Janeiro (UFRRJ) in 2007 and Msc. in 2012 at UFRRJ, in the area of Forestry and Environmental Sciences, with specialization in Wood Technology. She started her career in the Secretary for Environment of Rio de Janeiro, with Urban and Environmental Planning. Elaine was a professor of Zoology, Entomology, Forestry Parasitology and Introduction to Research at the Pitagoras University. Since 2013 she works as an international consultant for forest management and environmental licensing.

ISBN 978-1-77407-427-5

A standard 1D barcode representing the ISBN number 978-1-77407-427-5. To the right of the barcode, the numbers "00000" are printed vertically. Below the barcode, the ISBN number is repeated: "9781774074275".