

Machine Learning Theory and Applications

**Hands-on Use Cases with Python
on Classical and Quantum Machines**

Xavier Vasques

WILEY

Machine Learning Theory and Applications

Machine Learning Theory and Applications

Hands-on Use Cases with Python on Classical and Quantum Machines

Xavier Vasques

IBM Technology, Bois-Colombes, France

Laboratoire de Recherche en Neurosciences Cliniques, Montferrier sur lez, France

Ecole Nationale Supérieure de Cognitique Bordeaux, Bordeaux, France

WILEY

Copyright © 2024 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Names: Vasques, Xavier, author.

Title: Machine learning theory and applications : hands-on use cases with Python on classical and quantum machines / Xavier Vasques.

Description: Hoboken, New Jersey : Wiley, [2024] | Includes index.

Identifiers: LCCN 2023039030 (print) | LCCN 2023039031 (ebook) | ISBN 9781394220618 (hardback) | ISBN 9781394220632 (adobe pdf) | ISBN 9781394220625 (epub)

Subjects: LCSH: Machine learning. | Quantum computing. | Python (Computer program language)

Classification: LCC Q325.5 .V37 2024 (print) | LCC Q325.5 (ebook) | DDC 006.3/1-dc23/eng/20231023

LC record available at <https://lccn.loc.gov/2023039030>

LC ebook record available at <https://lccn.loc.gov/2023039031>

Cover image: Wiley

Cover design: © anand purohit/Getty Images

Set in 9.5/12.5pt STIXTwoText by Straive, Pondicherry, India

To my wife Laura and my daughter Elsa

Contents

Foreword *xiii*

Acknowledgments *xv*

General Introduction *xvii*

1 Concepts, Libraries, and Essential Tools in Machine Learning and Deep Learning 1

1.1 Learning Styles for Machine Learning 2

1.1.1 Supervised Learning 2

1.1.1.1 Overfitting and Underfitting 3

1.1.1.2 K-Folds Cross-Validation 4

1.1.1.3 Train/Test Split 4

1.1.1.4 Confusion Matrix 5

1.1.1.5 Loss Functions 7

1.1.2 Unsupervised Learning 9

1.1.3 Semi-Supervised Learning 9

1.1.4 Reinforcement Learning 9

1.2 Essential Python Tools for Machine Learning 9

1.2.1 Data Manipulation with Python 10

1.2.2 Python Machine Learning Libraries 10

1.2.2.1 Scikit-learn 10

1.2.2.2 TensorFlow 10

1.2.2.3 Keras 12

1.2.2.4 PyTorch 12

1.2.3 Jupyter Notebook and JupyterLab 13

1.3 HephAistos for Running Machine Learning on CPUs, GPUs, and QPUs 13

1.3.1 Installation 13

1.3.2 HephAistos Function 15

1.4 Where to Find the Datasets and Code Examples 32

Further Reading 33

2 Feature Engineering Techniques in Machine Learning 35

2.1 Feature Rescaling: Structured Continuous Numeric Data 36

2.1.1 Data Transformation 37

2.1.1.1 StandardScaler 37

2.1.1.2 MinMaxScaler 39

2.1.1.3 MaxAbsScaler 40

2.1.1.4 RobustScaler 40

2.1.1.5 Normalizer: Unit Vector Normalization 42

2.1.1.6 Other Options 43

2.1.1.7 Transformation to Improve Normal Distribution 44

2.1.1.8	Quantile Transformation	48
2.1.2	Example: Rescaling Applied to an SVM Model	50
2.2	Strategies to Work with Categorical (Discrete) Data	57
2.2.1	Ordinal Encoding	59
2.2.2	One-Hot Encoding	61
2.2.3	Label Encoding	62
2.2.4	Helmert Encoding	63
2.2.5	Binary Encoding	64
2.2.6	Frequency Encoding	65
2.2.7	Mean Encoding	66
2.2.8	Sum Encoding	68
2.2.9	Weight of Evidence Encoding	68
2.2.10	Probability Ratio Encoding	70
2.2.11	Hashing Encoding	71
2.2.12	Backward Difference Encoding	72
2.2.13	Leave-One-Out Encoding	73
2.2.14	James-Stein Encoding	74
2.2.15	M-Estimator Encoding	76
2.2.16	Using HephAistos to Encode Categorical Data	77
2.3	Time-Related Features Engineering	77
2.3.1	Date-Related Features	79
2.3.2	Lag Variables	79
2.3.3	Rolling Window Feature	82
2.3.4	Expanding Window Feature	84
2.3.5	Understanding Time Series Data in Context	85
2.4	Handling Missing Values in Machine Learning	88
2.4.1	Row or Column Removal	89
2.4.2	Statistical Imputation: Mean, Median, and Mode	90
2.4.3	Linear Interpolation	91
2.4.4	Multivariate Imputation by Chained Equation Imputation	92
2.4.5	KNN Imputation	93
2.5	Feature Extraction and Selection	97
2.5.1	Feature Extraction	97
2.5.1.1	Principal Component Analysis	98
2.5.1.2	Independent Component Analysis	102
2.5.1.3	Linear Discriminant Analysis	110
2.5.1.4	Locally Linear Embedding	115
2.5.1.5	The <i>t</i> -Distributed Stochastic Neighbor Embedding Technique	123
2.5.1.6	More Manifold Learning Techniques	125
2.5.1.7	Feature Extraction with HephAistos	130
2.5.2	Feature Selection	131
2.5.2.1	Filter Methods	132
2.5.2.2	Wrapper Methods	146
2.5.2.3	Embedded Methods	154
2.5.2.4	Feature Importance Using Graphics Processing Units (GPUs)	167
2.5.2.5	Feature Selection Using HephAistos	168
	Further Reading	170
3	Machine Learning Algorithms	175
3.1	Linear Regression	176
3.1.1	The Math	176
3.1.2	Gradient Descent to Optimize the Cost Function	177

3.1.3	Implementation of Linear Regression	182
3.1.3.1	Univariate Linear Regression	182
3.1.3.2	Multiple Linear Regression: Predicting Water Temperature	185
3.2	Logistic Regression	202
3.2.1	Binary Logistic Regression	202
3.2.1.1	Cost Function	203
3.2.1.2	Gradient Descent	204
3.2.2	Multinomial Logistic Regression	204
3.2.3	Multinomial Logistic Regression Applied to Fashion MNIST	204
3.2.3.1	Logistic Regression with scikit-learn	205
3.2.3.2	Logistic Regression with Keras on TensorFlow	208
3.2.4	Binary Logistic Regression with Keras on TensorFlow	210
3.3	Support Vector Machine	211
3.3.1	Linearly Separable Data	212
3.3.2	Not Fully Linearly Separable Data	214
3.3.3	Nonlinear SVMs	216
3.3.4	SVMs for Regression	217
3.3.5	Application of SVMs	219
3.3.5.1	SVM Using scikit-learn for Classification	220
3.3.5.2	SVM Using scikit-learn for Regression	222
3.4	Artificial Neural Networks	223
3.4.1	Multilayer Perceptron	224
3.4.2	Estimation of the Parameters	225
3.4.2.1	Loss Functions	225
3.4.2.2	Backpropagation: Binary Classification	226
3.4.2.3	Backpropagation: Multi-class Classification	227
3.4.3	Convolutional Neural Networks	230
3.4.4	Recurrent Neural Network	232
3.4.5	Application of MLP Neural Networks	233
3.4.6	Application of RNNs: LST Memory	242
3.4.7	Building a CNN	246
3.5	Many More Algorithms to Explore	249
3.6	Unsupervised Machine Learning Algorithms	251
3.6.1	Clustering	251
3.6.1.1	K-means	253
3.6.1.2	Mini-batch K-means	255
3.6.1.3	Mean Shift	257
3.6.1.4	Affinity Propagation	259
3.6.1.5	Density-based Spatial Clustering of Applications with Noise	262
3.7	Machine Learning Algorithms with HephAistos	264
	References	270
	Further Reading	270
4	Natural Language Processing	273
4.1	Classifying Messages as Spam or Ham	274
4.2	Sentiment Analysis	281
4.3	Bidirectional Encoder Representations from Transformers	286
4.4	BERT's Functionality	287
4.5	Installing and Training BERT for Binary Text Classification Using TensorFlow	288
4.6	Utilizing BERT for Text Summarization	294
4.7	Utilizing BERT for Question Answering	296
	Further Reading	297

5	Machine Learning Algorithms in Quantum Computing	299
5.1	Quantum Machine Learning	303
5.2	Quantum Kernel Machine Learning	306
5.3	Quantum Kernel Training	328
5.4	Pegasos QSVC: Binary Classification	333
5.5	Quantum Neural Networks	337
5.5.1	Binary Classification with EstimatorQNN	338
5.5.2	Classification with a SamplerQNN	343
5.5.3	Classification with Variational Quantum Classifier	348
5.5.4	Regression	351
5.6	Quantum Generative Adversarial Network	352
5.7	Quantum Algorithms with HephAIstos	368
	References	372
	Further Reading	373
6	Machine Learning in Production	375
6.1	Why Use Docker Containers for Machine Learning?	375
6.1.1	First Things First: The Microservices	375
6.1.2	Containerization	376
6.1.3	Docker and Machine Learning: Resolving the “It Works in My Machine” Problem	376
6.1.4	Quick Install and First Use of Docker	377
6.1.4.1	Install Docker	377
6.1.4.2	Using Docker from the Command Line	378
6.1.5	Dockerfile	380
6.1.6	Build and Run a Docker Container for Your Machine Learning Model	381
6.2	Machine Learning Prediction in Real Time Using Docker and Python REST APIs with Flask	389
6.2.1	Flask-RESTful APIs	390
6.2.2	Machine Learning Models	392
6.2.3	Docker Image for the Online Inference	393
6.2.4	Running Docker Online Inference	394
6.3	From DevOps to MLOPS: Integrate Machine Learning Models Using Jenkins and Docker	396
6.3.1	Jenkins Installation	397
6.3.2	Scenario Implementation	399
6.4	Machine Learning with Docker and Kubernetes: Install a Cluster from Scratch	405
6.4.1	Kubernetes Vocabulary	405
6.4.2	Kubernetes Quick Install	406
6.4.3	Install a Kubernetes Cluster	407
6.4.4	Kubernetes: Initialization and Internal Network	410
6.5	Machine Learning with Docker and Kubernetes: Training Models	415
6.5.1	Kubernetes Jobs: Model Training and Batch Inference	415
6.5.2	Create and Prepare the Virtual Machines	415
6.5.3	Kubeadm Installation	415
6.5.4	Create a Kubernetes Cluster	416
6.5.5	Containerize our Python Application that Trains Models	418
6.5.6	Create Configuration Files for Kubernetes	422
6.5.7	Commands to Delete the Cluster	424
6.6	Machine Learning with Docker and Kubernetes: Batch Inference	424
6.6.1	Create Configuration Files for Kubernetes	427
6.7	Machine Learning Prediction in Real Time Using Docker, Python Rest APIs with Flask, and Kubernetes: Online Inference	428
6.7.1	Flask-RESTful APIs	428
6.7.2	Machine Learning Models	431

6.7.3	Docker Image for Online Inference	432
6.7.4	Running Docker Online Inference	433
6.7.5	Create and Prepare the Virtual Machines	434
6.7.6	Kubeadm Installation	434
6.7.7	Create a Kubernetes Cluster	435
6.7.8	Deploying the Containerized Machine Learning Model to Kubernetes	437
6.8	A Machine Learning Application that Deploys to the IBM Cloud Kubernetes Service: Python, Docker, Kubernetes	440
6.8.1	Create Kubernetes Service on IBM Cloud	440
6.8.2	Containerization of a Machine Learning Application	443
6.8.3	Push the Image to the IBM Cloud Registry	446
6.8.4	Deploy the Application to Kubernetes	448
6.9	Red Hat OpenShift to Develop and Deploy Enterprise ML/DL Applications	452
6.9.1	What is OpenShift?	453
6.9.2	What Is the Difference Between OpenShift and Kubernetes?	453
6.9.3	Why Red Hat OpenShift for ML/DL? To Build a Production-Ready ML/DL Environment	454
6.10	Deploying a Machine Learning Model as an API on the Red Hat OpenShift Container Platform: From Source Code in a GitHub Repository with Flask, Scikit-Learn, and Docker	454
6.10.1	Create an OpenShift Cluster Instance	455
6.10.1.1	Deploying an Application from Source Code in a GitHub Repository	457
	Further Reading	463
	Conclusion: The Future of Computing for Data Science?	465
	Index	477

Foreword

The wheels of time turn faster and faster, and as individuals and as human society we all need to adapt and follow. Progress is uncountable in all domains.

Over the last two years, the author has dedicated many hours over weekends and late evenings to provide a volume of reference to serve as a guide for all those who plan to travel through machine learning from scratch, and to use it in elaborated domains where it could make a real difference, for the good of the people, society, and our planet.

The story of the book started with some blog post series that reached many readers, visits, and interactions. This initiative was not a surprise for me, knowing the author's background and following his developments in the fields of science and technology. Almost 20 years passed since I first met Xavier Vasques. He was freshly appointed for a PhD in applied mathematics, but he was still searching for a salient and tangible application of mathematics. Therefore, he switched to a domain where mathematics was applied to neurosciences and medicine. The topic related to deep brain stimulation (DBS), a neurosurgical intervention using electric stimulation to modulate dysfunctional brain networks to alleviate disabling symptoms in neurological and psychiatric conditions. Therapeutic electric field modelling was the topic of his PhD thesis in Neurosciences. He further completed his training by a master's in computer science from The Conservatoire National des Arts et Métiers, and continued his career in IBM where all his skills combined to push technological development further and fill the gap in many domains through fruitful projects. Neurosciences remained one of his main interests. He joined the Ecole Polytechnique Fédérale de Lausanne in Switzerland as researcher and manager of both the Data Analysis and the Brain Atlasing sections for the Blue Brain Project and the Human Brain Project in the Neuroinformatics division. Back at IBM, he is currently Vice-President and CTO of IBM Technology and Research and Development in France.

Throughout his career, Xavier could contemplate the stringent need but also the lack of communication, understanding, and exchanges between mathematics, computer science, and industry to support technological development. Informatics use routines and algorithms but we do not know precisely what lies behind it from the mathematical standpoint. Mathematicians do not master codes and coding. Industry involved in production of hardware does not always master some of the concepts and knowledge from these two domains to favor progress.

The overall intention of this book is to provide a tool for both beginners and advanced users and facilitate translation from theoretical mathematics to coding, from software to hardware, by understanding and mastering machine learning.

The very personal approach with handwriting, "hand-crafted" figures, and "hands on" approach, makes the book even more accessible and friendly.

May this book be an opportunity for many people, and a guidance for understanding and bringing forth, in a constructive and meaningful way, data science solely for the good of mankind in this busy, febrile, and unsteady twenty-first century.

I am writing from the perspective of the clinician who so many times wondered what is a code, an algorithm, supervised and unsupervised machine learning, deep learning, and so forth.

I see already vocations from very young ages, where future geeks (if the term is still accepted by today's youth) will be able to structure their skills and why not push forward the large amount of work, by challenging the author, criticizing and completing the work.

It is always a joy to see somebody achieving. This is the case for this work by Xavier who spared no energy and time to leave the signature of his curriculum but especially that of his engagement and sharing for today's society.

Montpellier, July 2023

*Dr. Laura Cif, MD, PhD
Service de Neurologie, Département des
Neurosciences Cliniques, Lausanne University
Hospital, Lausanne, Switzerland*
*Laboratoire de Recherche en Neurosciences
Cliniques, France*

Acknowledgments

I would like to express my deepest gratitude to my loving wife and daughter, whose unwavering support and understanding have been invaluable throughout the journey of writing this book. Their patience, encouragement, and belief in my abilities have been a constant source of motivation.

To my wife, thank you for your endless love, understanding, and for standing by me during the countless hours spent researching, writing, and editing. Your unwavering support and belief in my work have been a guiding light.

To my dear daughter, thank you for your patience, understanding, and for being a constant source of inspiration. Your enthusiasm for learning and exploring new ideas has fueled my passion for this project.

I am truly grateful for the love, understanding, and encouragement that my wife and daughter have provided. Without them, this book would not have been possible. Their presence in my life has made every step of this journey meaningful and fulfilling.

Thank you, from the bottom of my heart.

Montpellier, July, 2023

Xavier Vasques

General Introduction

The Birth of the Artificial Intelligence Concept

Thomas Hobbes begins his *Leviathan* by saying, “Reason is nothing but reckoning.” This aphorism implies that we could behave like machines. The film *The Matrix*, meanwhile, lets us imagine that we are controlled by an artificial creature in silico. This machine projects into our brains an imaginary, fictional world that we believe to be real. We are therefore deceived by calculations and an electrode piercing the back of our skull. The scenarios abound in our imagination. Fiction suggests to us that one day, it will be easy to replicate our brains, like simple machines, and far from the complexity that we currently imagine. Any mainstream conference on artificial intelligence (AI) routinely shows an image from *The Terminator* or *2001: A Space Odyssey*.

If “reason is nothing but reckoning,” we could find a mathematical equation that simulates our thinking, our consciousness, and our unconsciousness. This thought is not new. Since the dawn of time, humans have constantly sought to reproduce nature. The question of thought, as such, is one of the great questions that humanity has asked itself. What makes Odysseus able to get away with tricks, flair, imagination, and intuition? How do we reflect, reason, argue, demonstrate, predict, invent, adapt, make analogies, induce, deduce, or understand? Is there a model that allows us to approach these things? Throughout our history, we have claimed that a machine cannot calculate like humans or speak, debate, or multitask like humans. Our desire for mechanization over millennia has shown us that machines, tools, and techniques can accomplish these tasks that we had thought were purely human. Does this mean that machines have surpassed humans? We can only acquiesce to a wide range of tasks. Are machines human? No!

Since our species emerged, we have continued to create tools intended to improve our daily lives, increase our comfort, make our tasks less painful, protect us against predators, and discover our world and beyond. These same tools have also turned against us, even though they had not been endowed with any intelligence. Beyond the use as a tool of AI, the quest for the thinking machine can be viewed in a slightly different way. It can be seen as a desire to know who we are, or what we are. It can also be considered as a desire to play God. Since ancient times, philosophers and scientists have been asking these questions and trying to understand and imitate nature in the hope of giving meaning to our being and sometimes to gain control. This imitation involves the creation of simple or complex models more or less approaching reality. So it is with the history of AI. AI comes not only from the history of the evolution of human thought on the body and the nature of the mind through philosophy, myths, or science but also from the technologies that have accompanied us throughout our history, from the pebble used to break shells to the supercolliders used to investigate quantum mechanics. Some historians have found ancient evidence of human interest in artificial creatures, particularly in ancient Egypt, millennia before the coming of Jesus Christ (BCE). Articulated statues, which could be described as automatons, were used during religious ceremonies to depict a tradesperson such as a kneading baker or to represent Anubis or Qebehsenouf as a dog’s head with movable jaws. Even if they are only toys or animated statuettes using screws, levers, or pulleys, we can see a desire to artificially reproduce humans in action. These objects are not capable of moving on their own, but imagination can help. Automatons may become a symbol of our progress and emancipatory promises. These advances have also been an opportunity for humans to question their own humanity. The use of AI has been the subject of many questions and sources of concern about the future of the human species and its dehumanization.

In Greek mythology, robot servants made by the divine blacksmith Hephaestus lay the foundations for this quest for artificial creation. Despite the fact that Hephaestus is a god with deformed, twisted, crippled feet, this master of fire is considered an exceptional craftsman who has created magnificent divine works. A peculiarity of the blacksmith, recounted in the *Iliad*, is his ability to create and animate objects capable of moving on their own and imitating life. He is credited with creating golden servants who assist him in his work and many other automatons with different functions, including guard

dogs to protect the palace of Alkinoos, horses for the chariot of the Cabires, or the giant Talos to guard the island of Crete. Items crafted by Hephaestus are also clever, allowing the gates of Olympus to open on their own or the bellows of the forge to work autonomously. The materials such as gold and bronze used to make these artificial creatures offer them immense resistance and even immortality. These automatons are there to serve the gods and to perform tedious, repetitive, and daunting tasks to perfection by surpassing mortals. No one can escape the dog forged by Hephaestus, and no one can circumnavigate Crete three times a day as Talos does. The human dream may have found its origins here. In the time of Kronos, humans lived with the gods and led a life without suffering, without pain, and without work, because nature produced abundantly without effort. All you had to do was stretch out your arm to pick up the fruit. The young golden servants “perfectly embody the wealth, the beauty, the strength, the vitality of this bygone golden age for humans” (J.W. Alexandre Marcinkowski). This perfect world, without slavery, without thankless tasks, where humans do not experience fatigue and can dedicate themselves to noble causes, was taken up by certain philosophers including Aristotle, who in a famous passage from *Politics* sees in artificial creatures an advantage that is certain:

If every tool, when ordered, or even of its own accord, could do the work that benefits it... then there would be no need either of apprentices for the master workers or of slaves for the lords.

Aristotle, *Politics*

We can see in this citation one of the first definitions of AI. Hephaestus does not imitate the living but rather creates it, which is different from imitation. Blacksmith automatons have intelligence, voice, and strength. His creations do not equal the gods, who are immortal and impossible to equal. This difference shows a hierarchy between the gods and those living automatons who are their subordinates. The latter are also superior to humans when considering the perfection of the tasks that are performed simply, without defects or deception. This superiority is not entirely accurate in the sense that some humans have shown themselves to be more intelligent than automatons to achieve their ends. We can cite Medea's overcoming of Talos. Hephaestus is the only deity capable of creating these wondrous creatures. But these myths lay the foundations of the relationship between humans and technology. Hephaestus is inspired by nature, living beings, and the world. He makes models that do not threaten the mortal world. These creatures are even prehumans if we think of Pandora. In the Hellenistic period, machines were created, thanks to scientists and engineers such as Philo of Byzantium or Heron of Alexandria. We have seen the appearance of automatic doors that open by themselves at the sound of a trumpet, an automatic water dispenser, and a machine using the contraction of air or its rarefaction to operate a clock. Many automatons are also described in the *Pneumatika* and *Automaton-Making* by Héron. These automatons amaze but are not considered to produce things industrially and have no economic or societal impact; these machines make shows. At that time, there was likely no doubt that we could perhaps imitate nature and provide the illusion but surely not match it, unlike Hephaestus who instead competes with nature. The works of Hephaestus are perfect, immortal, and capable of “engendering offspring.” When his creatures leave Olympus and join humans, they degenerate and die. Hephaestus, unlike humans, does not imitate the living but instead manufactures it. Despite thousands of years of stories, myths, attempts, and discoveries, we are still far from Hephaestus.

Nevertheless, our understanding has evolved. We have known for a century that our brain runs on fuel, oxygen, and glucose. It also works with electricity since neurons transmit what they have to transmit, thanks to electrical phenomena, using what are called action potentials. Electricity is something we can model. In his time, Galileo said that “nature is a book written in mathematical language.” So, can we seriously consider the creation of a human brain, thanks to mathematics? To imagine programming or simulating thought, you must first understand it, take it apart, and break it down. To encode a reasoning process, you must first be able to decode it. The analysis of this process, or the desire for analysis in any case, has existed for a very long time.

The concepts of modern computing have their origins in a time when mathematics and logic were two unrelated subjects. Logic was notably developed, thanks to two philosophers, Plato and Aristotle. We do not necessarily make the connection, but without Plato, Aristotle, or Galileo, we might not have seen IBM, Microsoft, Amazon, or Google. Mathematics and logic are the basis of computer science. When AI began to develop, it was assumed that the functioning of thought could be mechanized. The study of the mechanization of thought or reasoning has a very long history, as Chinese, Indian, and Greek philosophers had already developed formal deduction methods during the first millennium BCE. Aristotle developed the formal analysis of what is called the syllogism:

- All men are mortal
- Socrates is a man
- Therefore, Socrates is mortal

This looks like an algorithm. Euclid, around 300 BCE. J.-C., subsequently wrote the *Elements*, which develops a formal model of reasoning. Al-Khwārizmī (born around 780 CE) developed algebra and gave the algorithm its name. Moving forward several centuries, in the seventeenth century the philosophers Leibniz, Hobbes, and Descartes explored the possibility that all rational thought could be systematically translated into algebra or geometry. In 1936, Alan Turing laid down the basic principles of computation. This was also a time when mathematicians and logicians worked together and gave birth to the first machines.

In 1956, the expression “artificial intelligence” was born during a conference at the Dartmouth College in the United States. Although computers at the time were being used primarily for what was called scientific computing, researchers John McCarthy and Marvin Minsky used computers for more than just computing; they had big ambitions with AI. Three years later, they opened the first AI laboratory at MIT. There was considerable investment, great ambitions, and a lot of unrealized hope at the time. Among the promises? Build a computer that can mimic the human brain. These promises have not been kept to this day despite some advances: Garry Kasparov was beaten in chess by the IBM Deep Blue machine, IBM’s Watson AI system defeated the greatest players in the game *Jeopardy!*, and AlphaGo beat the greatest players in the board game Go by learning without human intervention. Demis Hassabis, whose goal was to create the best Go player, created AlphaGo. We learned that we were very bad players, contrary to what we had thought. The game of Go was considered at that time to be impregnable. In October 2015, AlphaGo became the first program to defeat a professional (the French player Fan Hui). In March 2016, AlphaGo beat Lee Sedol, one of the best players in the world (ninth dan professional). In May 2017, it defeated world champion Ke Jie.

These are accomplishments. But there are still important differences between human and machine. A machine today can perform more than 200 million billion operations per second, and it is progressing. By the time you read this book, this figure will surely have already been exceeded. On the other hand, if there is a fire in the room, Kasparov will take to his heels while the machine will continue to play chess! Machines are not aware of themselves, and this is important to mention. AI is a tool that can help us search for information, identify patterns, and process natural language. It is machine learning that allows elimination of bias or detection of weak signals. The human component involves common sense, morality, creativity, imagination, compassion, abstraction, dilemmas, dreams, generalization, relationships, friendship, and love.

Machine Learning

In this book, we are not going to philosophize but we will explore how to apply machine learning concretely. Machine learning is a subfield of AI that aims to understand the structure of data and fit that data into models that we can use for different applications. Since the optimism of the 1950s, smaller subsets of AI such as machine learning, followed by deep learning, have created much more concrete applications and larger disruptions in our economies and lives.

Machine learning is a very active field, and some considerations are important to keep in mind. This technology is used anywhere from automating tasks to providing intelligent insights. It concerns every industry, and you are almost certainly using AI applications without knowing it. We can make predictions, recognize images and speech, perform medical diagnoses, devise intelligent supply chains, and much more. In this book, we will explore the common machine learning methods. The reader is expected to understand basic Python programming and libraries such as NumPy or Pandas. We will study how to prepare data before feeding the models by showing the math and the code using well-known open-source frameworks. We will also learn how to run these models on not only classical computers (CPU- or GPU-based) but also quantum computers. We will also learn the basic mathematical concepts behind machine learning models.

From Theory to Production

One important step in our journey to AI is how we put the models we have trained into production. The best AI companies have created data hubs to simplify access to governed, curated, and high-quality data. These data are accessible to any user who is authorized, regardless of where the data or the user is located. It is a kind of self-service architecture for data consumption. The reason we need to consider the notion of a data hub is that we are in a world of companies that have multiple public clouds, on-premises environments, private clouds, hybrid clouds, distributed clouds, and other platforms.

Understanding this world is a key differentiator for a data scientist. This is what we call data governance, which is critical to an organization if they really want to benefit from AI. How much time do we spend retrieving data?

Another important topic regarding AI is how we ensure that the models we develop are trustworthy. As humans and AI systems are increasingly working together, it is essential that we trust the output of these systems. As scientists or engineers, we need to work on defining the dimensions of trusted AI, outlining diverse approaches to achieve the different dimensions, and determining how to integrate them throughout the entire lifecycle of an AI application.

The topic of AI ethics has garnered broad interest from the media, industry, academia, and government. An AI system itself is not biased per se but simply learns from whatever the data teaches it. As an example of apparent bias, recent research has shown significantly higher error rates in image classification for dark-skinned females than for men or other skin tones. When we write a line of code, it is our duty and our responsibility to make sure that unwanted bias in datasets and machine learning models does not appear and is anticipated before putting something into production. We cannot ignore that machine learning models are being used increasingly to inform high-stakes decisions about real people. Although machine learning, by its very nature, is always a form of statistical discrimination, the discrimination becomes objectionable when it places certain privileged groups at a systematic advantage and certain unprivileged groups at a systematic disadvantage. Bias in training data, due to either prejudice in labels or under- or over-sampling, yields models with undesirable results.

I believe that most people are increasingly interested in rights in the workplace, access to health care and education, and economic, social, and cultural rights. I am convinced that AI can provide us with the opportunity and the choice to improve these rights. It will improve the way we perform tasks and allow us to focus on what really matters, such as human relations, and give us the freedom and time to develop our creativity and somehow, as in the past, have time to reflect. AI is already improving our customer experience. When we think of customer experience, we can consider one of the most powerful concepts that Mahatma Gandhi mentioned – the concept of Antyodaya, which means focusing on the benefits for the very last person in a line or a company. When you have to make choices, you must always ask yourself what impact it has on the very last person. So, how are our decisions on AI or our lines of code going to affect a young patient in a hospital or a young girl in school? How will AI affect the end user? The main point is about how technology can improve the customer experience. AI is a technology that will improve our experiences, and I believe it will help us focus on improving humanity and give us time to develop our creativity.

AI can truly help us manage knowledge. As an example, roughly 160,000 cancer studies are published every year. The amount of information available in the world is so vast in quantity that a human cannot process this information. If we take 15 minutes to read a research paper, we will need 40,000 hours a year to read 160,000 research papers; we only have 8,760 hours in a year. Would we not want each of our doctors to be able to take advantage of this knowledge and more to learn about us and how to help us stay healthy or help us deal with illnesses? Cognitive systems can be trained by top doctors and read enormous amounts of information such as medical notes, MRIs, and scientific research in seconds and improve research and development by analyzing millions of papers not only from a specific field but also from all related areas and new ways to treat patients. We can use and share these trained systems to provide access to care for all populations.

This general introduction has aimed to clarify the potential of AI, but if you are reading these lines, it is certainly because you are already convinced. The real purpose of this book is to introduce you to the world of machine learning by explaining the main mathematical concepts and applying them to real-world data. Therefore, we will use Python and the most often-used open-source libraries. We will learn several concepts such as feature rescaling, feature extraction, and feature selection. We will explore the different ways to manipulate data such as handling missing data, analyzing categorical data, or processing time-related data. After the study of the different preprocessing strategies, we will approach the most often-used machine learning algorithms such as support vector machine or neural networks and see them run on classical (CPU- and GPU-based) or quantum computers. Finally, an important goal of this book is to apply our models into production in real life through application programming interfaces (APIs) and containerized applications by using Kubernetes and OpenShift as well as integration through machine learning operations (MLOps).

I would like to take the opportunity here to say a warm thank you to all the data scientists around the world who have openly shared their knowledge or code through blogs or data science platforms, as well as the open-source community that has allowed us to improve our knowledge in the field. We are grateful to have all these communities – there is always someone who has written about something we need or face.

This book was written with the idea to always have nearby a book that I can open when I need to refresh some machine learning concepts and reuse some code. All code is available online and the links are provided.

I hope you will enjoy reading this book, and any feedback to improve it is most welcome!

1

Concepts, Libraries, and Essential Tools in Machine Learning and Deep Learning



Photo by Annamária Borsos

In this first chapter, we will explore the different concepts in statistical learning as well as popular open-source libraries and tools. This chapter will serve as an introduction to the field of machine learning for those with a basic mathematical background and software development skills. In general, machine learning is used to understand the structure of the data we have at our disposal and fit that data into models to be used for anything from automating tasks to providing intelligent insights to predicting a behavior. As we will see, machine learning differs from traditional computational approaches, as we will train our algorithms on our data as opposed to explicitly coded instructions and we will use the output to automate decision-making processes based on the data we have provided.

Machine learning, deep learning, and neural networks are branches of artificial intelligence (AI) and computer science. Specifically, deep learning is a subfield of machine learning, and neural networks are a subfield of deep learning. Deep learning is more focused on feature extraction to enable the use of large datasets. Unlike machine learning, deep learning does not require human intervention to process data.

Exploration and iterations are necessary for machine learning, and the process is composed of different steps. A typical machine learning workflow is summarized in Figure 1.1.

In this chapter, we will see concepts that are applied in machine learning, including unsupervised methods such as clustering to group unlabeled data such as K-means or dimensionality reduction to reduce the number of features in order to better summarize and visualize the data such as principal component analysis, feature extraction to define attributes in image and text data, feature selection to identify meaningful features to create better supervised models, and cross-validation to estimate the performance of models on new data or ensemble methods to combine the predictions of multiple models.

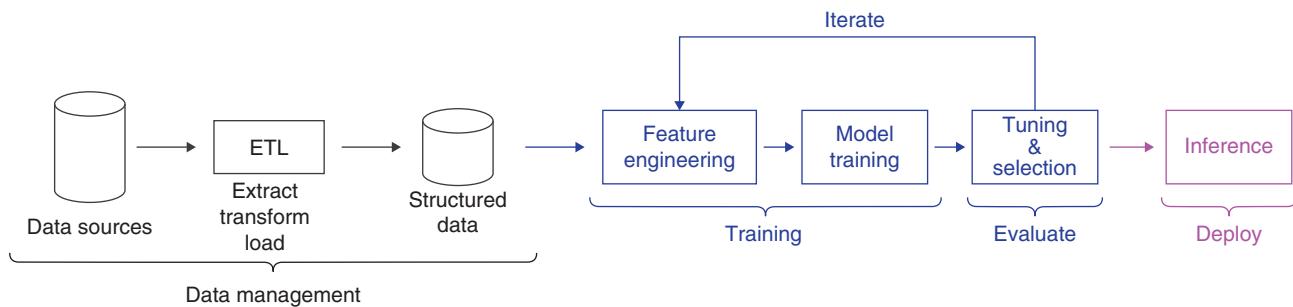


Figure 1.1 An example of a machine learning workflow.

1.1 Learning Styles for Machine Learning

The literature describes different types of machine learning algorithms classified into categories. Depending on the way we provide information to the learning system or on whether we provide feedback on the learning, these types fall into categories of supervised learning, unsupervised learning, or reinforcement learning.

1.1.1 Supervised Learning

Supervised learning algorithms are by far the most widely adopted methods. The algorithms are based on labeled and organized data for training. For example, you can feed a machine learning algorithm with images labeled as “dog” or “cat.” After training on these data, the algorithm should be able to identify unlabeled “dog” and “cat” images and recognize which image is a dog and which image is a cat. The main concept for the algorithm is to be able to “learn” by comparing its actual output with “taught” outputs. Supervised learning involves identifying patterns in data to predict label values on additional unlabeled data. In other words, a supervised machine learning algorithm suggests that the expected answer in upcoming data has already been identified in a historical dataset containing the correct answers.

There are many examples in which supervised learning can be applied, including using historical data to predict future events such as upcoming stock market fluctuations, spam detection, bioinformatics, or object recognition. Supervised learning algorithms have some challenges such as the preprocessing of data and the need for regular updates.

Mathematically, we start with some dataset:

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\} \subseteq \mathcal{R}^d \times C$$

where each x_i is a d -dimensional feature vector of the i th example, y_i the label of the i th example, n the size of the dataset, C the label space and \mathcal{R}^d the d -dimensional feature space. We assume that the data points have been drawn from some unknown distribution $(x_i, y_i) \sim \varphi$ with (x_i, y_i) to be independent and identically distributed (iid). The objective of supervised machine learning is to find a function $f : \mathcal{R}^d \rightarrow C$ that for every new input or output (x, y) sampled from φ we have $f(x) \approx y$.

Usually, the data that are being processed do not represent images by itself but rather metadata associated with it such as the well-known Iris dataset (<https://archive.ics.uci.edu/ml/datasets/Iris>), which provides different iris features (petal width and length, sepal width and length) as inputs and the classes of iris (Setosa, Versicolour, Virginica) as outputs.

When considering supervised learning methods, we need to understand the feature and label spaces that we have. Usually, supervised learning operates with the following label spaces:

- **Binary classification:** The algorithm will classify the data into two categories such as spam or not spam. The label space is $\{0, 1\}$ or $\{-1, 1\}$.
 - **Multi-class classification:** The algorithm needs to choose among more than two types of answers for a target variable such as the recognition of images containing animals (e.g., dog = 1, cat = 2, fish = 3, etc.). If we have N image classes, we have $C = \{1, 2, \dots, N\}$.
 - **Regression:** Regression models predict continuous variables as opposed to classification models, which consider categorical variables. For example, if we attempt to predict measures such as temperature, net profit, or the weight of a person, this will require regression models. Here, $C = \mathcal{R}$.

In supervised learning, we can find popular classification models such as decision trees, support vector machine, naïve Bayes classifiers, random forest, or neural networks. We can also find popular regression models such as linear regression, ridge regression, ordinary least squares regression, or stepwise regression.

As mentioned above, we need to find a function $f : \mathcal{R}^d \rightarrow C$. This requires some steps such as making some assumptions regarding what the function f looks like and what space of functions we will be using (linear, decision trees, polynomials, etc.). This is what we call the hypothesis space \mathcal{H} . This choice is very important because it impacts how our model will generalize to completely new data that has not been used for training.

1.1.1.1 Overfitting and Underfitting

The largest challenge in supervised learning is effective generalization, which means the ability of a machine learning model to provide a suitable output by adapting to the given set of unknown input. In other words, the question is how to ensure our model will perform well on new data after training (Figure 1.2). Given our dataset D , we can define the function f as follows:

$$f(x) = \begin{cases} y_i, & \text{if there exists } (x_i, y_i) \text{ such that } x = x_i \\ 0, & \text{otherwise} \end{cases}$$

We can see that the function would perform perfectly with our training data, but if something new is introduced, the results would certainly be wrong. Underfitting and overfitting are two things we need to control to make the performance of the model stable and to determine whether the model is generalizing well. There is a trade-off that we need to accept

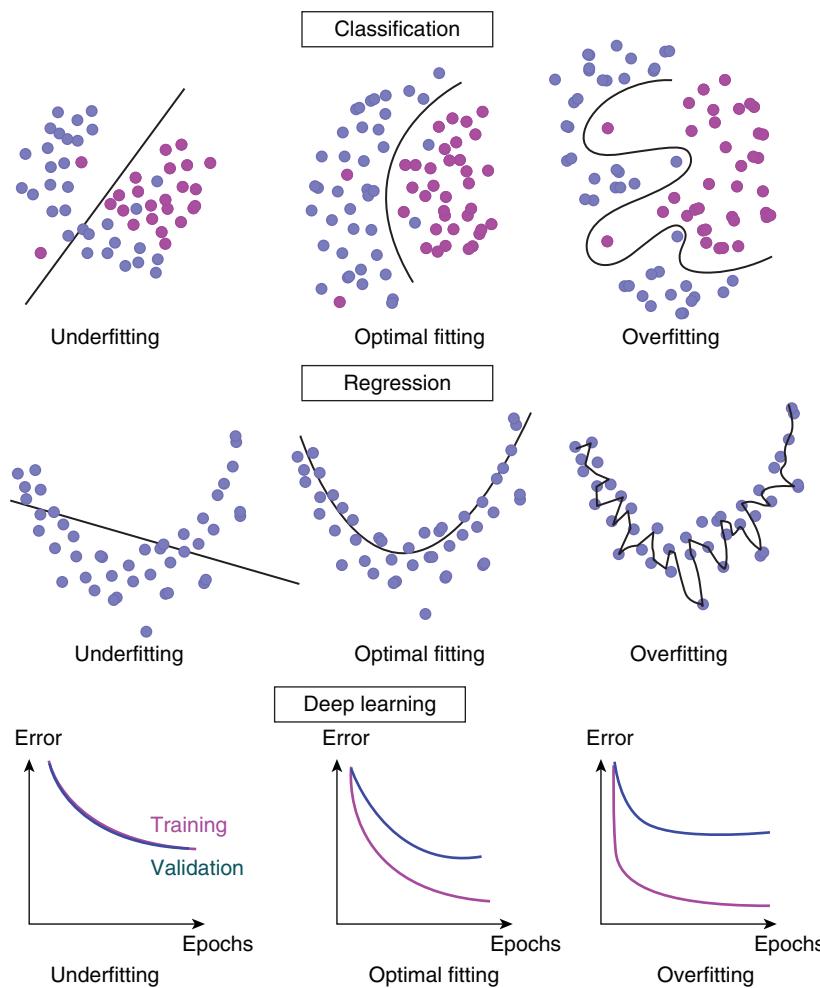


Figure 1.2 Underfitting, optimal fitting, and overfitting in classification, regression, and deep learning.

between underfitting or bias and overfitting or variance. Bias means a prediction error that is introduced in the algorithm due to oversimplification or the differences between the predicted values and the actual values. Variance occurs when the model performs well with training data but not with test data. We should also introduce two other words: signal and noise. Signal refers to the true underlying pattern of the data that allows the algorithm to learn from data, whereas noise is irrelevant and unnecessary data that reduces the performance of the algorithm.

In **overfitting**, the machine learning model will attempt to cover more than the necessary data points present in a dataset or simply all data points. It is a modeling error in statistics because the function is too closely aligned to the training dataset and will certainly translate to a reduction in efficiency and accuracy with new data because the model has considered inaccurate values in the dataset. Models that are overfitted have low bias and high variance. For example, overfitting can come when we train our model excessively. To reduce overfitting, we can perform cross-validation, regularization, or ensembling, train our model with more data, remove unnecessary features, or stop the training of the model earlier. We will see all these techniques.

On the opposite, we have **underfitting**, with high bias and low variance, which occurs when our model is not able to capture the underlying trend of the data, generating a high error rate on both the training dataset and new data. An underfitted model is not able to capture the relationships between input and output variables accurately and produces unreliable predictions. In other words, it does not generalize well to new data. To avoid underfitting, we can decrease the regularization used to reduce the variance, increase the duration of the training, and perform feature selection.

Some models are more prone to overfitting than others such as KNN or decision trees. The goal of machine learning is to achieve “goodness of fit,” which is a term based on the statistics that define how closely the results or predicted values match the true values of the dataset. As we can imagine, the ideal fit of our model is between underfitting and overfitting making predictions with zero errors. As we will see during our journey to mastering machine learning, this goal is difficult to achieve. In addition, overfitting can be more difficult to identify than underfitting because the training data perform at high accuracy. We can assess the accuracy of our model by using a method called k-folds cross-validation.

1.1.1.2 K-Folds Cross-Validation

Cross-validation consists of a resampling procedure to assess machine learning models on a limited data sample. Cross-validation has a parameter called k that refers to the number of groups into which a given dataset is to be split. Therefore, in k-folds cross-validation, data are split into k equally sized subsets (folds). When we define a specific value for k , for instance, $k = 10$, k-fold cross-validation becomes tenfold cross-validation. When the data are separated into k-folds, one of the k-folds is considered as a test set (holdout set or validation set), and the rest of the folds serve to train the model. This process is repeated until each of the folds has acted as validation test. Each repetition comes with an evaluation. When all iterations have been completed, the scores are averaged, providing a general assessment of the performance of the model. It is a popular method because of its simplicity and robustness, as it generally results in a less biased or less optimistic estimate of models than other methods such as train/test split.

As mentioned above, the general procedure consists of shuffling the dataset randomly and splitting the dataset into k groups. For each group, we consider an initial group as test dataset and the rest as training dataset; we fit a model on the training set and evaluate it with the test dataset. We then retain the evaluation score, discard the model, and repeat the steps with other groups. In the end, we average the scores to evaluate the performance of the global model. The technique allows each sample to be used in the test set one time and used to train the model ($k - 1$) times. Cross-validation can have some drawbacks, for example, when processing data evolving over a period or inconsistent data. To illustrate the method, let's say we divide a dataset into five subgroups as in Figure 1.3.

1.1.1.3 Train/Test Split

A general approach in machine learning is to split a dataset D into three sets : training D_{TR} , validation D_{VA} , and test D_{TE} . We can consider an effective split as approximately 80% of the data for training and 20% for validation (10%) and testing (10%). Of course, this split depends on the amount of data and context.

The split is performed at the very beginning of the machine learning process when we start our search for the function $f : \mathcal{R}^d \rightarrow C$ described above. The training dataset helps to find $f : \mathcal{R}^d \rightarrow C$, and the validation set helps evaluate a given model. The split is used to fine-tune the model hyperparameters by providing an unbiased evaluation of a model fit on

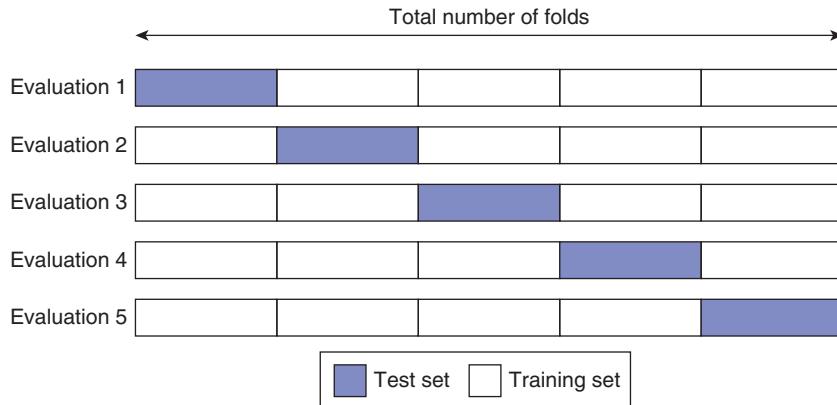


Figure 1.3 Dataset divided into five subgroups for fivefold cross-validation.

the training dataset. The more the validation dataset is incorporated into the model configuration, the more the evaluation becomes biased. The test dataset is used to provide an unbiased evaluation of a final model; it is only used when the model has been completely trained using training and validation datasets. It is important to carefully curate the test dataset in order to represent the real world.

1.1.1.4 Confusion Matrix

The confusion matrix is another important concept in machine learning used to determine the performance of a classification model for a given set of test data. It shows the errors in the form of an $N \times N$ matrix (error matrix) where N is the number of target classes. It compares the actual target values with those predicted by the machine learning algorithm. For example, for a binary classification problem, we will have a 2×2 matrix; for three classes, we will have a 3×3 table; and so on.

In Figure 1.4, the matrix contains two dimensions, representing predicted values and actual values, along with the total number of predictions. The target variable has two values (positive or negative); the columns represent the actual values and the rows the predicted values of the target variable. Inside the matrix, we have the following possibilities:

- **True positive (TP):** The actual value was positive, and the model predicted a positive value.
- **True negative (TN):** The actual value was negative, and the model predicted a negative value.
- **False positive (FP):** The actual value was negative, but the model predicted a positive value (the predicted value was falsely predicted, also known as type 1 error).
- **False negative (FN):** The actual value was positive, but the model predicted a negative value (the predicted value was falsely predicted, also known as type 2 error).

Let us take an example (Figure 1.5) with a classification dataset of 1000 data points with a fitted classifier that has provided the confusion matrix shown. In Figure 1.5, we see that 560 positive-class data points were correctly classified by the model, 330 negative-class data points were correctly classified, 60 negative-class data points were not correctly classified as belonging to the positive class, and 50 positive-class data points were not classified correctly as belonging to the negative class by the model. We can conclude that the classifier is acceptable.

		Actual values	
		Positive	Negative
Predicted values	Positive	TP	FP
	Negative	FN	TN

Figure 1.4 Confusion matrix.

		Actual values	
		Positive	Negative
Predicted values	Positive	560	60
	Negative	50	330

Figure 1.5 An example of a confusion matrix.

To make things more concrete visually, let us say we have a set of 10 people and we want to build a model to predict whether they are sick. Depending on the predicted value, the outcome can be TP, TN, FP, or FN:

ID	Actual sick	Predicted sick	Confusion matrix
1	1	1	TP
2	1	1	TP
3	1	0	FP
4	0	1	FN
5	0	1	FN
6	0	0	TN
7	1	1	TP
8	0	0	TN
9	1	0	FP
10	0	1	FN

With the help of a confusion matrix, it is possible to calculate a measure of performance such as accuracy, precision, recall, or others.

Classification Accuracy Classification accuracy is one of the most important parameters to assess the accuracy of a classification problem. It simply indicates how often a model predicts the correct output. It is calculated as the ratio of the number of correct predictions to the total number of predictions made by the classifiers:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Misclassification Rate The misclassification rate, also known as error rate, defines how often the model provides incorrect predictions. To calculate the error rate, we compute the number of incorrect predictions to the total number of predictions made by the classification model:

$$\text{Error rate} = \frac{FP + FN}{TP + TN + FP + FN}$$

Precision and Recall Precision is the number of correct outputs, or how many of the predicted positive cases were truly positive. This is a way to estimate whether our model is reliable.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall defines how our model predicted correctly out of total positive classes. In other words, how many of the actual positive cases were predicted correctly:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F-Score The traditional F-measure or balanced F-score (F₁ score) is used to evaluate recall and precision at the same time. It is a harmonic mean of precision and recall. The F-score is highest when precision and recall are equal.

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} = 2 \times \frac{precision \cdot recall}{precision + recall}$$

The F-score is widely used in the natural language processing literature (name entity recognition, word segmentation, etc.).

The metrics described above are the most widely used ones. There are a number of other important metrics that we can explore to fit our context, including the Fowlkes–Mallows index, Matthews correlation coefficient, Jaccard index, diagnostic odds ratio, and others. We can find all these metrics in the literature regarding diagnostic testing in machine learning.

To provide a simple example of code, let us say that we wish to build a supervised machine learning model called linear discriminant analysis (LDA) and print the classification accuracy, precision, recall, and F_1 score. Before doing this, we have previously split the dataset into training data (X_{train} and y_{train}) and testing data (X_{test} and y_{test}). We want to apply cross-validation ($k = 5$) and print the scores. To run this code, we need to install scikit-learn, as described later in this chapter.

Input:

```
# Importing required libraries
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import cross_val_score

# Define a function that performs linear discriminant analysis
def lda(X_train, X_test, y_train, y_test):

    # Instantiate a LinearDiscriminantAnalysis object
    clf=LinearDiscriminantAnalysis()

    # Fit the LDA model using the training data
    clf.fit(X_train,y_train)

    # Use the trained model to make predictions on the test data
    y_pred = clf.predict(X_test)

    # Print out the results of the linear discriminant analysis
    print("\n")
    print("Linear Discriminant Analysis Metrics")
    print("Classification accuracy:", metrics.accuracy_score(y_test, y_pred))
    print("Precision:", metrics.precision_score(y_test, y_pred, average='micro'))
    print("Recall:", metrics.recall_score(y_test, y_pred, average='micro'))
    print("F1 score:", metrics.f1_score(y_test, y_pred, average='micro'))

    # Use cross-validation to estimate the performance of the model on new data
    print("Cross validation:",cross_val_score(clf, X_train, y_train, cv=5))
    print("\n")
```

1.1.1.5 Loss Functions

In supervised machine learning algorithms, the objective is to minimize the error of each training example during the learning process. The loss function $L : \mathcal{H} \rightarrow \mathbb{R}$ is a measure of how well our prediction model f can predict the expected outcome by assigning a loss to each $f \in \mathcal{H}$. Here, we simply convert the learning problem into an optimization problem with the aim of defining a loss function and then optimizing the model to minimize the loss function:

$$\arg \min_{h \in \mathcal{H}} L(h) = \arg \min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n l(x_i, y | h)$$

where we use the loss function L of a hypothesis h given D and l for the loss of a single data pair (x_i, y) given h . There are different types of loss functions such as squared error loss, absolute error loss, Huber loss, or hinge loss. For clarity, it is also important to mention the cost function, which is the average loss over the entire training dataset; in contrast, the loss function is computed for a single training example. The loss function will have a higher value if the predictions do not appear

accurate and a lower value if the model performs fairly well. The cost function quantifies the error between predicted values and expected values and presents it in the form of a single number. The purpose of the cost function is to be either minimized (cost, loss, or error) or maximized (reward). Cost and loss refer almost to the same concept, but the cost function invokes a penalty for a number of training sets or the complete batch, whereas the loss function mainly applies to a single training set. The cost function is computed as an average of the loss functions and is calculated once; in contrast, the loss function is calculated at every instance.

For example, in linear regression, the loss function used is the squared error loss and the cost function is the mean squared error (MSE). The **squared error loss**, also known as L2 loss, is the square of the difference between the actual and the predicted values:

$$L = (y - f(x))^2$$

The corresponding cost function is the **mean of these squared** errors. The overall loss is then:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - f(x_i))^2$$

The MSE can be implemented as follows:

```
def MSE(y_predicted, y_actual):
    squared_error = (y_predicted - y_actual) ** 2
    sum_squared_error = np.sum(squared_error)
    mse = sum_squared_error / y_actual.size
    return mse
```

Another regression loss function is the **absolute error loss**, also known as the L1 loss, which is the difference between the predicted and the actual values, irrespective of the sign:

$$L = |y - f(x)|$$

The cost is the mean of the absolute errors (MAE), which is more robust than MSE regarding outliers.

Another example is the Huber loss, which combines the MSE and MAE by taking a quadratic form for smaller errors and a linear form otherwise:

$$\text{Huber} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

In the formula above, Huber loss is defined by a δ parameter; it is usually used in robust regression or m -estimation and is more robust to outliers than MSE.

For binary classification, we can use the **binary cross-entropy loss** (also called log loss), which is measured for a random variable X with probability distribution $p(X)$:

$$L = \begin{cases} - \int p(x) \cdot \log p(x) \cdot dx, & \text{if } x \text{ is continuous} \\ - \sum_x p(x) \cdot \log p(x), & \text{if } x \text{ is discrete} \end{cases}$$

If there is greater uncertainty in the distribution, the entropy for a probability distribution will be greater.

The **hinge loss** function is very popular in support vector machines (SVMs) with class labels 1 and -1:

$$L = \max(0, 1 - y * f(x))$$

Finally, for multi-class classification, we can use the **categorical cross-entropy loss**, also called softmax loss. It is a softmax activation plus a cross-entropy loss. In a multi-label classification problem, the target represents multiple classes at once. In this case, we calculate the binary cross-entropy loss for each class separately and then sum them up for the complete loss.

We can find numerous methodologies that we can explore to better optimize our models.

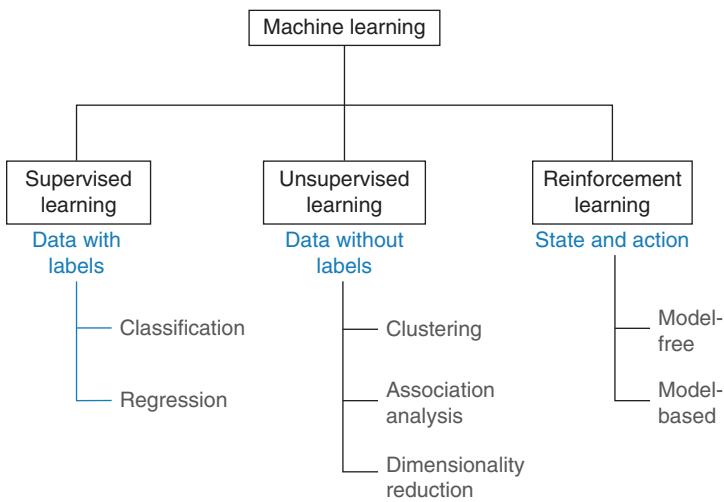


Figure 1.6 Summary of the different learning methods.

1.1.2 Unsupervised Learning

Unsupervised learning uses machine learning algorithms to analyze and cluster unlabeled datasets that are not classified or categorized. Here, the machine learning algorithms do not need external supervision to learn from the data and have no predefined output. The goal is to find insights from a large amount of data. The data can be classified as clusters or by association. We can use unsupervised learning for exploratory data analysis or for reducing the number of features in a model through the process of dimensionality reduction such as principal component analysis (PCA). Examples of some unsupervised learning algorithms are K-means clustering, neural networks, and probabilistic clustering methods.

1.1.3 Semi-Supervised Learning

Semi-supervised learning is the middle ground between supervised and unsupervised learning. It means that during training, semi-supervised algorithms use a smaller labeled dataset to guide classification and feature extraction from a larger and unlabeled dataset. This strategy is ideal when there is an insufficient amount of labeled data to train supervised learning algorithms. Similar to supervised learning, semi-supervised learning can solve classification and regression problems.

1.1.4 Reinforcement Learning

In reinforcement learning, the algorithm is not trained using sample data but rather “learns as it goes” by using trial and error. In this method, an agent is interacting with its environment by producing actions. The agent will learn from the feedback it receives in the form of rewards: a positive reward if it performs well, a negative reward if it performs poorly.

The different learning methods are summarized in Figure 1.6.

1.2 Essential Python Tools for Machine Learning

I will not present very much on Python (<https://www.python.org>) itself in this section, but I must say that the language of data scientists is indeed Python! This platform is widely used for machine learning applications and is clearly a common choice across academia and industry. Python is easy to learn and to read, and its syntax is accessible to beginners. What also makes Python powerful is its huge community of developers and data scientists who make Python easier for beginners by sharing open-source projects, libraries, tutorials, and other examples. I am grateful to this community, which has helped me progress.

1.2.1 Data Manipulation with Python

I of course will not cite all the available tools, but I will present a few of them here. **NumPy** (<https://numpy.org>) is certainly one of the first libraries that we need to download for Python. NumPy is the tool to process data and numbers, for example, by offering an N -dimensional array object for user data and providing transformative functions (linear algebra, transformations, random numbers, etc.). **Pandas** (an abbreviation of the term “panel data”) is another must-have. Pandas (<https://pandas.pydata.org>) provides easy-to-use data structures for easier data preprocessing and manipulation as well as data analysis and statistical functions. Pandas offers an easy way to read and write data in memory data structures as well as methods for indexing, reshaping, merging, joining, filtration, or time series functionalities. After all these manipulations, there is something else important: visualization! **Matplotlib** (<https://matplotlib.org>) is certainly one of the most popular libraries for visualizing data. With Matplotlib, it is possible to build and design graphs such as line, bar, scatter, or histogram plots. We can also use **Seaborn** (<https://seaborn.pydata.org>) on top of Matplotlib to offer beautiful plotting styles. **SciPy** (<https://scipy.org>) is another classic tool used for scientific computation. The open-source Python library contains modules for optimization, linear algebra, integration, interpolation, Fast Fourier Transform (FFT), signal and image processing, and Ordinary Differential Equation (ODE) solvers.

1.2.2 Python Machine Learning Libraries

1.2.2.1 Scikit-learn

Scikit-learn (<https://scikit-learn.org/stable/>), often referred to as sklearn, provides a wide range of open-source and commercially usable (BSD license) algorithms for feature extraction, feature selection, dimensionality reduction, clustering, regression, classification, ensemble methods, manifold learning, parameter tuning, or model selection via a consistent interface in Python. Scikit-learn provides popular tools that are accessible and simple to use. The library is built on NumPy, SciPy, Matplotlib, Sympy (symbolic mathematics), Pandas, and IPython (enhanced interactive console), which are necessary to install:

```
pip install numpy
pip install scipy
pip install matplotlib
pip install ipython
pip install sympy
pip install pandas
```

We can then install sklearn:

```
pip install scikit-learn
```

Then, we can simply import sklearn into the Python code:

```
import sklearn
```

We will see in Chapter 2 how to use sklearn. One of the great things about scikit-learn is that it is very well documented, with many code examples. Python is the interface, but C libraries are leveraged for performance such as NumPy for arrays and matrix operations. In addition, scikit-learn provides several in-built datasets that are easy to import.

1.2.2.2 TensorFlow

TensorFlow (<https://www.tensorflow.org>) is a Python library created by Google in late 2015. At first, Google used it for internal needs and open-sourced it (Apache open-source license) to the community. Since then, TensorFlow has become one of the largest Python libraries in the open-source machine learning domain. We all know Gmail, Google Search, and YouTube; these applications utilize TensorFlow. One of the advantages of TensorFlow is that it examines data in tensors, which are multi-dimensional arrays that can process large amounts of data. All library actions are performed within a graph (graph-based architecture) made of a sequence of computations, all interconnected. Another advantage is that TensorFlow

is designed to run on numerous CPUs, GPUs, or mobile devices. It is easy to execute the code in a distributed way across clusters and to use GPUs to dramatically improve the performance of training. The graphs are portable, allowing saving of computations and executing them at our conveyance. We can divide the operations of TensorFlow into three parts: preparing the data, model creation, and training and estimation of the model.

TensorFlow includes different models for computer vision (image or video classification, object detection, and segmentation), natural language processing, or recommendations. To perform these operations, TensorFlow uses models such as deep residual learning for image recognition (ResNet), Mask-CNN, ALBERT (a light Bidirectional Encoder Representations from Transformers [BERT] for self-supervised learning of language representations), or neural collaborative filtering.

Installation of TensorFlow is straightforward (with pip):

```
https://www.tensorflow.org/install/pip#system-install
```

As mentioned above, one of the main reasons to run TensorFlow is the use of GPUs. On a computer or cluster running with GPUs, it will be easy to run algorithms using TensorFlow. It will be necessary to install a CUDA toolkit and libraries (cuDNN libraries) that need to interact with GPUs.

Let us view an example by installing cuDNN and the CUDA toolkit on Ubuntu 20.04. To install the CUDA toolkit, we can apply the following instructions in a terminal:

```
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/
cuda-ubuntu2004.pin
sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600

wget https://developer.download.nvidia.com/compute/11.3.0/local_installers/
cuda-repo-ubuntu2004-11-3-local_11.3.0-465.19.01-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu2004-11-3-local_11.3.0-465.19.01-1_amd64.deb

sudo apt-key add /var/cuda-repo-ubuntu2004-11-3-local/7fa2af80.pub
sudo apt-get update

sudo apt-get -y install cuda
export PATH=/usr/local/cuda-11.3/bin${PATH:+:$PATH}
export LD_LIBRARY_PATH=/usr/local/cuda-11.3/lib64${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}
```

We can now check whether nvidia-persistenced is running as a daemon from system initialization:

```
systemctl status nvidia-persistenced
```

If the daemon is not running, then we can input the following:

```
sudo systemctl enable nvidia-persistenced
```

To obtain the NVIDIA driver version, we can enter the following:

```
cat /proc/driver/nvidia/version
```

To install the cuDNN libraries, we need to download the versions that correspond to our environment (<https://developer.nvidia.com/cudnn>). For a local computer running Ubuntu 20.04, they are the following:

- cuDNN runtime library for Ubuntu20.04 x86_64 (Deb).
- cuDNN developer library for Ubuntu20.04 x86_64 (Deb).
- cuDNN code samples and user guide for Ubuntu20.04 x86_64 (Deb).

After downloading files, we should have the following .deb packages:

- libcudnn8-samples_8.2.0.53-1+cuda11.3_amd64.deb.
- libcudnn8-dev_8.2.0.53-1+cuda11.3_amd64.deb.
- libcudnn8_8.2.0.53-1+cuda11.3_amd64.deb.

Now we can start the install procedure:

```
sudo dpkg -i libcudnn8_8.2.0.53-1+cuda11.3_amd64.deb
sudo dpkg -i libcudnn8-dev_8.2.0.53-1+cuda11.3_amd64.deb
sudo dpkg -i libcudnn8-samples_8.2.0.53-1+cuda11.3_amd64.deb
```

Then, we can run the command below in our terminal to see whether cuDNN is communicating with the NVIDIA driver:

```
nvidia-smi
```

To test the installation, we simply type the following commands in our terminal; we should see “Test passed”:

```
cp -r /usr/src/cudnn_samples_v8/ $HOME
cd $HOME/cudnn_samples_v8/mnistCUDNN/
make
./mnistCUDNN
```

1.2.2.3 Keras

Keras (<https://keras.io>) is a Python framework that provides a library with an emphasis on deep learning applications. Keras can run on top of TensorFlow, Theano, a Python library and optimizing compiler for manipulating and evaluating mathematical expressions, and enables fast experimentation through a user-friendly interface. It runs on both CPUs and GPUs. Keras is mainly used for computer vision using convolutional neural networks or for sequence and time series using recurrent neural networks. We can also use Keras applications, in which we can find deep learning models that are pre-trained. The models can be applied for feature extraction, fine-tuning, or prediction. The framework also provides different datasets such as the MNIST dataset containing 70,000 28×28 grayscale images with 10 different classes that we can load directly using Keras.

1.2.2.4 PyTorch

PyTorch (<https://pytorch.org>), released to open-source in 2017, is a Python machine learning library based on the Torch machine learning library with additional features and functionalities, making the deployment of machine learning models faster and easier. In other words, it combines the GPU-accelerated backend libraries from Torch with a user-friendly Python frontend. PyTorch is written in Python and integrated with libraries such as NumPy, SciPy, and Cython for better performance.

PyTorch contains many features for data analysis and preprocessing. Facebook has developed a machine learning library called Caffe2 that has been merged into PyTorch; thus, Caffe is now a portion of PyTorch. In addition, PyTorch has an ecosystem of libraries such as skorch (sklearn compatibility), Captum (model interpretability), or Glow, a compiler and execution engine for hardware accelerators.

Of course, there are many more machine learning frameworks in the open-source world such as OpenCV or R; we would need more than a book to describe all of them. But with scikit-learn, TensorFlow, Keras, and PyTorch, much can already be done! Beyond the open-source frameworks, it is also important to consider how we can improve AI lifecycle management, unite different teams around the data, and accelerate the time to value creation.

1.2.3 Jupyter Notebook and JupyterLab

Jupyter Notebook and JupyterLab (<https://jupyter.org>) are free software packages that provide web services for interactive computing across all programming languages. Jupyter Notebook is a web application to develop and present data-science projects interactively. We can use it on a local machine or connect it to a remote server. It integrates code and its output into a document to visualize the results and include narrative text, mathematical equations, or charts. The advantage of Jupyter Notebook is its interactivity and user-friendly interface with the possibility to communicate and share code and results. The code or a portion of the code is easily executable. JupyterLab is a web-based interactive development environment. It is widely used to manage workflows in data science. Each code segment in this book is saved in a Jupyter Notebook and is freely available.

We can run Jupyter Notebook on a remote server and access it through our local machine by following the steps below.

First, we launch Jupyter Notebook from remote server by selecting a port number such as 8080:

```
# Replace <PORT> with your selected port number
jupyter notebook --no-browser --port=8080
```

Then, we access the Notebook from the remote machine over SSH by setting up an SSH tunnel:

```
# Replace <PORT> with the port number you selected in the above step
# Replace <REMOTE_USER> with the remote server username
# Replace <REMOTE_HOST> with your remote server address
ssh -L 8080:localhost:<PORT> <REMOTE_USER>@<REMOTE_HOST>
```

Finally, we can open a browser and navigate, thanks to the links provided in the terminal.

1.3 HephAistos for Running Machine Learning on CPUs, GPUs, and QPUs

In this book, we introduce hephAistos, an open-source Python framework designed to execute machine learning pipelines on CPUs, GPUs, and quantum processing units (QPUs). The framework incorporates various libraries, including scikit-Learn, Keras with TensorFlow, and Qiskit, as well as custom code.

You may choose to skip this section if you prefer diving directly into the concepts of machine learning or if you already have ideas you would like to experiment with. Throughout the book, we will integrate hephAistos code examples to demonstrate the concepts discussed.

Our aim is to simplify the application of the techniques explored in this book. You can either learn to create your own pipelines or utilize hephAistos to streamline the process. Feel free to explore this section now if you already possess knowledge of machine learning and Python. Alternatively, you can return to it later, as we will incorporate hephAistos pipelines in various examples throughout the book. HephAistos is distributed under the Apache License, Version 2.0.

We encourage contributions to enhance the framework. You can create pipelines using Python functions with parameters or employ specific routines.

Find hephAistos on GitHub: <https://github.com/xaviervasques/hephaistos.git>

1.3.1 Installation

To install hephAistos, you can clone it from GitHub. You can either download the repository directly or use the following command in your terminal:

```
git clone https://github.com/xaviervasques/hephaistos.git
```

Next, navigate to the hephAistos directory and install the required dependencies:

```
pip install -r requirements.txt
```

HephAIstos has the following dependencies:

- Python
- joblib
- numpy
- scipy
- pandas
- scikit-learn
- category_encoders
- hashlib
- matplotlib
- tensorflow
- qiskit
- qiskit_machine_learning

The package includes several datasets for your convenience. Throughout this book, we will delve into various machine learning methods and provide code snippets to demonstrate how hephAIstos operates. The framework supports an array of techniques, as listed below:

- **Feature rescaling:** StandardScaler, MinMaxScaler, MaxAbsScaler, RobustScaler, unit vector normalization, log transformation, square root transformation, reciprocal transformation, Box-Cox, Yeo-Johnson, quantile Gaussian, and quantile uniform.
- **Categorical data encoding:** ordinal encoding, one hot encoding, label encoding, Helmert encoding, binary encoding, frequency encoding, mean encoding, sum encoding, weight of evidence encoding, probability ratio encoding, hashing encoding, backward difference encoding, leave one out encoding, James–Stein encoding, and M-estimator.
- **Time-related feature engineering:** time split (year, month, seconds, etc.), lag, rolling window, and expanding window.
- **Missing values:** row/column removal, statistical imputation (mean, median, mode), linear interpolation, multivariate imputation by chained equation (MICE) imputation, and KNN imputation.
- **Feature extraction:** principal component analysis, independent component analysis, linear discriminant analysis, locally linear embedding, t-distributed stochastic neighbor embedding, and manifold learning techniques.
- **Feature selection:** filter methods (variance threshold, statistical tests, chi-square test, ANOVA F-value, Pearson correlation coefficient), wrapper methods (forward stepwise selection, backward elimination, exhaustive feature selection), and embedded methods (least absolute shrinkage and selection operator, ridge regression, elastic net, regularization embedded into ML algorithms, tree-based feature importance, permutation feature importance).
- **Classification algorithms running on CPUs:** support vector machine with linear, radial basis function, sigmoid and polynomial kernel functions (svm_linear, svm_rbf, svm_sigmoid, svm_poly), multinomial logistic regression (logistic_regression), linear discriminant analysis (lda), quadratic discriminant analysis (qda), Gaussian naive Bayes (gnb), multinomial naive Bayes (mnb), K-neighbors naive Bayes (kneighbors), stochastic gradient descent (sgd), nearest centroid classifier (nearest_centroid), decision tree classifier (decision_tree), random forest classifier (random_forest), extremely randomized trees (extra_trees), multi-layer perceptron classifier (mlp_neural_network), and multi-layer perceptron classifier to run automatically different hyperparameters combinations and return the best result (mlp_neural_network_auto).
- **Classification algorithms running on GPUs:** logistic regression (gpu_logistic_regression), multi-layer perceptron (gpu_mlp), recurrent neural network (gpu_rnn), and two-dimensional convolutional neural network (conv2d).
- **Classification algorithms running on QPUs:** q_kernel_zz, q_kernel_default, q_kernel_8, q_kernel_9, q_kernel_10, q_kernel_11, q_kernel_12, q_kernel_training, q_kernel_8_pegasos, q_kernel_9_pegasos, q_kernel_10_pegasos, q_kernel_11_pegasos, q_kernel_12_pegasos, q_kernel_default_pegasos.
- **Regression algorithms running on CPUs:** linear regression (linear_regression), SVR with linear kernel (svr_linear), SVR with radial basis function (RBF) kernel (svr_rbf), SVR with sigmoid kernel (svr_sigmoid), SVR with polynomial

kernel (svr_poly), multi-layer perceptron for regression (mlp_regression), and multi-layer perceptron to run automatically different hyperparameters combinations and return the best result (mlp_auto_regression).

- **Regression algorithms running on GPUs:** linear regression (gpu_linear_regression).

1.3.2 HephaIstos Function

To run a machine learning pipeline with hephaIstos, you can use the Python function ml_pipeline_function composed of user-defined parameters. This section aims to explain all options of the ml_pipeline_function. To execute the following examples, create a Python file within the hephaIstos directory and proceed to run it.

The first parameter is mandatory, as you need to provide a DataFrame with a variable to predict that we call “Target.”

The rest of the parameters are optional; you can ignore them depending on what you need to run. If you provide a DataFrame defined as df, then you can apply the following options:

- **Save results**

- output_folder: To save figures, results, or inference models to an output folder, set the path of an output folder to where you want to save the results of the pipeline (such as the metrics accuracy of the models) in a .csv file.
- Let us take an example. Create a Python file in hephaIstos, write the following lines, and execute it:

```
from ml_pipeline_function import ml_pipeline_function

# Import dataset
from data.datasets import neurons
df = neurons() # Load the neurons dataset

# Run ML Pipeline
ml_pipeline_function(df, output_folder='./Outputs/')
# Execute the ML pipeline with the loaded dataset and store the output in the './Outputs/' folder
```

- **Split the data into training and testing datasets**

- test_size: If the dataset is not a time series dataset, you can set the amount of data you want to use for testing purposes. For example, if test_size = 0.2, it means that you take 20% of the data for testing.
- Example:

```
from ml_pipeline_function import ml_pipeline_function

# Import dataset
from data.datasets import neurons
df = neurons() # Load the neurons dataset

# Run ML Pipeline with row removal for handling missing values and 20% test set size
ml_pipeline_function(df, output_folder='./Outputs/', test_size=0.2)
# Execute the ML pipeline with the loaded dataset, split the data into train and test
sets with a test size of 20%, and store the output in the './Outputs/' folder
```

- test_time_size: If the dataset is a time series dataset, we do not use test_size but test_time_size instead. If we choose test_time_size = 1000, it will take the last 1000 values of the dataset for testing.
- time_feature_name: This is the name of the feature containing the time series.

- time_split: This is used to split the time variable by year, month, minutes, and seconds as described in <https://pandas.pydata.org/docs/reference/api/pandas.Series.dt.year.html>. Available options are “year,” “month,” “hour,” “minute,” “second.”
- time_format: This is the strftime to parse time, for example, “%d/%m/%Y.” See the strftime documentation for more information and the different options:
<https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior>. For example, if the data are “1981-7-1 15:44:31,” a format would be “%Y-%d-%m %H:%M:%S.”
- Example:

```
from ml_pipeline_function import ml_pipeline_function

# Import Dataset
from data.datasets import DailyDelhiClimateTrain
df = DailyDelhiClimateTrain() # Load the DailyDelhiClimateTrain dataset
df = df.rename(columns={"meantemp": "Target"}) # Rename the 'meantemp' column to 'Target'

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_time_size=365, # Set the test time size to 365 days
    time_feature_name='date', # Set the time feature name to 'date'
    time_format="%Y-%m-%d", # Define the time format as "%Y-%m-%d"
    time_split=['year', 'month', 'day']) # Split the time feature into 'year', 'month', and 'day' columns
)

# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 365 days and splitting the 'date' feature into 'year', 'month',
and 'day' columns.
```

• Time series transformation

- time_transformation: To transform time series data, we can use different techniques such as lag, rolling window, or expanding window. For example, to use lag, we need to set the time_transformation as follows: time_transformation = ‘lag’.
- If lag is selected, we need to add the following parameters:
 - number_of_lags: an integer defining the number of lags we want.
 - lagged_features: to select features for which we want to apply lag.
 - lag_aggregation: to select the aggregation method. For aggregation, the following options are available: “min,” “max,” “mean,” “std,” or “no.” Several options can be selected at the same time.
- If rolling_window is selected, we need to add the following parameters:
 - window_size: an integer indicating the size of the window.
 - rolling_features: to select the features for which we want to apply rolling window.
- If expanding_window is selected, we need to add the following parameters:
 - expanding_window_size: an integer indicating the size of the window
 - expanding_features: to select the features for which we want to apply to the expanding rolling window.

- Example:

```

from ml_pipeline_function import ml_pipeline_function

from data.datasets import DailyDelhiClimateTrain
df = DailyDelhiClimateTrain() # Load the DailyDelhiClimateTrain dataset
df = df.rename(columns={"meantemp": "Target"}) # Rename the 'meantemp' column to 'Target'

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_time_size=365, # Set the test time size to 365 days
    time_feature_name='date', # Set the time feature name to 'date'
    time_format="%Y-%m-%d", # Define the time format as "%Y-%m-%d"
    time_split=['year', 'month', 'day'], # Split the time feature into 'year', 'month',
    and 'day' columns
    time_transformation='lag', # Apply the lag transformation
    number_of_lags=2, # Set the number of lags to 2
    lagged_features=['wind_speed', 'meanpressure'], # Set the lagged features to
    'wind_speed' and 'meanpressure'
    lag_aggregation=['min', 'mean'] # Set the aggregation methods for the lagged
    features to 'min' and 'mean'
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 365 days, splitting the 'date' feature into 'year', 'month', and
# 'day' columns,
# applying the lag transformation with 2 lags for 'wind_speed' and 'meanpressure', and
# aggregating them with 'min' and 'mean'.

```

• Categorical data

- categorical: If the dataset is composed of categorical data that are labeled with text, we can select data encoding methods. The following options are available: “ordinal_encoding,” “one_hot_encoding,” “label_encoding,” “helmert_encoding,” “binary_encoding,” “frequency_encoding,” “mean_encoding,” “sum_encoding,” “weightofevidence_encoding,” “probability_ratio_encoding,” “hashing_encoding,” “backward_difference_encoding,” “leave_one_out_encoding,” “james_stein_encoding,” and “m_estimator_encoding.” Different encoding methods can be combined.
- We need to select the features that we want to encode with the specified method. For this, we indicate the features we want to encode for each method:
 - features_ordinal
 - features_one_hot
 - features_label
 - features_helmert
 - features_binary
 - features_frequency
 - features_mean
 - features_sum
 - features_weight
 - features_proba_ratio
 - features_hashing
 - features_backward

- features_leave_one_out
- features_james_stein
- features_m

- Example:

```
from ml_pipeline_function import ml_pipeline_function

from data.datasets import insurance
df = insurance() # Load the insurance dataset
df = df.rename(columns={"charges": "Target"}) # Rename the 'charges' column to
'Target'

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['binary_encoding', 'label_encoding'], # Apply binary and label encoding
    for categorical features
    features_binary=['smoker', 'sex'], # Apply binary encoding for 'smoker' and 'sex'
    features
    features_label=['region'] # Apply label encoding for the 'region' feature
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, and applying binary encoding for 'smoker' and 'sex'
# features and label encoding for the 'region' feature.
```

• Data rescaling

- rescaling: We can include a data rescaling method. The following options are available:

- standard_scaler
- minmax_scaler
- maxabs_scaler
- robust_scaler
- normalizer
- log_transformation
- square_root_transformation
- reciprocal_transformation
- box_cox
- yeo_johnson
- quantile_gaussian
- quantile_uniform

- Example:

```
from ml_pipeline_function import ml_pipeline_function

# Import Data
from data.datasets import neurons
df = neurons() # Load the neurons dataset
```

```
# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler' # Apply standard scaling to the features
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, and
rescaling the features using the standard scaler.
```

- **Feature extraction**

- feature_extraction: This option selects the method of feature extraction. The following choices are available:
 - pca
 - ica
 - icawithpca
 - lda_extraction
 - random_projection
 - truncatedSVD
 - isomap
 - standard_lle
 - modified_lle
 - hessian_lle
 - ltsa_lle
 - mds
 - spectral
 - tsne
 - nca
- number_components: This is the number of principal components we want to keep for PCA, ICA, LDA, or other purposes.
- n_neighbors: This is the number of neighbors to consider for manifold learning techniques.
- Example:

```
from ml_pipeline_function import ml_pipeline_function

# Import Data
from data.datasets import neurons
df = neurons() # Load the neurons dataset

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
```

```

features_label=['Target'], # Apply label encoding for the 'Target' feature
rescaling='standard_scaler', # Apply standard scaling to the features
features_extraction='pca', # Apply Principal Component Analysis (PCA) for feature
extraction
number_components=2 # Set the number of PCA components to 2
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing PCA feature extraction with 2 components.

```

• Feature selection

- feature_selection: Here we can select a feature selection method (filter, wrapper, or embedded):
 - The following filter options are available:
 - variance_threshold: Apply a variance threshold. If we choose this option, we also need to indicate the features we want to process (features_to_process= ['feature_1', 'feature_2', ...]) and the threshold (var_threshold = 0 or any number).
 - chi2: Perform a chi-squared test on the samples and retrieve only the k-best features. We can define k with the k_features parameter.
 - anova_f_c: Create a SelectKBest object to select features with the k-best ANOVA F-values for classification. We can define k with the k_features parameter.
 - anova_f_r: Create a SelectKBest object to select features with the k-best ANOVA F-values for regression. We can define k with the k_features parameter.
 - pearson: The main idea for feature selection is to retain the variables that are highly correlated with the target and keep features that are uncorrelated among themselves. The Pearson correlation coefficient between features is defined by cc_features and that between features and the target by cc_target.
- Examples:

```

from ml_pipeline_function import ml_pipeline_function

# Import Data
from data.datasets import neurons
df = neurons() # Load the neurons dataset

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    feature_selection='pearson', # Apply Pearson correlation-based feature selection
    cc_features=0.7, # Set the correlation coefficient threshold for pairwise feature
    correlation to 0.7
    cc_target=0.7 # Set the correlation coefficient threshold for correlation with the
    target variable to 0.7
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,

```

```
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing feature selection based on Pearson correlation with thresholds of 0.7
for pairwise feature correlation and correlation with the target variable.
```

or

```
from ml_pipeline_function import ml_pipeline_function

# Import Data
from data.datasets import neurons
df = neurons() # Load the neurons dataset

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    feature_selection='anova_f_c', # Apply ANOVA F-test based feature selection
    k_features=2 # Select the top 2 features based on their F-scores
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing feature selection based on ANOVA F-test, selecting the top 2 features.
```

- Wrapper methods: The following options are available for feature_selection: “forward_stepwise,” “backward_elimination,” and “exhaustive.”
 - wrapper_classifier: In wrapper methods, we need to select a classifier or regressor. Here, we can choose one from scikit-learn such as KneighborsClassifier(), RandomForestClassifier, LinearRegression, or others and apply it to forward stepwise (forward_stepwise), backward elimination (backward_elimination), or exhaustive (exhaustive) methods.
 - min_features and max_features are attributes for exhaustive option to specify the minimum and maximum number of features we desire in the combination.
- Example:

```
from ml_pipeline_function import ml_pipeline_function
from sklearn.neighbors import KNeighborsClassifier
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function()
```

```

df,
output_folder='./Outputs/', # Store the output in the './Outputs/' folder
missing_method='row_removal', # Remove rows with missing values
test_size=0.2, # Set the test set size to 20% of the dataset
categorical=['label_encoding'], # Apply label encoding for categorical features
features_label=['Target'], # Apply label encoding for the 'Target' feature
rescaling='standard_scaler', # Apply standard scaling to the features
feature_selection='backward_elimination', # Apply backward elimination for feature selection
wrapper_classifier=KNeighborsClassifier(), # Use K-nearest neighbors classifier for the wrapper method in backward elimination
k_features=2 # Select the top 2 features
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling the features using the standard scaler,
# and performing feature selection using backward elimination with a K-nearest neighbors classifier.

```

- Embedded methods:
 - feature_selection: We can select several methods.
- lasso: If we choose lasso, we need to add the alpha parameter (lasso_alpha).
- feat_reg_ml: Allows selection of features with regularization embedded into machine learning algorithms. We need to select the machine learning algorithms (in scikit-learn) by setting the parameter ml_penalty:
 - embedded_linear_regression
 - embedded_logistic_regression
 - embedded_decision_tree_regressor
 - embedded_decision_tree_classifier
 - embedded_random_forest_regressor
 - embedded_random_forest_classifier
 - embedded_permutation_regression
 - embedded_permutation_classification
 - embedded_xgboost_regression
 - embedded_xgboost_classification
- Example:

```

from ml_pipeline_function import ml_pipeline_function
from sklearn.svm import LinearSVC
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
)

```

```

features_label=['Target'], # Apply label encoding for the 'Target' feature
rescaling='standard_scaler', # Apply standard scaling to the features
feature_selection='feat_reg_ml', # Apply feature selection using a machine learning model
ml_penalty=LinearSVC(
    C=0.05, # Apply a regularization strength of 0.05
    penalty='l1', # Apply L1 regularization
    dual=False, # Use the primal form of the SVM problem
    max_iter=5000 # Set the maximum number of iterations to 5000
)
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing feature selection using a machine learning model with a LinearSVC
with specified parameters.

```

- **Classification algorithms**

- Classification_algorithms: The following classification algorithms are used only with CPUs:
 - svm_linear
 - svm_rbf
 - svm_sigmoid
 - svm_poly
 - logistic_regression
 - lda
 - qda
 - gnb
 - mnb
 - k-neighbors
 - For k-neighbors, we need to add an additional parameter that indicates the number of neighbors (n_neighbors).
 - sgd
 - nearest_centroid
 - decision_tree
 - random_forest
 - For random_forest, we can optionally add the number of estimators (n_estimators_forest).
 - extra_trees
 - For extra_trees, we add the number of estimators (n_estimators_forest).
 - mlp_neural_network
 - The following parameters are available: max_iter, hidden_layer_sizes, activation, solver, alpha, learning_rate, learning_rate_init.
 - max_iter: The maximum number of iterations (default = 200).
 - hidden_layer_sizes: The ith element represents the number of neurons in the ith hidden layer.
 - mlp_activation: The activation function for the hidden layer ("identity," "logistic," "relu," "softmax," "tanh"). The default is "relu."
 - solver: The solver for weight optimization ("lbfgs," "sgd," "adam"). The default is "adam."
 - alpha: The strength of the L2 regularization term (default = 0.0001).
 - mlp_learning_rate: The learning rate schedule for weight updates ("constant," "invscaling," "adaptive"). The default is "constant."
 - learning_rate_init: The initial learning rate used (for sgd or adam). It controls the step size in updating the weights.

- mlp_neural_network_auto
- For each classification algorithm, we also need to add the number of k-folds for cross-validation (cv).
- Example:

```

from ml_pipeline_function import ml_pipeline_function
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    classification_algorithms=[
        'svm_rbf', # Apply Support Vector Machine with radial basis function kernel
        'lda', # Apply Linear Discriminant Analysis
        'random_forest' # Apply Random Forest Classifier
    ],
    n_estimators_forest=100, # Set the number of trees in the random forest to 100
    cv=5 # Perform 5-fold cross-validation
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
# the features using the standard scaler,
# and performing classification using SVM with RBF kernel, LDA, and Random Forest with
# the specified parameters.

```

The code above will print the steps of the processes and provide the metrics of our models such as the following:

	SVM_rbf	lda	random_forest
Rescaling Method	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal
Extraction Method	None	None	None
Accuracy	0.982456	0.938596	0.95614
Precision	0.982456	0.938596	0.95614
Recall	0.982456	0.938596	0.95614
F1 Score	0.982456	0.938596	0.95614
Cross-validation mean	0.975824	0.947253	0.958242
Cross-validation std	0.012815	0.018906	0.015541

- Classification algorithms that use GPUs:

- gpu_logistic_regression: We need to add parameters for use with gpu_logistic_regression:
 - gpu_logistic_optimizer: The model optimizers such as stochastic gradient descent (SGD[learning_rate = 1e-2]), adam (“adam”), or RMSprop (“RMSprop”).
 - gpu_logistic_loss: The loss functions, such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
 - gpu_logistic_epochs: The number of epochs.

- gpu_mlp: We need to add parameters to use gpu_mlp:
 - o gpu_mlp_optimizer: The model optimizers such as stochastic gradient descent (SGD[learning_rate = 1e-2]), adam (“adam”), or RMSprop (“RMSprop”).
 - o gpu_mlp_activation: The activation functions such as softmax, sigmoid, linear, or tanh.
 - o gpu_mlp_loss: The loss functions such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
 - o gpu_mlp_epochs: The number of epochs.
- gpu_rnn: Recurrent neural network for classification. We need to set the following parameters:
 - o rnn_units: The dimensionality of the output space (positive integer).
 - o rnn_activation: The activation function to use (softmax, sigmoid, linear, or tanh).
 - o rnn_optimizer: The optimizer (adam, sgd, or RMSprop).
 - o rnn_loss: The loss function, such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
 - o rnn_epochs: The number of epochs (integer).
- Example:

```
from ml_pipeline_function import ml_pipeline_function
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    classification_algorithms=[
        'svm_rbf', # Apply Support Vector Machine with radial basis function kernel
        'lda', # Apply Linear Discriminant Analysis
        'random_forest', # Apply Random Forest Classifier
        'gpu_logistic_regression' # Apply GPU-accelerated Logistic Regression
    ],
    n_estimators_forest=100, # Set the number of trees in the random forest to 100
    gpu_logistic_activation='adam', # Set the activation function for GPU Logistic
    Regression to 'adam'
    gpu_logistic_optimizer='adam', # Set the optimizer for GPU Logistic Regression to
    'adam'
    gpu_logistic_epochs=50, # Set the number of training epochs for GPU Logistic
    Regression to 50
    cv=5 # Perform 5-fold cross-validation
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing classification using SVM with RBF kernel, LDA, Random Forest, and
GPU-accelerated Logistic Regression with the specified parameters.
```

The code above will print the steps of the processes and provide the metrics of our models such as the following:

	SVM_rbf	lda	random_forest	gpu_logistic_regression
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal	row_removal
Extraction Method	None	None	None	None
Accuracy	0.982456	0.938596	0.964912	0.982456
Precision	0.982456	0.938596	0.964912	0.982456
Recall	0.982456	0.938596	0.964912	0.982456
F1 Score	0.982456	0.938596	0.964912	0.982456
Cross-validation mean	0.975824	0.947253	0.956044	NaN
Cross-validation std	0.012815	0.018906	0.014906	NaN

Below is another example with the SGD optimizer:

```
from ml_pipeline_function import ml_pipeline_function
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    classification_algorithms=[
        'svm_rbf', # Apply Support Vector Machine with radial basis function kernel
        'lda', # Apply Linear Discriminant Analysis
        'random_forest', # Apply Random Forest Classifier
        'gpu_logistic_regression' # Apply GPU-accelerated Logistic Regression
    ],
    n_estimators_forest=100, # Set the number of trees in the random forest to 100
    gpu_logistic_optimizer='adam', # Set the optimizer for GPU Logistic Regression to
    'adam'
    gpu_logistic_epochs=50, # Set the number of training epochs for GPU Logistic
    Regression to 50
    gpu_logistic_loss = 'mse', # Set the loss functions, such as the mean squared error
    ("mse"), the binary logarithmic loss ("binary_crossentropy"), or the multi-class
    logarithmic loss ("categorical_crossentropy")
    cv=5 # Perform 5-fold cross-validation
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
the features using the standard scaler,
# and performing classification using SVM with RBF kernel, LDA, Random Forest, and
GPU-accelerated Logistic Regression with the specified parameters.
```

The code provides the following metrics at the end:

	SVM_rbf	lda	random_forest	gpu_logistic_regression
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal	row_removal
Extraction Method	None	None	None	None
Accuracy	0.982456	0.938596	0.964912	0.991228
Precision	0.982456	0.938596	0.964912	0.991228
Recall	0.982456	0.938596	0.964912	0.991228
F1 Score	0.982456	0.938596	0.964912	0.991228
Cross-validation mean	0.975824	0.947253	0.958242	NaN
Cross-validation std	0.012815	0.018906	0.020382	NaN

- Classification algorithms that use QPUs:
 - We use a default encoding function (`q_kernel_default`) and five encoding functions presented by Suzuki et al. in 2020 (`q_kernel_8`, `q_kernel_9`, `q_kernel_10`, `q_kernel_11`, and `q_kernel_12`) and apply SVC from scikit-learn.
 - We also use a default encoding function (`q_kernel_default_pegasos`) and five encoding functions presented by Suzuki et al. (`q_kernel_8_pegasos`, `q_kernel_9_pegasos`, `q_kernel_10_pegasos`, `q_kernel_11_pegasos`, and `q_kernel_12_pegasos`) and apply the Pegasos algorithm from Shalev-Shwartz.
 - We can also establish the `QuantumKernel` class to calculate a kernel matrix using the `ZZFeatureMap` (`q_kernel_zz`) with SVC from scikit-learn or the Pegasos algorithm (`q_kernel_zz_pegasos`).
 - The following algorithms can be selected: `q_kernel_default`, `q_kernel_8`, `q_kernel_9`, `q_kernel_10`, `q_kernel_11`, `q_kernel_12`, `q_kernel_default_pegasos`, `q_kernel_8_pegasos`, `q_kernel_9_pegasos`, `q_kernel_10_pegasos`, `q_kernel_11_pegasos`, `q_kernel_12_pegasos`, and `q_kernel_zz_pegasos`.
 - Neural networks are also available, as follows: `q_samplerqnn`, `q_estimatorqnn`, and `q_vqc`.
 - We also use `q_kernel_training`. It is possible to train a quantum kernel with quantum kernel alignment (QKA) that iteratively adapts a parameterized quantum kernel to a dataset and converges to the maximum SVM margin at the same time. To implement it, we prepare the dataset as usual and define the quantum feature map. Then, we use `QuantumKernelTrained.fit` method to train the kernel parameters and pass it to a machine learning model. Here, we also need to adapt several parameters that are in the code (see `classification_qpu.py` in the classification folder) such as the following:
 - Setup of the optimizer:

```
spsa_opt = SPSA(maxiter=10, callback=cb_qkt.callback, learning_rate=0.05,
perturbation=0.05)
```

Selection of the rotational layer to train and the number of circuits:

```
# Create a rotational layer to train. We will rotate each qubit the same amount.
training_params = ParameterVector("θ", 1)
fm0 = QuantumCircuit(feature_dimension)
for qubit in range(feature_dimension):
    fm0.ry(training_params[0], qubit)
```

- The following are inputs for running quantum algorithms:
 - `reps` = the number of times the feature map circuit is repeated.
 - `ibm_account` = none.
 - `quantum_backend`: depends on credentials. If this option is not provided, hephaIstos will select automatically a local simulator. Some examples of available online backends (both simulator and real hardware):
 - `ibmq_qasm_simulator`
 - `ibmq_lima`
 - `ibmq_belem`

- ibmq_quito
 - simulator_statevector
 - simulator_extended_stabilizer
 - simulator_stabilizer
 - ibmq_manila
- multi-class: We can use “OneVsRestClassifier,” “OneVsOneClassifier,” and “svc” if we want to pass our quantum kernel to SVC from scikit-learn or “None” if we wish to use QSVC from Qiskit.
- For Pegasos algorithms:
- n_steps = the number of steps performed during the training procedure.
 - C = the regularization parameter.
- Example:

```
from ml_pipeline_function import ml_pipeline_function
from data.datasets import breastcancer

# Import Data
df = breastcancer() # Load the breast cancer dataset
df = df.drop(["id"], axis=1) # Drop the 'id' column

# Run ML Pipeline
ml_pipeline_function(
    df,
    output_folder='./Outputs/', # Store the output in the './Outputs/' folder
    missing_method='row_removal', # Remove rows with missing values
    test_size=0.2, # Set the test set size to 20% of the dataset
    categorical=['label_encoding'], # Apply label encoding for categorical features
    features_label=['Target'], # Apply label encoding for the 'Target' feature
    rescaling='standard_scaler', # Apply standard scaling to the features
    features_extraction='pca', # Apply Principal Component Analysis for feature extraction
    classification_algorithms=['svm_linear'], # Apply Support Vector Machine with a
    linear kernel
    number_components=2, # Set the number of principal components to 2
    cv=5, # Perform 5-fold cross-validation
    quantum_algorithms=[
        'q_kernel_default',
        'q_kernel_zz',
        'q_kernel_8',
        'q_kernel_9',
        'q_kernel_10',
        'q_kernel_11',
        'q_kernel_12'
    ], # List of quantum algorithms to use
    reps=2, # Set the number of repetitions for the quantum circuits
    ibm_account=YOUR_API, # Replace with your IBM Quantum API key
    quantum_backend='qasm_simulator' # Use the QASM simulator as the quantum backend
)
# Execute the ML pipeline with the loaded dataset, removing rows with missing values,
# using a test set of 20%, applying label encoding for the 'Target' feature, rescaling
# the features using the standard scaler,
# and performing classification using SVM with a linear kernel, PCA for feature
# extraction, and quantum algorithms with the specified parameters.
```

We can also choose “least_busy” as a quantum_backend option in order to execute the algorithms on the chip that has the lower number of jobs in the queue:

```
quantum_backend = 'least_busy'
```

- Regression algorithms:

- Regression algorithms used only with CPUs:
 - linear_regression
 - svr_linear
 - svr_rbf
 - svr_sigmoid
 - svr_poly
 - mlp_regression
 - mlp_auto_regression

- Regression algorithms that use GPUs, if available:

- gpu_linear_regression: Linear regression using the SGD optimizer. As for classification, we need to add some parameters:
 - gpu_linear_activation: “Linear.”
 - gpu_linear_epochs: An integer to define the number of epochs.
 - gpu_linear_learning_rate: The learning rate for the SGD optimizer.
 - gpu_linear_loss: The loss functions such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
- gpu_mlp_regression: Multi-layer perceptron neural network using GPUs for regression, with the following parameters to set:
 - gpu_mlp_epochs_r: The number of epochs with an integer.
 - gpu_mlp_activation_r: The activation function such as softmax, sigmoid, linear, or tanh.
 - The chosen optimizer is “adam.” Note that no activation function is used for the output layer because it is a regression. We use mean_squared_error for the loss function.
- gpu_rnn_regression: Recurrent neural network for regression. We need to set the following parameters:
 - rnn_units: The dimensionality of the output space (positive integer).
 - rnn_activation: The activation function to use (softmax, sigmoid, linear, or tanh).
 - rnn_optimizer: The optimizer (adam, sgd, or RMSprop).
 - rnn_loss: The loss function such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
 - rnn_epochs: The number of epochs (integer).

- Example:

```
# Import the required modules
from ml_pipeline_function import ml_pipeline_function
from data.datasets import breastcancer

# Load the breast cancer dataset
df = breastcancer()

# Drop the 'id' column from the dataset as it is not relevant for analysis
df = df.drop(["id"], axis=1)

# Call the machine learning pipeline function with the following parameters:
# - DataFrame (df)
# - Output folder for saving results ('./Outputs/')
```

```

# - Method for handling missing values ('row_removal')
# - Test set size (0.2 or 20%)
# - Encoding method for categorical variables ('label_encoding')
# - Column name for target variable in the dataset (['Target'])
# - Data rescaling method ('standard_scaler')
# - Regression algorithms to be applied (['linear_regression', 'svr_linear',
'svr_rbf', 'gpu_rnn_regression'])
# - Number of epochs for RNN training (50)
# - Activation function for RNN ('linear')
# - Optimizer for RNN ('adam')
# - Number of units in RNN (500)
# - Loss function for RNN ('mse')
ml_pipeline_function(
    df,
    output_folder='./Outputs/',
    missing_method='row_removal',
    test_size=0.2,
    categorical=['label_encoding'],
    features_label=['Target'],
    rescaling='standard_scaler',
    regression_algorithms=['linear_regression', 'svr_linear', 'svr_rbf',
'gpu_rnn_regression'],
    rnn_epochs=50,
    rnn_activation='linear',
    rnn_optimizer='adam',
    rnn_units=500,
    rnn_loss='mse'
)

```

The few lines of code above will print the steps of the processes and provide the metrics of our models such as the following:

	linear_regression	gpu_linear_regression
Rescaling Method	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal
Extraction Method	None	None
MSE	0.051602	0.068005
R-squared	0.778978	0.708721

	svr_linear	svr_rbf
Rescaling Method	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal
Extraction Method	None	None
MSE	0.055885	0.016993
R-squared	0.760633	0.927214

Below is another example with an RNN:

```

# Import the required modules
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Load the Daily Delhi Climate Train dataset
DailyDelhiClimateTrain = './data/datasets/DailyDelhiClimateTrain.csv'

```

```

df = pd.read_csv(DailyDelhiClimateTrain, delimiter=',',)

# Convert the 'date' column to datetime format
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')

# Extract the year, month, and day from the 'date' column and create new columns for
# each
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day

# Drop the 'date' column
df.drop('date', inplace=True, axis=1)

# Rename the 'meantemp' column to 'Target'
df = df.rename(columns={"meantemp": "Target"})

# Drop rows with at least one missing value
df = df.dropna()

# Run the Machine Learning (ML) pipeline function with the following parameters:
# - Dataframe: df
# - Output folder: './Outputs/'
# - Missing data handling method: 'row_removal'
# - Test dataset size: 20% of the total dataset
# - Data rescaling method: 'standard_scaler'
# - Regression algorithm to be used: 'gpu_rnn_regression'
# - GPU RNN Regression specific parameters:
#   - Number of units: 500
#   - Activation function: 'tanh'
#   - Optimizer: 'RMSprop'
#   - Loss function: 'mse'
#   - Number of training epochs: 50
ml_pipeline_function(
    df,
    output_folder='./Outputs/',
    missing_method='row_removal',
    test_size=0.2,
    rescaling='standard_scaler',
    regression_algorithms=['gpu_rnn_regression'],
    rnn_units=500,
    rnn_activation='tanh',
    rnn_optimizer='RMSprop',
    rnn_loss='mse',
    rnn_epochs=50,
)

```

- **Convolutional neural networks**

- conv2d: two-dimensional convolutional neural network (CNN) using GPUs if available. The parameters are the following:
 - conv_kernel_size: The kernel_size is the size of the filter matrix for the convolution ($\text{conv_kernel_size} \times \text{conv_kernel_size}$).
 - conv_activation: The activation function to use (softmax, sigmoid, linear, relu, or tanh)
 - conv_optimizer: The optimizer (adam, sgd, RMSprop).
 - conv_loss: The loss function, such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”).
 - conv_epochs: The number of epochs (integer).
- Example:

```
# Import the ml_pipeline_function from the external module
from ml_pipeline_function import ml_pipeline_function
# Import pandas for data manipulation
import pandas as pd

# Import TensorFlow library for machine learning and deep learning
import tensorflow as tf
# Import the mnist dataset from the Keras library
from keras.datasets import mnist
# Import to_categorical function to convert integer labels to binary class matrices
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset into a tuple of tuples (train and test data)
df = mnist.load_data()
# Separate the dataset into features (X) and labels (y) for both train and test data
(X, y), (_, _) = mnist.load_data()
# Assign the train and test data to variables
(X_train, y_train), (X_test, y_test) = df

# Reshape the training data to fit the model's input shape (number of images, height,
width, and channels)
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], X_train.shape[2], 1)
# Reshape the testing data to fit the model's input shape
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], X_test.shape[2], 1)
# Reshape the whole dataset to fit the model's input shape
X = X.reshape(X.shape[0], X.shape[1], X.shape[2], 1)

# Call the ml_pipeline_function with the given parameters to train and evaluate a
# convolutional neural network
ml_pipeline_function(df, X, y, X_train, y_train, X_test, y_test, output_folder =
'./Outputs/', convolutional=['conv2d'], conv_activation='relu', conv_kernel_size =
3, conv_optimizer = 'adam', conv_loss='categorical_crossentropy', conv_epochs=1)
```

1.4 Where to Find the Datasets and Code Examples

Almost all the datasets used in this book can be found at the following link:
<https://github.com/xaviervasques/hephaistos/tree/main/data/datasets>.

Another way to use the datasets is to move them to the hephAIstos folder so that you can download or clone them from GitHub (<https://github.com/xaviervasques/hephaistos.git>). Within the hephAIstos folder, the data are located in data/datasets and all code examples are in the Notebooks folder.

If you open a terminal, move to hephAIstos/Notebooks, and type jupyter notebook; your browser will open and you will find all the Jupyter Notebooks that are available for this book. If you open any Jupyter Notebook, the path for the datasets is already defined in the various code examples.

Further Reading

- Chicco, D. and Jurman, G. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics* 21 (6): 6. <https://doi.org/10.1186/s12864-019-6413-7>. PMC 6941312. PMID 31898477.
- Derczynski, L. (2016). Complementarity, F-score, and NLP evaluation. In: *Proceedings of the International Conference on Language Resources and Evaluation*.
- Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters* 27 (8): 861–874. <https://doi.org/10.1016/j.patrec.2005.10.010>.
- Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning Book*. MIT Press. <http://www.deeplearningbook.org>.
- Hand, D. and Christen, P. (2018). A note on using the F-measure for evaluating record linkage algorithms. *Statistics and Computing* 28: 539–547. <https://doi.org/10.1007/s11222-017-9746-6>.
- Madeh, P.S. and El-Diraby Tamer, E. (2020). Data analytics in asset management: cost-effective prediction of the pavement condition index. *Journal of Infrastructure Systems* 26 (1): 04019036. [https://doi.org/10.1061/\(ASCE\)IS.1943-555X.0000512](https://doi.org/10.1061/(ASCE)IS.1943-555X.0000512).
- Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (2011). Scikit-learn: machine learning in Python. *Journal of Machine Learning Research* 12 (85): 2825–2830.
- Powers, D.M.W. (2011). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2 (1): 37–63.
- Christen, P., Hand, D.J., and Kirielle, N. (2023). A review of the F-measure: its history, properties, criticism, and alternatives. *ACM Computing Surveys*. <https://doi.org/10.1145/3606367>.
- Siblini, W., Fréry, J., He-Guelton, L. et al. (2020). Master your metrics with calibration. In: *Advances in Intelligent Data Analysis XVIII* (ed. M. Berthold, A. Feelders, and G. Krempl), 457–469. Springer. https://doi.org/10.1007/978-3-030-44584-3_36.
- Taha, A.A. and Hanbury, A. (2015). Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging* 15 (29): 1–28. <https://doi.org/10.1186/s12880-015-0068-x>.
- Van Rijsbergen, C.J. (1979). *Information Retrieval*, 2e. Butterworth-Heinemann.
- Williams, C.K.I. (2021). The effect of class imbalance on precision-recall curves. *Neural Computation* 33 (4): 853–857. arXiv:2007.01905. https://doi.org/10.1162/neco_a_01362. PMID 33513323.
- https://developer.nvidia.com/cuda-downloads?target_os=Linux&target_arch=x86_64&=&Ubuntu&target_version=20.04&target_type=deb_local
- <https://developer.nvidia.com/cudnn>
- <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- <https://dorianbrown.dev/what-is-supervised-learning/>
- <https://github.com/tensorflow/models/tree/master/official>
- <https://machinelearningmastery.com/k-fold-cross-validation/>
- <https://medium.com/geekculture/installing-cudnn-and-cuda-toolkit-on-ubuntu-20-04-for-machine-learning-tasks-f41985fcf9b2>
- <https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/>
- <https://www.hindawi.com/journals/cmmm/2021/8500314/>
- <https://www.ibm.com/cloud/learn/underfitting>
- <https://www.toolbox.com/tech/artificial-intelligence/articles/top-python-libraries-for-machine-learning/>

2

Feature Engineering Techniques in Machine Learning



Photo by Annamária Borsos

Datasets are samples that are composed of several observations. Each observation is a set of values associated with a set of variables. The transformations we need to perform before starting to train models depend on the nature of features or variables. In the variable's realm, we can find quantitative and qualitative variables. Quantitative variables can be continuous, which means composed of real values, or discrete, which means that they can only take values from within a finite set or an infinite but countable set. Qualitative variables do not translate mathematical magnitudes and can be ordinal, for which an order relationship can be defined, or nominal, where no order relationship can be found between values.

Feature engineering is an important part of the data science workflow that can greatly impact the performance of machine learning algorithms. Anomalies must be corrected or at least not ignored, and we need to adjust for missing values, eliminate duplicate observations, digitalize the data to facilitate the use of machine learning tools, encode categorical data, rescale data, and perform other tasks. For instance, often we can replace a missing value by the mean, but when the number of missing values is important, it can create bias. Instead, we can choose linear regression, which can be more complicated but more helpful as the new values will not be chosen randomly. The detection of outliers is also a good exercise. Some extreme values and errors that are considered normal for the models need to be detected using different techniques such as boxplots. A descriptive analysis of the data is something to consider before going forward in machine learning workflows. We will make some decisions in order to have a dataset that is representative of the real world. In the real world, datasets are not expressed in the same scale, forcing us to perform data standardization, normalization, or scaling to produce comparable scales across our datasets.

Data preprocessing is a crucial step in machine and deep learning. It is indeed integral to machine learning, as the quality of data and the useful information that can be derived from it directly affects the ability of a model to learn; therefore, it is extremely important that we preprocess our data before feeding it into our model. It is necessary to perform a certain number of operations to obtain data of quality, as we will see in this chapter.

2.1 Feature Rescaling: Structured Continuous Numeric Data

Most of the time, we will encounter different types of variables with different ranges that can differ considerably in the same dataset. If we use the data with the original scale, we will certainly put more weight on the variables with a large range. Therefore, we need to apply what is called feature rescaling to ensure that the variables are almost on the same scale, which allows consideration of features as equally important (apples to apples).

Often, we refer to the concepts of standardization, normalization, or scaling. Scaling indicates that we change the range of the values without changing the shape of the distribution. The range is often set at 0 to 1 or -1 to 1. Standardization refers to converting the data distribution into a normal form and transforming the mean of the data to 0 and its variance to 1. Normalization refers to transforming the data into the range [0, 1] or dividing a vector by a norm. Normalization can reflect not only this definition but also others, creating confusion because it has many definitions in statistical learning. Perhaps the best choice is to always define the word we use before any communication.

Many machine learning algorithms perform better or converge faster when features are close to normally distributed and on a relatively similar scale. It is important to transform data for each feature to have a variance equivalent to others in orders of magnitude to avoid dominance of an objective function (such as the radial basis function kernel of support vector machines) that would make the estimator unable to learn correctly. Data scaling is required in algorithms such as those based on gradient descent (linear regression, neural networks) and distance-based algorithms (k-nearest neighbors, k-means, SVM). For example, it is meaningless if we do not standardize our data before measuring the importance of variables in regression models or before lasso and ridge regressions. The reason we standardize data is not the same for all machine learning algorithms. For example, in distance-based models, we perform standardization to prevent features with wider ranges from dominating the distance metric. Some algorithms, including tree-based algorithms such as decision tree, random forest, and gradient boosting, are not sensitive to the magnitude of variables.

We can implement data preprocessing for machine learning using many available methods, such as doing it yourself or using NumPy, SciPy, or those proposed in scikit-learn (MinMaxScaler, RobustScaler, StandardScaler, MaxAbsScaler, Normalizer). If you download hephAistos from GitHub (<https://github.com/xaviervasques/hephaistos.git>), you will find all code examples in hephaistos/Notebooks/Features_rescaling.ipynb.

Before going into the details, let us import some libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use("seaborn")
from scipy.stats import skew
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
from sklearn import preprocessing
import os
```

Nature does not always produce perfectly normal distributions. We will apply the transformations described above with a dataset coming from 46 MRIs (17 healthy controls, 14 patients with disease type 1, 15 patients with disease type 2). Cortical and subcortical features were extracted using Freesurfer, version 7; in total, 474 features were extracted from 3D MRI T1-weighted images. (Freesurfer is a set of tools for analysis and visualization of cortical and subcortical brain imaging data. It is designed around an automated workflow that includes several standard image processing phases.)

To start, we will extract from the 474 features only the “TotalGrayVol” data, which corresponds to the total gray volume of each brain analyzed.

Let us load the data from a .csv file and extract the “TotalGrayVol” feature.

Input:

```
# Load data
csv_data = '../data/datasets/brain_train.csv'
data = pd.read_csv(csv_data, delimiter=';')
df = data[['Target', "TotalGrayVol"]]
y = df.loc[:, data_train.columns == 'Class'].values.ravel()
X = df.loc[:, data_train.columns != 'Class']
print(df.head())
```

Output:

	Target	TotalGrayVol
0	1	684516.128934
1	1	615126.611828
2	1	678687.178551
3	1	638615.189584
4	1	627356.125850

2.1.1 Data Transformation

2.1.1.1 StandardScaler

The result of the dataset standardization (or **Z-score normalization**) is that the features have been rescaled to ensure the mean and the standard deviation to be 0 and 1, respectively. This process is necessary when continuous independent variables have been measured at different scales such as in different measurement units.

The **StandardScaler** removes the mean and scales to unit variance:

$$z = \frac{x - \text{mean}(x)}{\text{standard deviation}(x)}$$

Many machine learning estimators will behave badly if the individual features do not appear as standard, normally distributed data (Gaussian with zero mean and unit variance). Most of the time, data scientists do not look at the shape of the distribution. They simply center data by removing the mean value of each feature and scale it by dividing non-constant values by their standard deviation. The StandardScaler technique assumes that data are normally distributed (Figures 2.1 and 2.2). Essentially, the idea of dataset standardization is to have equal range and variance from a rescaled original variable.

Let us imagine that the data are not normally distributed and that 50% of the data points have a value of 0 and the remaining 50% have a value of 1. Gaussian scaling will send half of the data to -1 and the other half to $+1$; the data are moved away from 0. Depending on the shape of our data, we need to consider other kinds of scaling that will produce better results.

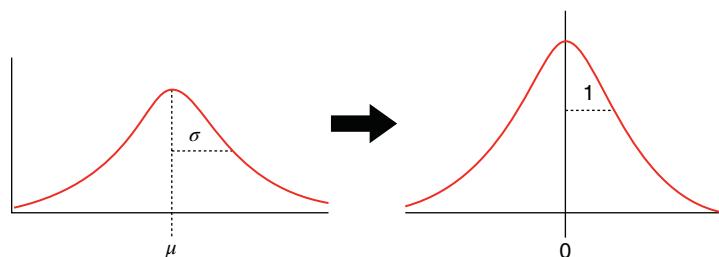


Figure 2.1 In standardization, features are rescaled to ensure that the mean and the standard deviation are 0 and 1, respectively.

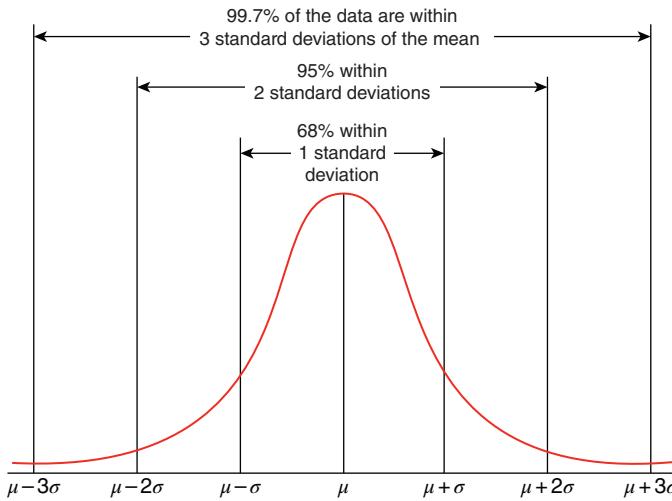


Figure 2.2 The result of this transformation is that we have shifted the means to 0; most of the data (68%) would thus be between -1 and 1.

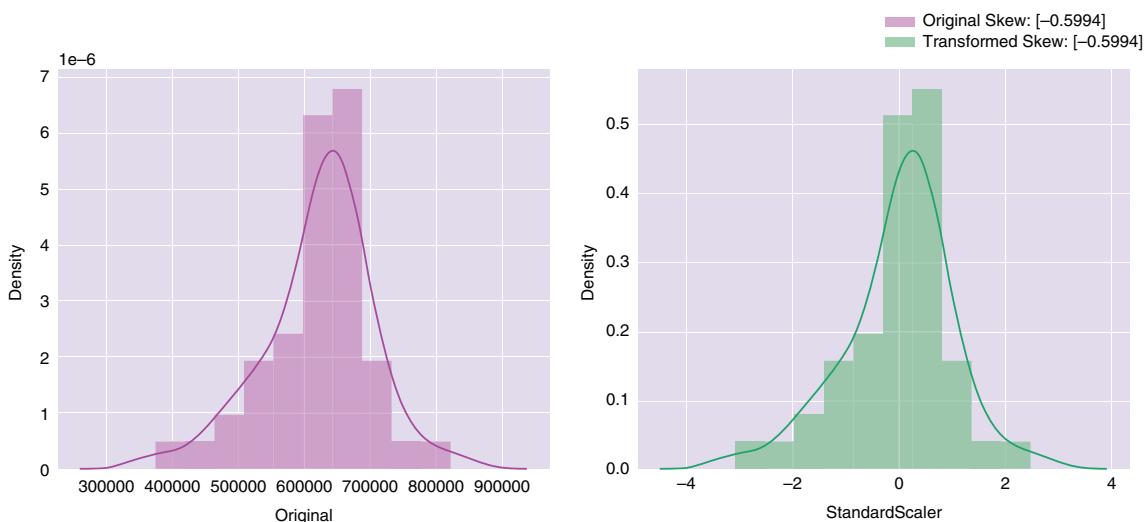
We can code StandardScaler as follows:

Input:

```
StandardScaler = preprocessing.StandardScaler()
stdscaler_transformed = StandardScaler.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X),4)),
color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(stdscaler_transformed, label= "Transformed Skew:{0}".format(np.round(
skew(stdscaler_transformed),4)), color="g", ax=ax[1], xlabel="StandardScaler")
fig.legend()
plt.show()
```

Output:



We can see that the shape of the distribution has not changed. The top of the curve is centered at 0, as expected.

2.1.1.2 MinMaxScaler

We can also transform features by scaling each of them to a defined range (e.g., between -1 and 1 or between 0 and 1). Min-max scaling (**MinMaxScaler**), for instance, can be very useful for some machine learning models. **MinMaxScaler** has some advantages over **StandardScaler** when the data distribution is not Gaussian and the feature falls within a bounded interval, which is typically the case with pixel intensities that fit within a range of 0 – 255 .

MinMaxScaler is calculated as follows:

$$z = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

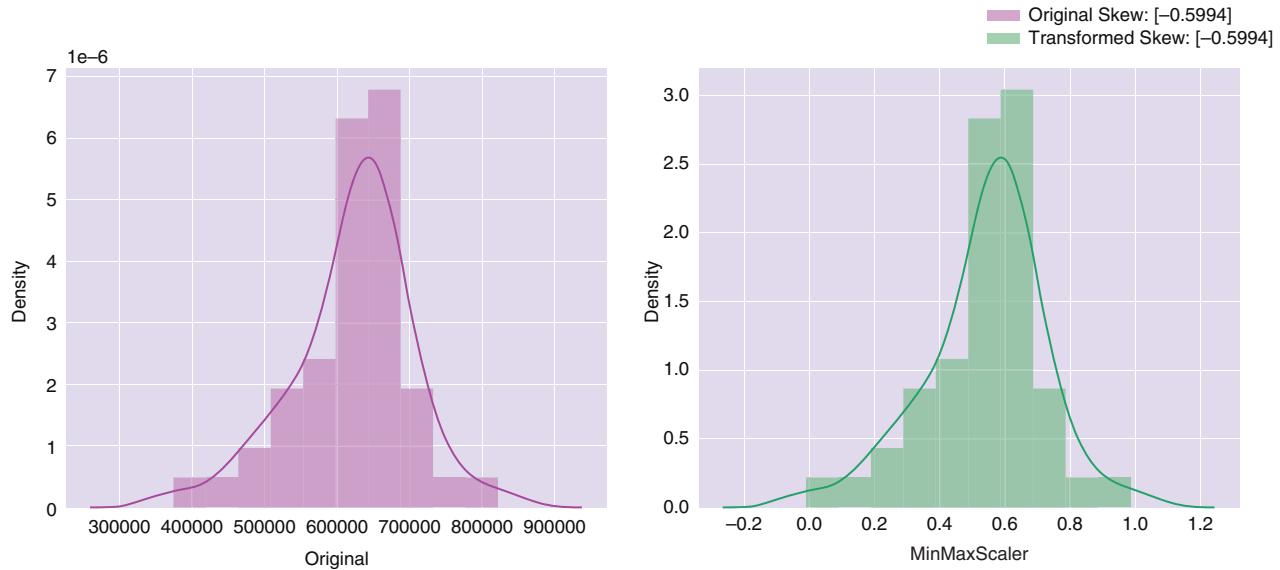
We can code **MinMaxScaler** as follows:

Input:

```
min_max_scaler = preprocessing.MinMaxScaler()
minmax_transformed = min_max_scaler.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X) , 4)) ,
color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(minmax_transformed, label= "Transformed Skew:{0}".format(np.round(skew(minmax_transformed) , 4)) , color="g", ax=ax[1], xlabel="MinMaxScaler")
fig.legend()
plt.show()
```

Output:



By default, the feature range is between 0 and 1 . We can modify this range by adding the option `feature_range`:

```
preprocessing.MinMaxScaler(feature_range=(0, 1))
```

2.1.1.3 MaxAbsScaler

The **MaxAbsScaler** is similar to the MinMaxScaler with the difference that it automatically scales the data between 0 and 1 based on the absolute maximum. This scaler is specifically suitable for data that is already centered at zero or sparse data and does not center the data, maintaining sparsity:

$$z = \frac{x_i}{\max(\text{abs}(x))}$$

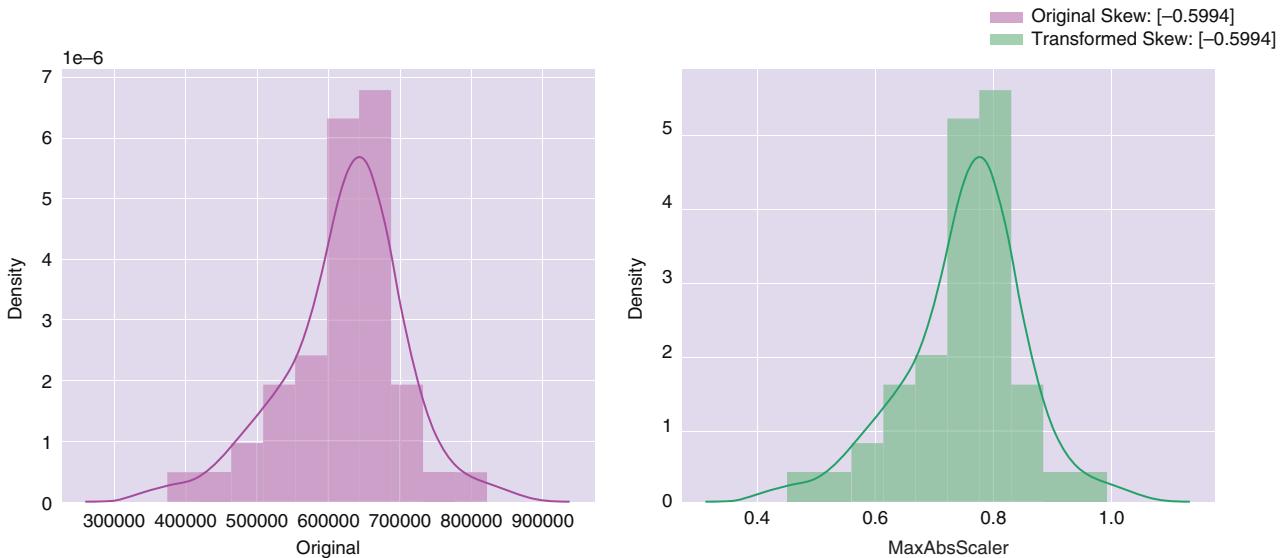
We can code MaxAbsScaler as follows:

Input:

```
max_abs_scaler = preprocessing.MaxAbsScaler()
maxabs_transformed = max_abs_scaler.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X), 4)),
color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(maxabs_transformed, label= "Transformed Skew:{0}".format(np.round(skew(maxabs_transformed),4)), color="g", ax=ax[1], xlabel="MaxAbsScaler")
fig.legend()
plt.show()
```

We will receive the following output:



2.1.1.4 RobustScaler

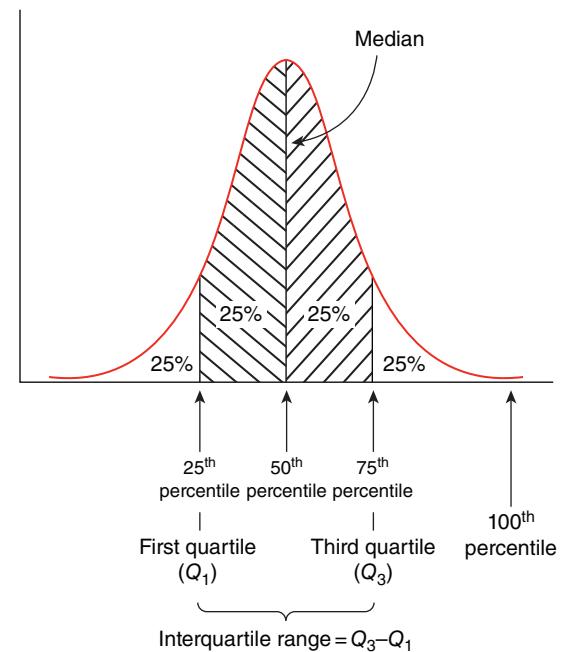
If the data contain a considerable number of outliers, the use of the mean and variance to scale the data will probably not work correctly. In this case, an option is to use **RobustScaler**, which removes the median and scales the data according to the quantile range:

$$z = \frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}$$

As we can see (Figure 2.3), the scaler uses the interquartile range (IQR), which is the range between the first quartile $Q_1(x)$ and the third quartile $Q_3(x)$.

We will see how feature scaling can significantly improve the performance of some machine learning algorithms but not improve it at all for others.

Figure 2.3 The scaler uses the interquartile range (IQR), which is the range between the first quartile $Q_1(x)$ and the third quartile $Q_3(x)$.



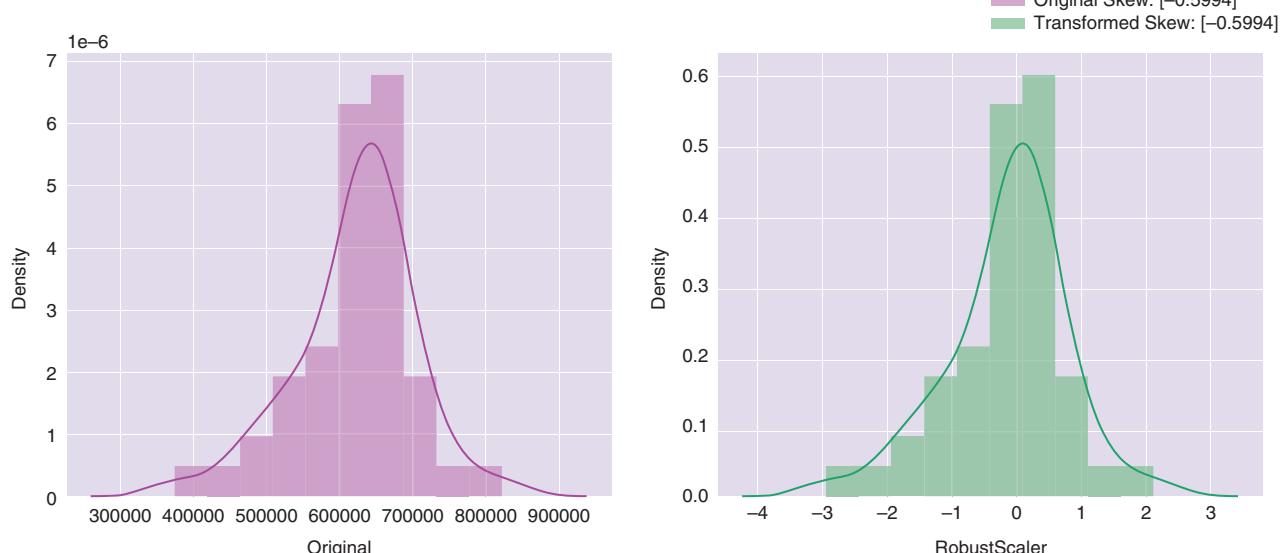
RobustScaler can be implemented as follows:

Input:

```
RobustScaler = preprocessing.RobustScaler()
Robust_transformed = RobustScaler.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X) ,4)) ,
color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(robust_transformed, label= "Transformed Skew:{0}".format(np.round(skew(robust_transformed),4)), color="g", ax=ax[1], xlabel="RobustScaler")
fig.legend()
plt.show()
```

The following output results:



2.1.1.5 Normalizer: Unit Vector Normalization

Normalization is another technique of scaling individual samples to produce a unit norm and is a common operation for clustering or text classification. We need to normalize data when our model makes predictions based on the weighted relationships formed between data points. To provide a mental image, standardization is a column-wise operation while normalization is a row-wise operation. Similar to standardization, we also have different ways to normalize, namely l1, l2, and max.

To understand this technique, it is necessary to understand how to calculate vector norms or magnitudes. Mathematicians often use the term “norm” instead of “length.” The concept of norms is important in machine and deep learning because it is used to evaluate the error of models (the difference between the output of a model and what is expected) or to define a regularization term that includes the magnitude of the weights. Vector norms have important rules:

- The norm of a vector is always positive $\|\mathbf{a}\| \geq 0$ or zero if and only if the vector is a zero vector ($\mathbf{a} = \mathbf{0}$).
- A scalar multiple to a norm is equal to the product of the absolute value of the scalar and the norm: $\text{norm}||k\mathbf{a}|| = |k| \|\mathbf{a}\|$.
- Norm of a vector obeys triangular inequality. The norm of the sum of some vectors is less than or equal to the sum of the norms of these vectors: $\|\mathbf{a} + \mathbf{b}\| \leq \|\mathbf{a}\| + \|\mathbf{b}\|$.

The length of a vector $\|\mathbf{a}\|_1$ can be calculated using the l1 norm as the sum of the absolute value of the vector elements: $\|\mathbf{a}\|_1 = |a_1| + |a_2| + \dots + |a_n|$. The l2 norm, or Euclidian norm, is the most often used vector norm: $\|\mathbf{a}\|_2 = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2}$. The p-norm, called Minkowski norm, is defined as follows: $\|\mathbf{a}\|_p = \sqrt[p]{a_1^p + a_2^p + \dots + a_n^p}$. The max-norm, also called Chebyshev norm, is the largest absolute element in the vector: $\|\mathbf{a}\| = \max[|a_1|, |a_2|, \dots, |a_n|]$.

In machine learning, we usually use the l1 norm when the sparsity of the vector matters. The sparsity corresponds to the property of having highly significant coefficients (either very near to or very far from zero) where the coefficient very near zero could later be eliminated. This technique is an alternative when we are processing a large set of features. One consideration when building a model is its ability to ignore extreme values in the data, in other words, its resistance to outliers in a dataset. The l1 norm is more robust than the l2 norm from this point of view, as it only considers the absolute values; thus, it treats them linearly, whereas the l2 norm increases the cost of outliers exponentially. In an opposite sense, the resistance to horizontal adjustments is more stable with the l2 norm than with the l1 norm. The l1 norm is computationally more expensive than the l2 norm because we cannot solve it in terms of matrix operations.

If \mathbf{x} is the vector of covariates of length n , and we assume that the normalized vector $\mathbf{z} = \mathbf{x}/y$, then three options denote what to use for y :

For the l1 option,

$$y = \|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

For the l2 option,

$$y = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

For the max option,

$$y = \max x_i$$

As stated above, there are different types of normalization. Here, we consider unit vector normalization. Assume we have a dataset X with D columns (features) and N rows (entries). The unit vector normalization of the dataset is calculated as follows:

$$Z_{[j,:]} = \frac{X[j,:]}{\|X[j,:]\|}$$

where $X[j,:]$ represents entry j and $X[:,j]$ represents feature i .

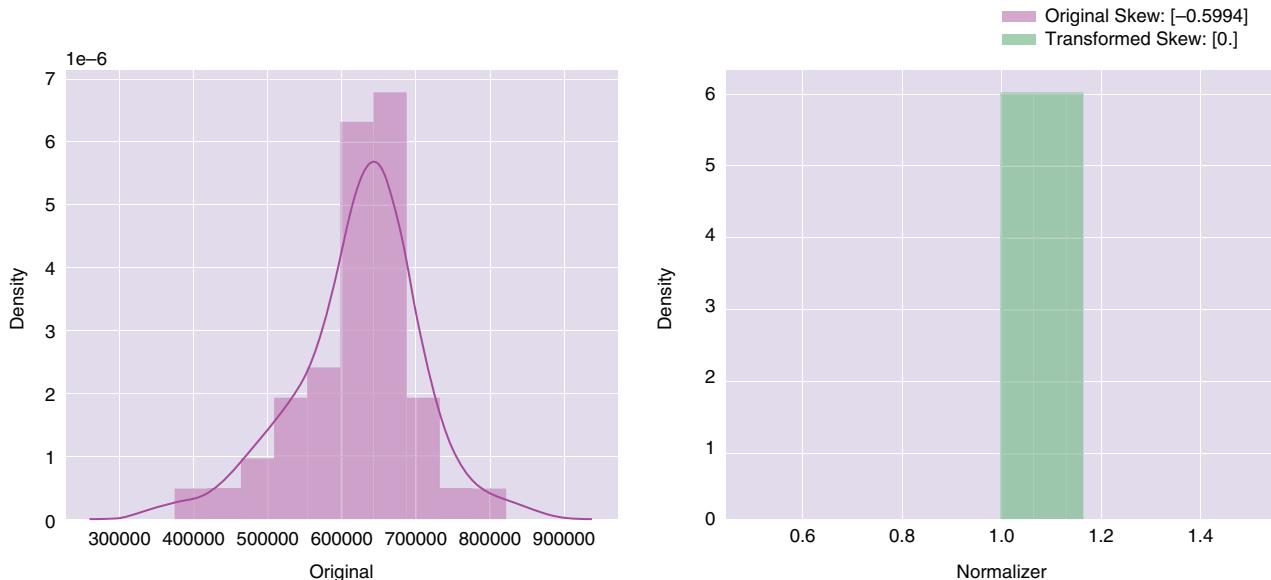
We can code a normalizer using scikit-learn as follows:

Input:

```
Normalize = preprocessing.Normalizer()
Norm_transformed = Normalize.fit(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X),4)),
color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(norm_transformed, label= "Transformed Skew:{0}".format(np.round(skew(
(norm_transformed),4)), color="g", ax=ax[1], xlabel="Normalizer")
fig.legend()
plt.show()
```

The following output results:



In scikit-learn, the normalizer uses l2 by default. We can change this behavior by using the norm option ("l1," "l2," "max"):

```
sklearn.preprocessing.Normalizer(norm='l2', *, copy=True)
```

2.1.1.6 Other Options

There are many more options for data transformation such as logistic, lognormal, or hyperbolic tangent. **In logistic data transformation**, the values in the column are calculated as follows:

$$z = \frac{1}{1 + e^{-x}}$$

Lognormal transformation converts the values to a lognormal scale using the following formula for the cumulative distribution function of the lognormal distribution:

$$F(x) = \phi\left(\frac{\ln(x)}{\sigma}\right), x \geq 0; \sigma \geq 0$$

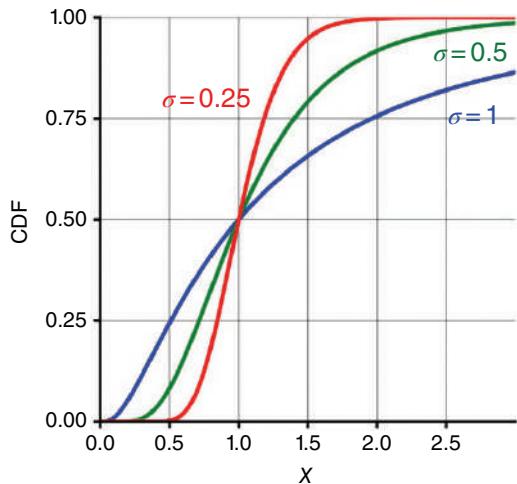


Figure 2.4 Lognormal cumulative distribution functions. Source: https://en.wikipedia.org/wiki/Log-normal_distribution.

where ϕ is the cumulative distribution function of the normal distribution. The cumulative function of the standard normal distribution is the following:

$$F(x) = \int_{-\infty}^x \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$$

The output shown in Figure 2.4 is the plot of lognormal cumulative distribution functions.

2.1.1.7 Transformation to Improve Normal Distribution

As stated above, generating a better fit for normal distribution (Gaussian distribution) is often a key to improving accuracy scores. As we have seen, data transformation is no more than applying a mathematical function to the data. We also need to study data skewness, which is the asymmetry in a statistical distribution; the curve can appear distorted or skewed to either the left or the right. In a normal distribution, the graph is a symmetrical, “bell-shaped” curve. Lower skewness produces a better result. To address highly skewed variables and non-normal distributions that can be due to outliers or highly exponential distributions, we can use the following:

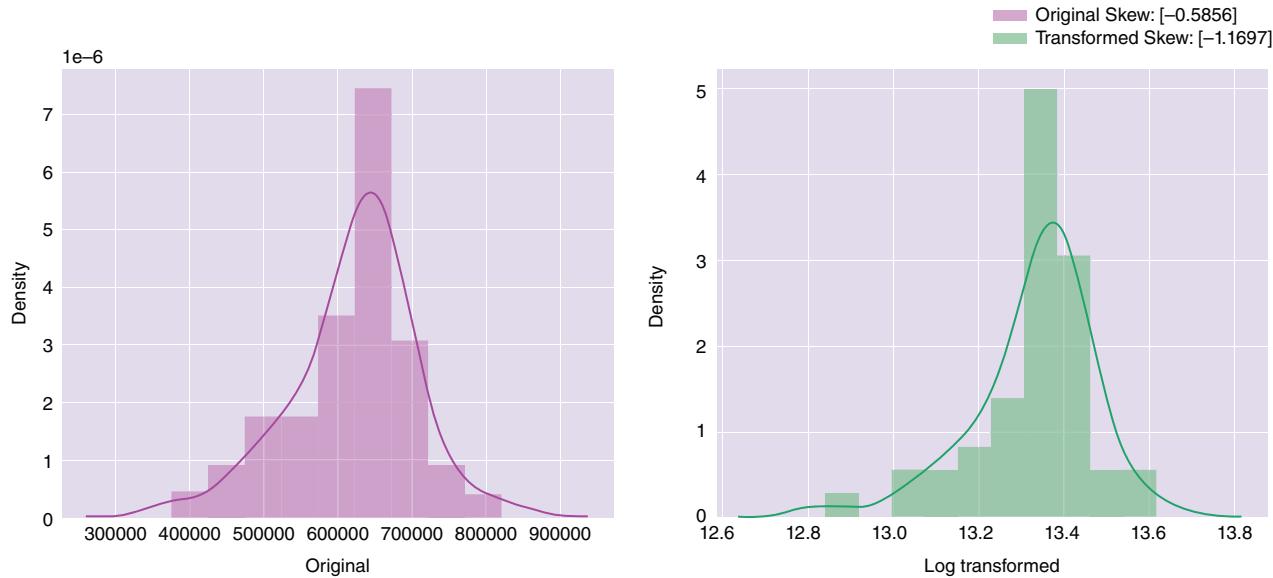
- Log transformation, in which each variable x is replaced by $\log(x)$ using a natural, base-10, or base-2 logarithm.
- Square root transformation, in which x is replaced by the square root of x . (This will produce a moderate effect but can be applied to zero values.)
- Reciprocal transformation, in which x is replaced by its inverse ($1/x$).
- Box-Cox transformation.
- Yeo-Johnson transformation.

The log transformation can be coded using NumPy:

Input:

```
log_target = np.log1p(X)
plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew:{0}".format(np.round(skew(X),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(log_target, label= "Transformed Skew:{0}".format(np.round(skew(log_target),4)), color="g", ax=ax[1], xlabel="LOG TRANSFORMED")
fig.legend()
plt.show()
```

When we apply it to our data, we can see the following output:

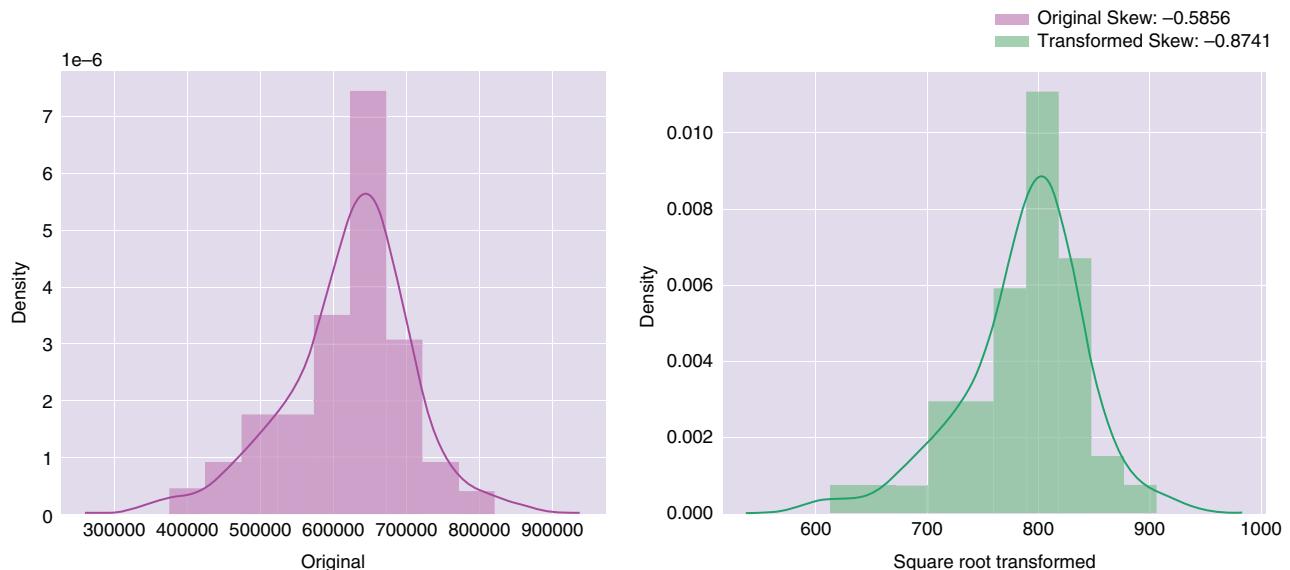


We proceed in the same way for square root transformation:

Input:

```
sqrrt_target = df['TotalGrayVol']**(1/2)
plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(df['TotalGrayVol'], label= "Orginal Skew:{0}".format(np.round(skew(df['TotalGrayVol']),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(sqrrt_target, label= "Transformed Skew:{0}".format(np.round(skew(sqrrt_target),4)), color="g", ax=ax[1], xlabel="SQUARE ROOT TRANSFORMED")
fig.legend()
plt.show()
```

And we see a different transformed skew:

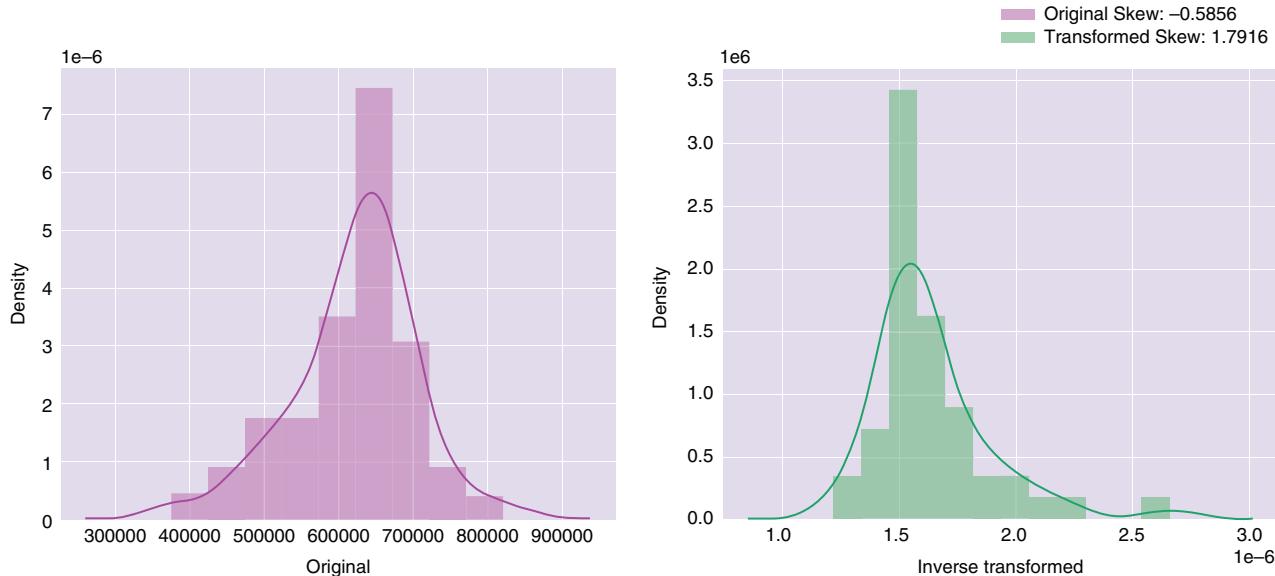


Reciprocal transformation can be coded as follows:

Input:

```
re_target = 1/df['TotalGrayVol']
plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(df['TotalGrayVol'], label= "Orginal Skew :{0}".format(np.round(skew(df['TotalGrayVol']),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(re_target, label= "Transformed Skew:{0}".format(np.round(skew(re_target),4)), color="g", ax=ax[1], xlabel="INVERSE TRANSFORMED")
fig.legend()
plt.show()
```

Output:



For all x that are strictly positive, we define the Box-Cox transformation as follows:

$$B(x, \lambda) = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(x) & \text{if } \lambda = 0 \end{cases}$$

where λ is a parameter that we choose. We can perform the Box-Cox transformation on both time-based and non-time series data. We must choose a value for λ that provides the best approximation of the normal distribution of our feature. In Python, SciPy has a `boxcox` function that chooses the optimal value of λ for us (`scipy.stats.boxcox`). It is also possible to specify an alpha number that calculates the confidence interval for the optimal value of λ .

Box-Cox transformation requires input data to be strictly positive. To code Box-Cox, we can use SciPy as follows:

Input:

```
from scipy.stats import boxcox

bcx_target, lam = boxcox(df['TotalGrayVol'])

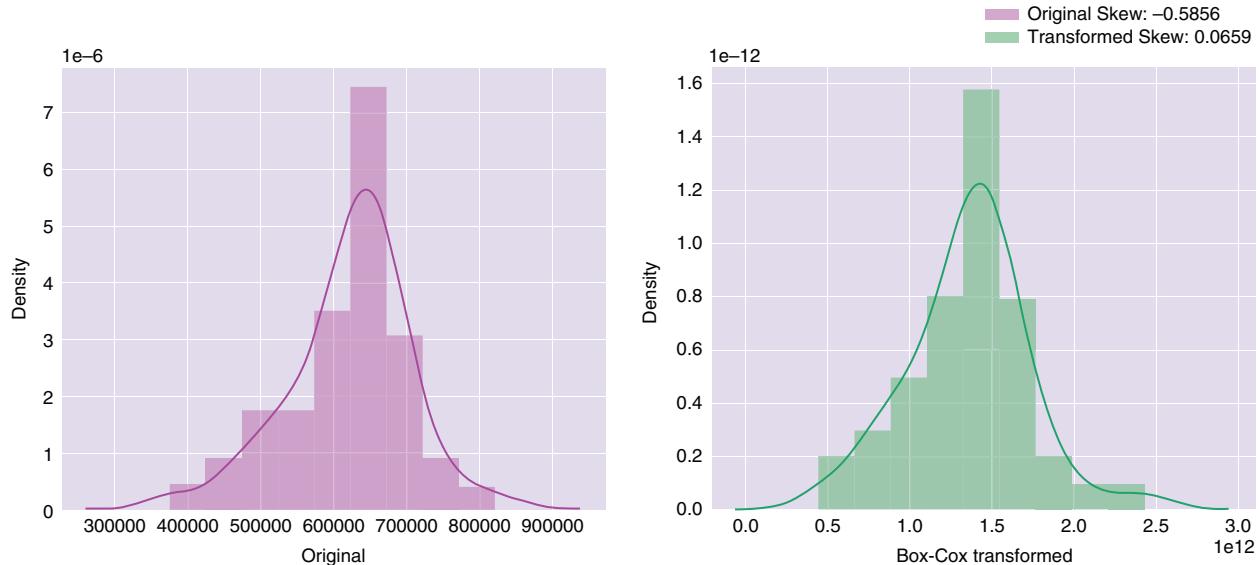
plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
```

```

sns.distplot(df['TotalGrayVol'], label= "Orginal Skew :{0}".format(np.round(skew(df['TotalGrayVol']),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(bcx_target, label= "Transformed Skew:{0}".format(np.round(skew(bcx_target),4)), color="g", ax=ax[1], xlabel="BOX-COX TRANSFORMED")
fig.legend()
plt.show()

```

We can see in the output that the skew of the transformed data is very close to zero:



We could also use sklearn in this way:

```
preprocessing.power_transform(data, method='box-cox', standardize=True, copy=True)
```

Like the Box-Cox transformation, the Yeo-Johnson is a nonlinear transformation that allows reducing the skewness and obtaining a distribution closer to normal:

$$\psi(\lambda, y) = \begin{cases} \frac{(y + 1)^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, y \geq 0 \\ \log(y + 1) & \text{if } \lambda = 0, y \geq 0 \\ -\frac{[(-y + 1)^{(2-\lambda)} - 1]}{2-\lambda} & \text{if } \lambda \neq 2, y < 0 \\ -\log(-y + 1) & \text{if } \lambda = 2, y < 0 \end{cases}$$

Yeo-Johnson transformation supports both positive and negative data. As is the case for Box-Cox, we can use SciPy to code Yeo-Johnson transformation.

Input:

```

from scipy.stats import yeojohnson

yj_target, lam = yeojohnson(df['TotalGrayVol'])

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)

```

```

sns.distplot(df['TotalGrayVol'], label= "Orginal Skew :{}".format(np.round(skew(df['TotalGrayVol']),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(bcx_target, label= "Transformed Skew:{}".format(np.round(skew(bcx_target),4)), color="g", ax=ax[1], xlabel="YEO-JOHNSON TRANSFORMED")
fig.legend()
# plt.savefig(folder+'bcx.png', bbox_inches='tight')
plt.show()

```

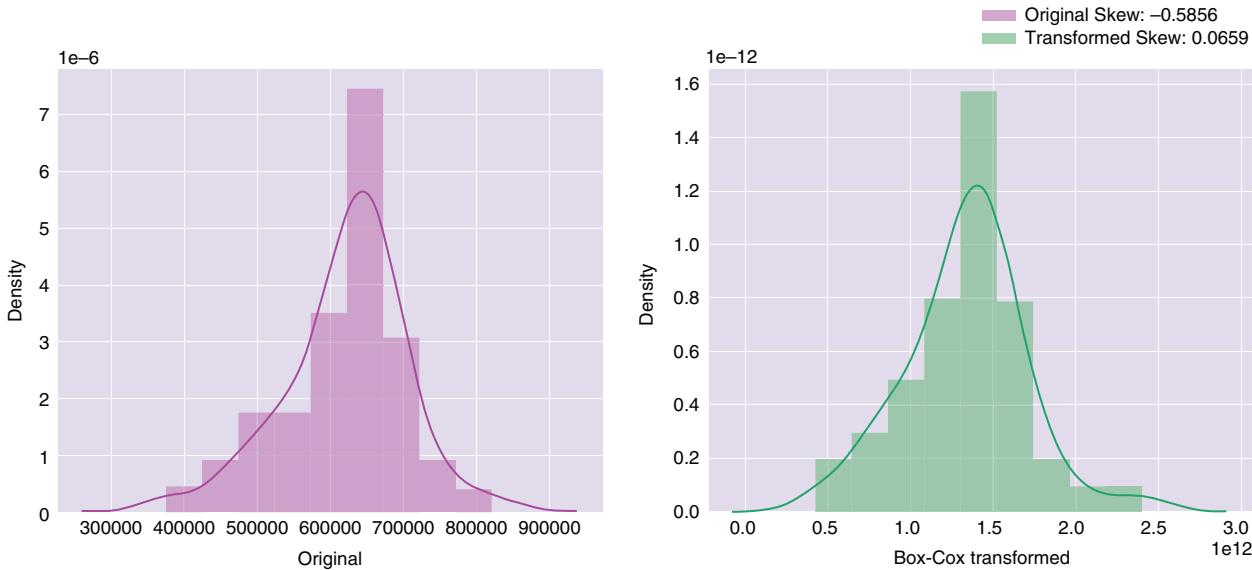
or

```

preprocessing.power_transform(data, method='yeo-johnson', standardize=True,
copy=True)

```

Output:



2.1.1.8 Quantile Transformation

Nonparametric quantile transformation transforms the data to a certain data distribution such as normal or uniform distribution by applying the quantile function, which is an inverse function of the cumulative distribution function, into the data.

Let X be a random variable following a normal distribution:

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

The quantile function of X is then calculated by the following equation:

$$Q_X(p) = \sqrt{2\sigma} \cdot \text{erf}^{-1}(2p - 1) + \mu$$

where $\text{erf}^{-1}(x)$ is the inverse error function.

Alternatively, let X be a random variable following a continuous uniform distribution:

$$X \sim \mathcal{U}(a, b)$$

The quantile function of X is then the following:

$$Q_X(p) = \begin{cases} -\infty, & \text{if } p = 0 \\ bp + a(1-p), & \text{if } p > 0 \end{cases}$$

We can code quantile transformation with Gaussian distribution using scikit-learn, as follows:

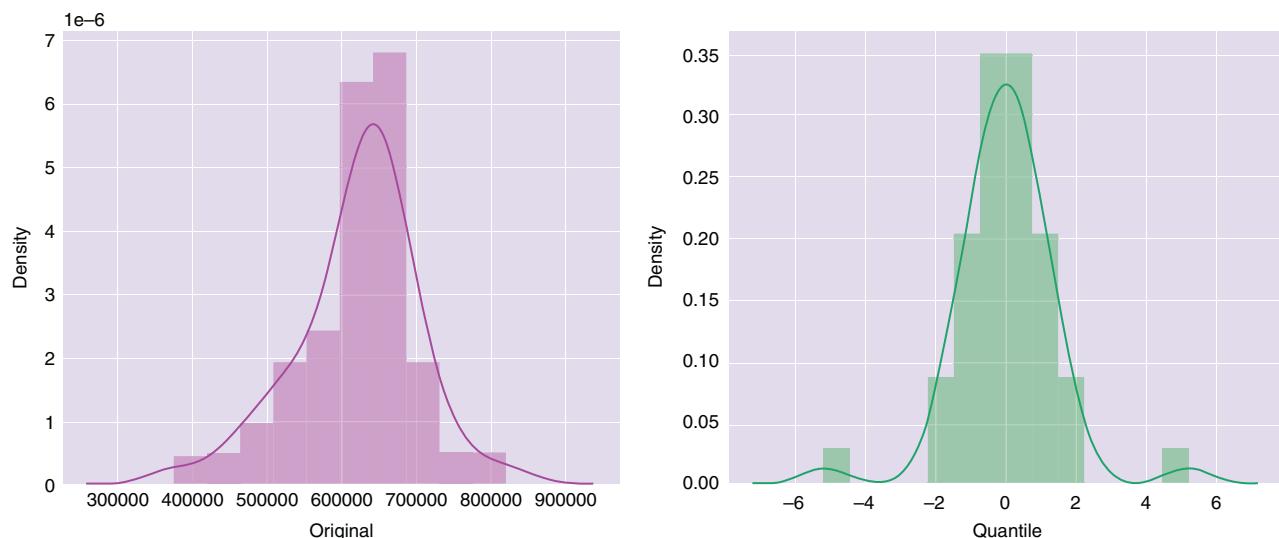
Input:

```
from sklearn.preprocessing import QuantileTransformer
Transformer=QuantileTransformer(n_quantiles=900, output_distribution="normal")

quantile_transform = Transformer.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(quantile_transform, label= "Transformed Skew:{0}".format(np.round(skew(quantile_transform),4)), color="g", ax=ax[1], xlabel="Quantile")
fig.legend()
plt.show()
```

Output:



We can also code quantile transformation with uniform distribution, as follows:

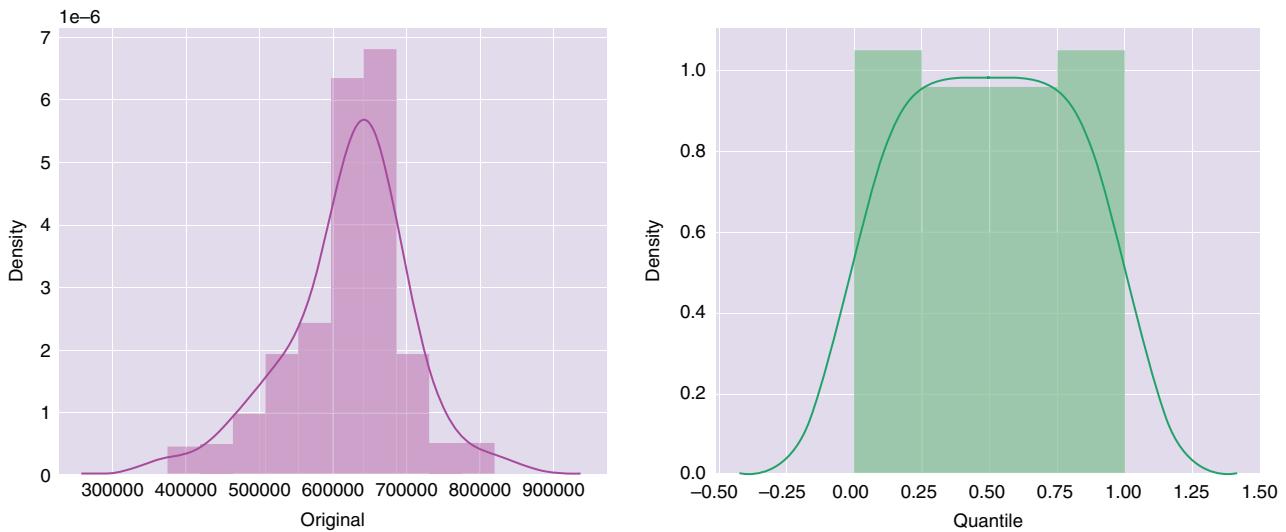
Input:

```
from sklearn.preprocessing import QuantileTransformer
Transformer=QuantileTransformer(n_quantiles=900, output_distribution="uniform")

quantile_transform = Transformer.fit_transform(X)

plt.rcParams["figure.figsize"] = 13,5
fig,ax = plt.subplots(1,2)
sns.distplot(X, label= "Orginal Skew :{0}".format(np.round(skew(X),4)), color="magenta", ax=ax[0], xlabel="ORIGINAL")
sns.distplot(quantile_transform, label= "Transformed Skew:{0}".format(np.round(skew(quantile_transform),4)), color="g", ax=ax[1], xlabel="Quantile")
fig.legend()
plt.show()
```

Output:



2.1.2 Example: Rescaling Applied to an SVM Model

The rescaling method we choose will have an impact on the accuracy of our models, as can be seen in the table below showing the accuracy of our brain data sample using 80% of data to train an SVM model with RBF (radio basis function) kernels and 20% to test model accuracy:

Rescaling method	SVM with RBF kernel: Accuracy
StandardScaler	0.7
MinMaxScaler	0.2
MaxAbsScaler	0.8
RobustScaler	0.9
Normalizer	0.8
Log Transformation	0.8
Yeo-Johnson	0.9
Quantile with Gaussian	0.5
Quantile with Uniform	0.8

If we wish to use hephAIstos to create a pipeline with the same data as above, we can go to the hephAIstos main folder and create a Python file (for example, mypipeline.py). Let us start with an example in which we desire to rescale our brain dataset using RobustScaler and apply an SVM with RBF kernel. As we did previously, we select “Target” and “TotalGravVol” features and take 20% of the data for testing purposes. The 80% of training data and 20% of testing data are not the same as previously (randomly chosen). Adding cross-validation is always better than focusing only on accuracy. Here, we will obtain results for a fivefold cross-validation (cv = 5).

Input:

```
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Import dataset
from data.datasets import brain_train
data = brain_train()
df = data[["Target", "TotalGrayVol"]]

ml_pipeline_function(df, output_folder = './Outputs/', test_size = 0.2, rescaling =
'robust_scaler', classification_algorithms=['svm_rbf'], cv = 5)
```

Output:

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10
 SVM_rbf				
Rescaling Method	RobustScaler			
Missing Method	None			
Extraction Method	None			
Accuracy	0.6			
Precision	0.6			
Recall	0.6			
F1 Score	0.6			
Cross-validation mean	0.75			
Cross-validation std	0.055328			

In the Outputs/ folder, we can find the model (svm_rbf.joblib), which can be used later, and a file named classification_metrics.csv with the results of the pipeline.

To run a pipeline with several rescaling methods, we can use the code below.

Input:

```
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Import dataset
from data.datasets import brain_train
data = brain_train()
df = data[["Target", "TotalGrayVol"]]

# Rescaling methods to test
scale = [
    'standard_scaler',
    'minmax_scaler',
    'maxabs_scaler',
    'robust_scaler',
    'normalizer',
    'log_transformation',
    'square_root_transformation',
    'reciprocal_transformation',
    'box_cox',
    'yeo_johnson',
    'quantile_gaussian',
    'quantile_uniform',
]

for rescale in scale:
    ml_pipeline_function(df, test_size = 0.2, rescaling = rescale,
classification_algorithms=['svm_rbf'], cv = 5)
```

Output:

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10
 SVM_rbf				
Rescaling Method	StandardScaler			
Missing Method	None			
Extraction Method	None			
Accuracy	0.6			
Precision	0.6			
Recall	0.6			
F1 Score	0.6			
Cross-validation mean	0.75			
Cross-validation std	0.055328			

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10
 SVM_rbf				
Rescaling Method	MinMaxScaler			
Missing Method	None			
Extraction Method	None			
Accuracy	0.6			
Precision	0.6			
Recall	0.6			
F1 Score	0.6			
Cross-validation mean	0.75			
Cross-validation std	0.055328			

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10

SVM_rbf

Rescaling Method	MaxAbsScaler
Missing Method	None
Extraction Method	None
Accuracy	0.6
Precision	0.6
Recall	0.6
F1 Score	0.6
Cross-validation mean	0.75
Cross-validation std	0.055328

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10

SVM_rbf

Rescaling Method	RobustScaler
Missing Method	None
Extraction Method	None
Accuracy	0.6
Precision	0.6
Recall	0.6
F1 Score	0.6
Cross-validation mean	0.75
Cross-validation std	0.055328

	precision	recall	f1-score	support
1	0.50	1.00	0.67	5
2	0.00	0.00	0.00	5
accuracy			0.50	10
macro avg	0.25	0.50	0.33	10
weighted avg	0.25	0.50	0.33	10

SVM_rbf

Rescaling Method	Normalizer
Missing Method	None
Extraction Method	None
Accuracy	0.5
Precision	0.5
Recall	0.5
F1 Score	0.5
Cross-validation mean	0.75
Cross-validation std	0.055328

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.44	0.80	0.57	5
2	0.00	0.00	0.00	5
accuracy			0.40	10
macro avg	0.22	0.40	0.29	10
weighted avg	0.22	0.40	0.29	10

SVM_rbf

Rescaling Method	Log
Missing Method	None
Extraction Method	None
Accuracy	0.4
Precision	0.4
Recall	0.4
F1 Score	0.4
Cross-validation mean	0.778571
Cross-validation std	0.065465

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.44	0.80	0.57	5
2	0.00	0.00	0.00	5
accuracy			0.40	10
macro avg	0.22	0.40	0.29	10
weighted avg	0.22	0.40	0.29	10

SVM_rbf

Rescaling Method	SquareRoot
Missing Method	None
Extraction Method	None
Accuracy	0.4
Precision	0.4
Recall	0.4

F1 Score	0.4
Cross-validation mean	0.75
Cross-validation std	0.055328

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10

SVM_rbf

Rescaling Method	Box-Cox
Missing Method	None
Extraction Method	None
Accuracy	0.6
Precision	0.6
Recall	0.6
F1 Score	0.6
Cross-validation mean	0.75
Cross-validation std	0.055328

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10

SVM_rbf

Rescaling Method	Yeo-Johnson
Missing Method	None
Extraction Method	None
Accuracy	0.6
Precision	0.6
Recall	0.6
F1 Score	0.6
Cross-validation mean	0.75
Cross-validation std	0.055328

Classification Report for SVM RBF

	precision	recall	f1-score	support
1	0.56	1.00	0.71	5
2	1.00	0.20	0.33	5
accuracy			0.60	10
macro avg	0.78	0.60	0.52	10
weighted avg	0.78	0.60	0.52	10

SVM_rbf

Rescaling Method	Quantile-Gaussian
Missing Method	None
Extraction Method	None
Accuracy	0.6
Precision	0.6
Recall	0.6
F1 Score	0.6
Cross-validation mean	0.75
Cross-validation std	0.055328

	precision	recall	f1-score	support
1	0.50	1.00	0.67	5
2	0.00	0.00	0.00	5
accuracy			0.50	10
macro avg	0.25	0.50	0.33	10
weighted avg	0.25	0.50	0.33	10

SVM_rbf

Rescaling Method	Quantile-Uniform
Missing Method	None
Extraction Method	None
Accuracy	0.5
Precision	0.5
Recall	0.5
F1 Score	0.5
Cross-validation mean	0.721429
Cross-validation std	0.014286

We have explored a few techniques for feature scaling, but there are many more, and there is not an obvious answer regarding which is the best feature scaling method. Depending on context, it is important to explore different techniques; there are many parameters to consider. A certitude is that machine learning does not know the difference between the weight of a basket of strawberries and its price. This would be a “no-brainer” for humans, but machine learning simply sees numbers. If there is a vast difference in the range, the machine makes the underlying assumption that higher ranging numbers have superiority. This is particularly true for machine learning algorithms that calculate distances between data points. In addition, few algorithms converge faster with feature scaling than without it, which is the case for the example of neural network gradient descent. There are also algorithms that do not really require feature scaling; most of these such as

tree-based algorithms (CART, random forests, gradient-boosted decision tree, etc.) rely on rules. Tree-based algorithms use series of inequalities. If the standard deviation is small and the distribution is not Gaussian, MinMaxScaler responds well but is sensitive to outliers. MaxAbsScaler is similar to MinMaxScaler, with the difference that the values are mapped in the range [0, 1], and it also suffers from large outliers. The StandardScaler is also very sensitive to outliers, and if the data are not normally distributed, it is clearly not the best scaler. RobustScaler and quantile transformation are more robust to outliers. The power transformer is a family of parametric and monotonic transformations that can be applied to make data more Gaussian, stabilizing variance and minimizing skewness through maximum likelihood estimation.

2.2 Strategies to Work with Categorical (Discrete) Data

In machine learning, we need to process different types of data. Some of these types are continuous variables, and others are categorical variables. In a way, we can compare the difference between continuous and categorical data with regression and classification algorithms, at least for data inputs. As we are using data, it is critically important to consider and process the categorical data correctly to avoid any incorrect impact on the performance of the machine learning models. We do not really have a choice here, as in any case we need to transform categorical data, often text, into numeric and usable data for calculation. Most of the time, we encounter three major classes of categorical data: binary, nominal, and ordinal (Figure 2.5).

The nominal categorical data attribute indicates that there is no concept of ordering among their values, such as types of video games (simulation and sport, action-adventure, real-time strategy, shooters, multiplayer online battle arena, etc.) or a “pet” variable that can have the values of “dog,” “cat,” and “bird.” Ordinal categorical attributes reflect a notion of order among their values. For example, we can think of the size of T-shirts (small, medium, large) or a podium result (first, second, third). Both nominal and ordinal categories are often called labels or classes. These discrete values can be text, numbers, or even images depending on the context. For instance, we can convert a numerical variable to an ordinal variable by dividing its range into bins and assigning values to each bin; we call this process discretization. A numerical variable between 1 and 50 can be divided into an ordinal variable with five labels (1–10, 11–20, 21–30, 31–40, 41–50). Some algorithms, such as a decision tree, can work with categorical data directly with no specific transformations. Most algorithms and tools need to operate on numeric data (for more efficient implementation rather than as hard limitations on the algorithms themselves) and not label data directly. For instance, if we use scikit-learn, all variables should be in numerical form.

It is preferable to apply continuous and categorical transformations after splitting the data between training and testing, as illustrated in Figure 2.6. We can choose different encoders for different features. The output, which is encoded data, can be merged with the rescaled continuous data to train the models. The same process is applied to the test data before applying the trained models.

Many methods are available to encode categorical variables to feed models. In this chapter, we will investigate the following well-known encoding methods:

- Ordinal encoding
- One hot encoding
- Label encoding
- Helmert encoding
- Binary encoding



Figure 2.5 Types of categorical data.

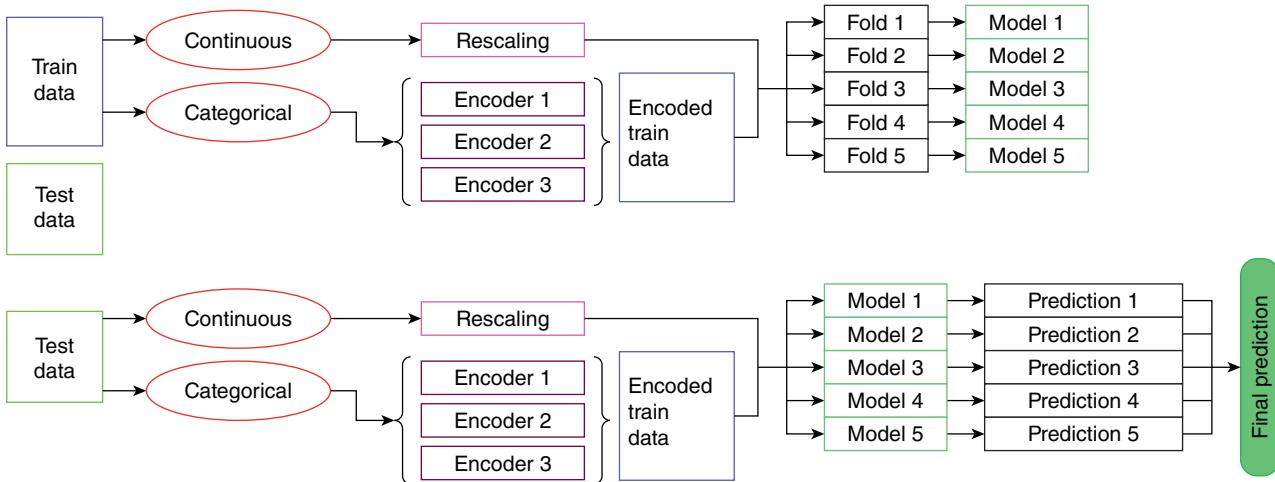


Figure 2.6 A classic workflow with continuous and categorical data.

- Frequency encoding
- Mean or target encoding
- Sum encoding
- Weight of evidence encoding
- Probability ratio encoding
- Hashing encoding
- Backward difference encoding
- Leave-one-out encoding
- James-Stein encoding
- M-estimator encoding

Different coding systems exist for categorical variables, including classic encoders that are well known and widely used (ordinal, one-hot, binary, frequency, hashing), the contrast encoders that encode data by examining different categories (or levels) of features, such as Helmert or backward difference, and Bayesian encoders, which use the target as a foundation for encoding. Target, leave-one-out, weight of evidence, James-Stein, and M-estimator are Bayesian encoders. Even though we already have a good list of encoders to explore, there are many more! It is important to master a couple of them and then explore further.

Libraries to code these methods, such as scikit-learn or pandas, are available in the open-source world. In addition, `category_encoders` is a very useful library for encoding categorical columns:

```
from category_encoders.ordinal import OrdinalEncoder
from category_encoders.woe import WOEEncoder
from category_encoders.target_encoder import TargetEncoder
from category_encoders.sum_coding import SumEncoder
from category_encoders.m_estimate import MEstimateEncoder
from category_encoders.leave_one_out import LeaveOneOutEncoder
from category_encoders.helmert import HelmertEncoder
from category_encoders.cat_boost import CatBoostEncoder
from category_encoders.james_stein import JamesSteinEncoder
from category_encoders.one_hot import OneHotEncoder
```

Depending on the context, some methods are more suitable than others, as shown in Figure 2.7. We will explain and code each of them.

All code examples for this section can be found in hephaistos/Notebooks/Categorical_transform.ipynb.

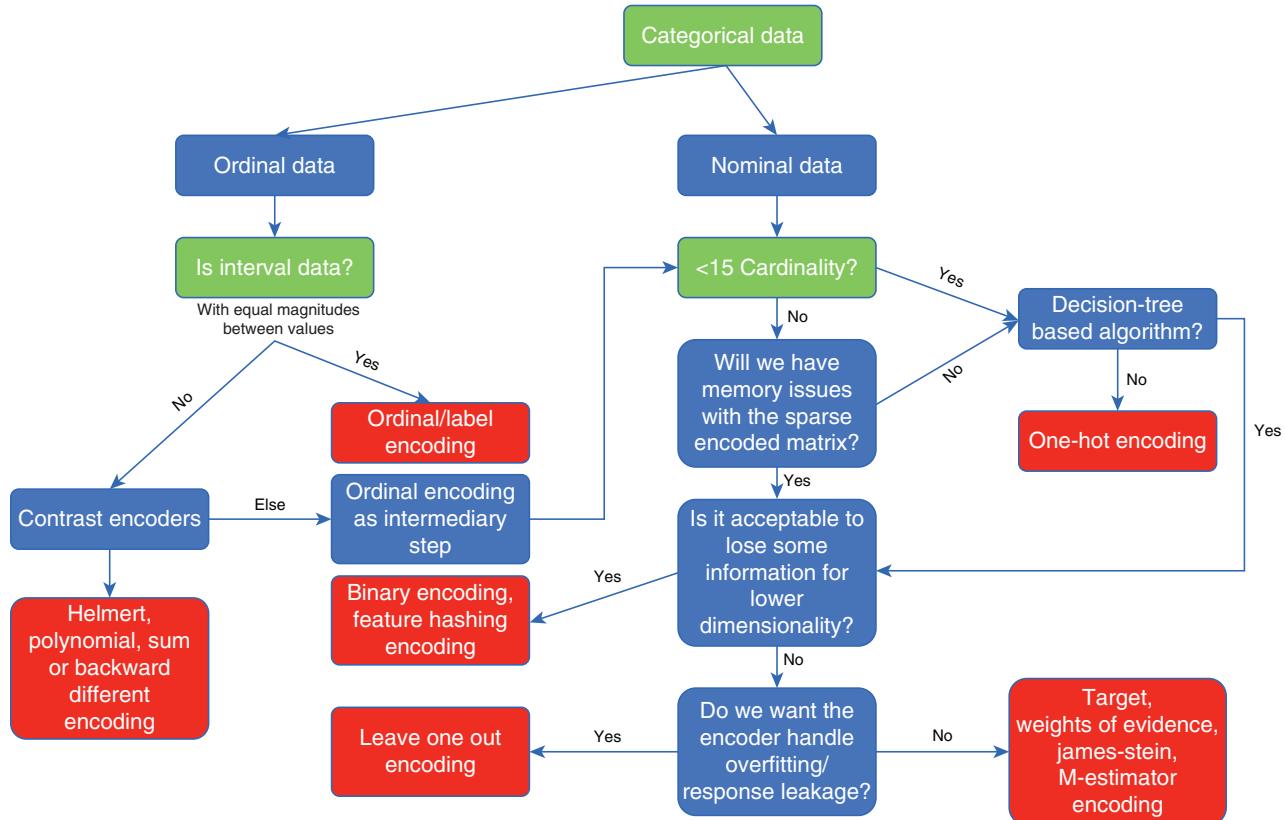


Figure 2.7 Summary of methods depending on the context.

2.2.1 Ordinal Encoding

The easiest way to encode ordinal data is to assign it an integer value (integer encoding). For example, if we have a variable “size,” we can assign 0 to “small,” 1 to “medium,” and 2 to “large.” Integer encoding is easily reversible. Ordinal encoding can be applied if there is a known relationship between categories. We can use pandas and assign the original order of the variable through a dictionary and then map each row for the variable as per the dictionary.

Let us first create a pandas DataFrame with some data in it (Size, Color, Class columns).

Input:

```

import os
import numpy as np
import pandas as pd

data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
                'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black'],
        'Class': [1, 1, 1, 0, 1, 0, 0, 1]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Class'])
print(df)
  
```

Output:

	Size	Color	Class
0	small	red	1
1	small	green	1
2	large	black	1
3	medium	white	0
4	large	blue	1
5	large	red	0
6	small	green	0
7	medium	black	1

The ordinal encoding transformation is also available in scikit-learn via the `OrdinalEncoder` class, which by default will assign integers to labels in the order that is observed in the data. If we need to specify a desired order, we can use the “categories” argument with the rank order of all expected labels.

Let us encode the “Size” and “Color” features.

Input:

```
from sklearn.preprocessing import OrdinalEncoder

# Creating an instance of OrdinalEncoder
enc = OrdinalEncoder()

# Assigning numerical values and storing it
enc.fit(df[["Size", "Color"]])
df[["Size", "Color"]] = enc.transform(df[["Size", "Color"]])

# Display Dataframe
print(df)
```

Output:

	Size	Color	Class
0	2.0	3.0	1
1	2.0	2.0	1
2	0.0	0.0	1
3	1.0	4.0	0
4	0.0	1.0	1
5	0.0	3.0	0
6	2.0	2.0	0
7	1.0	0.0	1

2.2.2 One-Hot Encoding

One-hot encoding is very popular. With the one-hot encoding method, we map each category to a vector containing 1 (presence) or 0 (absence). This is applied when no order relationship exists. It creates new binary columns in which 1 indicates the presence of each possible value from the original data.

In this approach (Figure 2.8), for each category of a feature, we create a new column (sometimes called a dummy variable) with binary encoding (0 or 1) to denote whether a particular row belongs to this category. This method can be challenging if our categorical variable takes on many values, and it is preferable to avoid it for variables with more than 15 different values. The drawback of this method is the size of the variable in memory because it uses as many bits as there are states, meaning that the necessary memory space increases linearly with the number of states. Creating many columns can slow down the learning significantly.

Pandas has a function called `get_dummies` to perform one-hot encoding.

Input:

```
df = pd.get_dummies(df, prefix="One", columns=['Size', 'Color'])
print(df)
```

Output:

Class	One_large	One_medium	One_small	One_black	One_blue	One_green	One_red	One_white
0	1	0	0	1	0	0	0	1
1	1	0	0	1	0	0	1	0
2	1	1	0	0	1	0	0	0
3	0	0	1	0	0	0	0	1
4	1	1	0	0	0	1	0	0
5	0	1	0	0	0	0	0	1
6	0	0	0	1	0	0	1	0
7	1	0	1	0	1	0	0	0

Input using scikit-learn:

```
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(handle_unknown='ignore')
enc_df = pd.DataFrame(enc.fit_transform(df[['Size', 'Color']]).toarray())
df = df.join(enc_df)
print(df)
```



Color	Blue	Red	Green
Blue	1	0	0
Blue	1	0	0
Red	0	1	0
Green	0	0	1
Red	0	0	1

Figure 2.8 Example of one-hot encoding.

Output:

	Size	Color	Class	0	1	2	3	4	5	6	7	
0	small	red		1	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
1	small	green		1	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0
2	large	black		1	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
3	medium	white		0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0
4	large	blue		1	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
5	large	red		0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
6	small	green		0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0
7	medium	black		1	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0

Here, we have applied one hot encoding to the “Size” and “Color” variables. The “Size” feature has an ordinality, which means that applying one-hot encoding is not appropriate. In addition, if we turn [red, green, black, white, blue, red, green, black] into [1, 2, 3, 4, 5, 1, 2, 3], we impose ordinality on a variable that is not ordinal. There are algorithms such as decision trees that can handle categorical variables well, meaning that we can optimize memory; however, for other types of algorithms, one-hot encoding can make a major difference. One-hot encoding has the advantage that the result is binary and not ordinal, meaning that we are in an orthogonal vector space. One way to address dimensionality is to use principal component analysis (PCA) as well, just after application of one-hot encoding.

2.2.3 Label Encoding

With label encoding, we replace a categorical value with a numeric value (from 0 to N , where N is the number of categories for the feature) for each category. If the feature contains five categories, we will use 0, 1, 2, 3, and 4. This approach can create a major issue because, even if there is no relationship or order between categories, the algorithm might interpret some order or relationship.

With the same data, we can code label encoding as follows:

```
from sklearn.preprocessing import LabelEncoder

# Creating an instance of Labelencoder
enc = LabelEncoder()

# Assigning numerical value and storing it
X = X.apply(enc.fit_transform)
print(X)
```

Output:

	Size	Color	Class
0	2	3	1
1	2	2	1
2	0	0	1
3	1	4	0
4	0	1	1
5	0	3	0
6	2	2	0
7	1	0	1

2.2.4 Helmert Encoding

Helmert encoding compares each level of a categorical variable to the mean of the subsequent levels.

Let us say we have the following categorical variable and categories (small, medium, large, x-large):

Input:

```
data = {'Size': ['small', 'small', 'small', 'small', 'medium', 'medium', 'medium',
'large','large', 'x-large']}
df = pd.DataFrame(data, columns = ['Size'])
print(df)
```

Output:

	Size
0	small
1	small
2	small
3	small
4	medium
5	medium
6	medium
7	large
8	large
9	x-large

The Helmert encoding method compares the mean of the dependent variable for “small” with the mean of all of the subsequent levels of the categorical column (“medium,” “large,” “x-large”), the mean of the dependent variable for “medium” with the mean of all of the subsequent levels (“large,” “x-large”), and the mean of the dependent variable for “large” with the mean of all of the subsequent levels (in our case only one level, “x-large”).

Helmert encoding can be implemented with the *category_encoders* library. We must first import the category_encoders library after installing it. We invoke the HelmertEncoder function and call the *.fit_transform()* method on it with the DataFrame as the argument.

Input:

```
import category_encoders as ce
enc = ce.HelmertEncoder()
df = enc.fit_transform(df['Size'])
print(df)
```

Output:

	intercept	Size_0	Size_1	Size_2
0	1	-1.0	-1.0	-1.0
1	1	-1.0	-1.0	-1.0
2	1	-1.0	-1.0	-1.0
3	1	-1.0	-1.0	-1.0
4	1	1.0	-1.0	-1.0
5	1	1.0	-1.0	-1.0
6	1	1.0	-1.0	-1.0
7	1	0.0	2.0	-1.0
8	1	0.0	2.0	-1.0
9	1	0.0	0.0	3.0

We can ignore the intercept (columns with zero variance) by adding the `drop_invariant = True` option to `ce.HelmertEncoder()`. We can use Helmert encoding when levels of the categorical variable are ordered (smallest to largest, for instance).

2.2.5 Binary Encoding

The binary encoding method consists of different operations. First, the categories are encoded as ordinal, then the resulting integers are converted into a binary code; finally, the digits from that binary code are split into separate columns (Figure 2.9).

This process results in fewer dimensions than in one-hot encoding. As for Helmert encoding, we can use the `category_encoders` library to code it. We need to invoke the `BinaryEncoder` function by specifying the columns we wish to encode and then call the `.fit_transform()` method with the DataFrame as the argument.

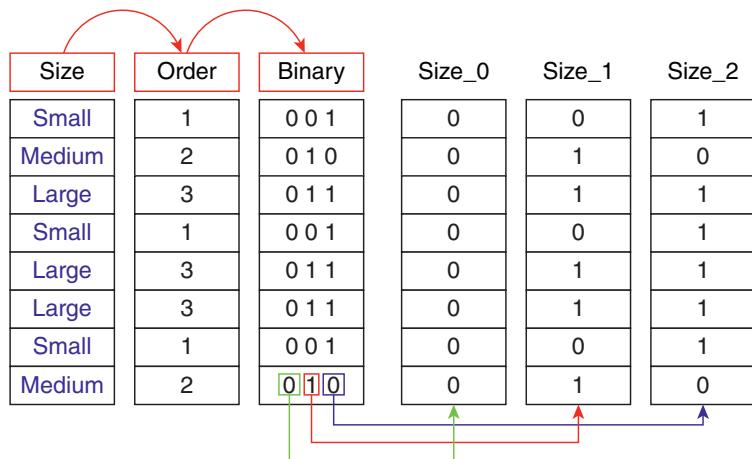


Figure 2.9 Binary encoding method.

Input:

```
import category_encoders as ce

data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
    'medium'],
    'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black'],
    'Class': [1, 1, 1, 0, 1, 0, 0, 1]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Class'])

enc = ce.BinaryEncoder(cols=['Color','Size'])
df_binary = enc.fit_transform(df)

df_binary
print(df_binary)
```

Output:

	Size_0	Size_1	Color_0	Color_1	Color_2	Class
0	0	1	0	0	1	1
1	0	1	0	1	0	1
2	1	0	0	1	1	1
3	1	1	1	0	0	0
4	1	0	1	0	1	1
5	1	0	0	0	1	0
6	0	1	0	1	0	0
7	1	1	0	1	1	1

2.2.6 Frequency Encoding

The frequency encoding method encodes by frequency, which means we create a new feature with the number of categories from the data (counts of each category).

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
    'medium'],
    'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black'],
    'Class': [1, 1, 1, 0, 1, 0, 0, 1]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Class'])

frequency = df.groupby('Color').size()/len(df)
df.loc[:, 'Frequency'] = df['Color'].map(frequency)
print(df)
```

Output:

	Size	Color	Class	Frequency
0	small	red	1	0.250
1	small	green	1	0.250
2	large	black	1	0.250
3	medium	white	0	0.125
4	large	blue	1	0.125
5	large	red	0	0.250
6	small	green	0	0.250
7	medium	black	1	0.250

2.2.7 Mean Encoding

To explain mean encoding, also called target encoding, let us take our DataFrame above with the variables “Size” and “Color.” We have also assigned a target variable that points to a binary classification problem with target variables 1 and 0.

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
               'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black'],
        'Target': [1, 1, 1, 0, 1, 0, 0, 1]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])
print(df)
```

Output:

	Size	Color	Target
0	small	red	1
1	small	green	1
2	large	black	1
3	medium	white	0
4	large	blue	1
5	large	red	0
6	small	green	0
7	medium	black	1

In this method, we encode, for each unique value of the categorical feature, based on the ratio of occurrence of the positive class in the target variable. For the feature “Color,” the value “Red” has two occurrences in the target variable, and one of those is the positive label. Mean encoding would be 0.5 for the value “Red” (Figure 2.10).

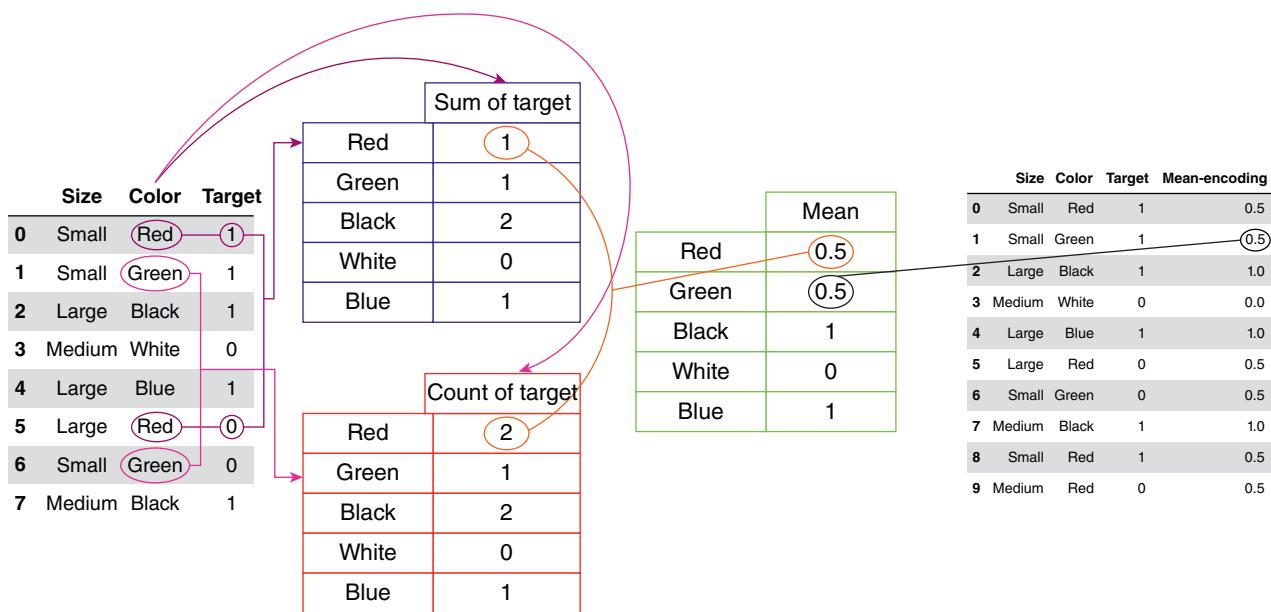


Figure 2.10 The mean encoding method.

Input:

```
mean_encoding = df.groupby('Color')['Target'].mean()
df.loc[:, 'Mean_encoding'] = df['Color'].map(mean_encoding)
print(df)
```

Output:

	Size	Color	Target	Mean_encoding
0	small	red	1	0.5
1	small	green	1	0.5
2	large	black	1	1.0
3	medium	white	0	0.0
4	large	blue	1	1.0
5	large	red	0	0.5
6	small	green	0	0.5
7	medium	black	1	1.0
8	small	red	1	0.5
9	medium	red	0	0.5

The volume of the data is not affected by this method, and it can help in faster learning. It provides more logic to the data in comparison with simple labeling, as we have a probability of our target variable that is conditional on each value of the feature. A well-known issue with mean encoding is overfitting. We need to increase regularization with cross-validation and add random noise to the representation of the category in the dataset. Mean encoding is particularly useful with gradient boosting trees as it decreases cardinality and reaches better loss with a shorter tree, thus improving the classification.

2.2.8 Sum Encoding

The sum encoding method, also called effect of deviation encoding, compares the mean of the target (dependent variable) for a given level of a categorical column to the overall mean of the target. It is similar to one hot encoding with the difference that we use values of 1, 0, and -1 to encode the data. It can be used in linear regression types of models and can be coded with the *category_encoders* library.

Input:

```
from category_encoders import SumEncoder

data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
                 'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

sum_encoder = SumEncoder()
df_encoded = sum_encoder.fit_transform(df['Size'], df['Target'])
print(df_encoded)
```

Output:

	intercept	Size_0	Size_1
0	1	1.0	0.0
1	1	1.0	0.0
2	1	0.0	1.0
3	1	-1.0	-1.0
4	1	0.0	1.0
5	1	0.0	1.0
6	1	1.0	0.0
7	1	-1.0	-1.0
8	1	1.0	0.0
9	1	-1.0	-1.0

2.2.9 Weight of Evidence Encoding

The weight of evidence (WoE) method comes from the credit-scoring world and measures the “strength” of a grouping technique to separate customers whether customers have defaulted on a loan or not:

$$WoE = \ln\left(\frac{\text{Distribution of Goods}}{\text{Distribution of Bads}}\right)$$

Strict application of this formula might lead to target leakage and overfit. To avoid this situation, we induce a regularization parameter a :

$$\text{nominator} = \frac{n^+ + a}{y^+ + 2 * a}$$

$$\text{denominator} = \frac{n - n^+ + a}{y - y^+ + 2 * a}$$

$$\hat{x}^k = \ln\left(\frac{\text{nominator}}{\text{denominator}}\right)$$

The variables y and y^+ are the total number of observations and the total number of positive observations, respectively ($y = 1$); n and n^+ are the number of observations and the number of positive observations, respectively ($y = 1$) for a given value of a categorical column; a is the regularization hyperparameter (selected by a user).

We could replace goods and bads with events (bads) and non-events (goods). Distribution refers to the proportion of goods or bads in the respective group relative to the column totals. The value of WoE will be 0 if the odds $P[\text{Distribution Goods}]/P[\text{Distribution Bads}]$ is equal to 1. In a group, if $P[\text{Distribution Goods}] < P[\text{Distribution Bads}]$, the odds ratio will be less than 1 and the WoE will be less than 0. On the contrary, if $P[\text{Distribution Goods}] > P[\text{Distribution Bads}]$, WoE will be a positive number.

Alternatively, let us say that we have a dataset of people suffering from disease A and that we would like to calculate the relationship between being male and the possibility of having disease A:

- The probability of males having disease A is 0.43.
- The probability of males not suffering from disease A is 0.41.
- We can calculate that $WoE = 0.048$.

Conventionally, we also calculate the information value (IV), which measures the importance of a feature:

$IV = (0.43 - 0.41) \times 0.048 = 0.001$, which means that gender is not a good predictor for disease A according to the following table:

<0.02	Not useful
0.02 to 0.1	Weak predictor
0.1 to 0.3	Medium predictor
0.3 to 0.5	Strong predictor
<0.5	Suspicious

In the context of machine learning, WoE is also used for the replacement of categorical values. With one-hot encoding, if we assume that a column contains five unique labels, there will be five new columns. In such a case, we can replace the values with the WoE. This method is particularly well suited for subsequent modeling using logistic regression. WoE transformation orders the categories on a “logistic” scale, which is natural for logistic regression.

If we use the `category_encoders` library, the code will look similar to the following:

```
from category_encoders import WOEEncoder

data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
                'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

# Creating an instance of SumEncoder
# regularization is mostly to prevent division by zero.
woe = WOEEncoder(random_state=42, regularization=0)
# Assigning numerical value and storing it
df_encoded = woe.fit_transform(df['Size'], df['Target'])
print(df_encoded)
```

It provides the following output by replacing the labels in “Size” by the WoE:

Size
0 0.000000
1 0.000000
2 0.693147
3 -0.693147
4 0.693147
5 0.693147
6 0.000000
7 -0.693147
8 0.000000
9 -0.693147

One of the drawbacks of WoE is the possible loss of information due to binning to relatively few categories. In addition, it does not consider correction between independent variables as an “univariate” measure. WoE assumes no interactions, which means that the same relationship should hold true across the spectrum of values. It might also lead to target leakage (and overfitting). On the other hand, it can transform an independent variable, and it establishes monotonic relationship to the dependent variable. WoE is advantageous for variables with too many discrete values (sparsely populated).

2.2.10 Probability Ratio Encoding

Probability ratio encoding is similar to WoE, but we only keep the ratio, not its logarithm. For each category, the mean of the target is calculated to equal 1 that is the probability $p(1)$ of being 1 and the probability $p(0)$ of not being 1 (it is 0). The ratio of happening and not happening is simply $p(1)/p(0)$. All the categorical values should be replaced with this ratio.

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
    'medium', 'small', 'medium'],
    'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
    'red', 'red'],
    'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

# Calculation of the probability of target being 1
probability_encoding_1 = df.groupby('Color')['Target'].mean()
# Calculation of the probability of target not being 1
probability_encoding_0 = 1 - probability_encoding_1
# Probability ratio calculation
df_encoded = probability_encoding_1 / probability_encoding_0
# Map the probability ratio into the data
df.loc[:, 'Proba_Ratio'] = df['Color'].map(df_encoded)
print(df)
```

Output:

	Size	Color	Target	Proba_Ratio
0	small	red	1	1.0
1	small	green	0	0.0
2	large	black	1	inf
3	medium	white	0	0.0
4	large	blue	1	inf
5	large	red	0	1.0
6	small	green	0	0.0
7	medium	black	1	inf
8	small	red	1	1.0
9	medium	red	0	1.0

Inf is due to a division by zero. To avoid this situation, we can replace the denominator with a small value:

```
probability_encoding_0 = np.where(probability_encoding_0 == 0, 0.00001,
probability_encoding_0)
```

2.2.11 Hashing Encoding

Hashing is a term mainly applied in cryptography and designates the action of converting an element, such as text or numbers, into a new representation space completely different from the original one. The idea is to make the initial element unrecognizable and inaccessible without having the cryptographic mechanism that performed the transformation. We can apply this logic to encode categorical variables. Hashing encoding is similar to one hot encoding, which converts the category into binary numbers using new variables. The difference is that we can fix the number of variables we desire. Hashing encoding maps each category to an integer within a predefined range with the help of a hash function. A simple hash function is when we assign $a = 1, b = 2, c = 3, d = 4$, and so on, and sum all the values of the label together. Depending on what we are performing with our model, we need to be careful not to have the same hash function or output if we have many categories.

```
ash("abc") = 1 + 2 + 3 = 6 # Hash Function
```

or

```
ash("Elsa") = 5 + 12 + 1 = 18 # Hash Function
```

If we desire four binary features, we can convert the output written in binary and select the last four bits. For example, $\text{hash}(\text{"Elsa"}) = 18$, and in binary $18 = 10,010$, which would give us the values 0, 0, 1, 0.

To implement feature hashing in Python, we can use the *category_encoders* library. Below, we transform the feature “Size” by selecting three bits in our hash value.

Input:

```

data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
                'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

import category_encoders as ce
# n_components contain the number of bits you want in your hash value.
encoder_purpose = ce.HashingEncoder(n_components=3)
df_encoded = encoder_purpose.fit_transform(df['Size'])
print(df_encoded)

```

Output:

	col_0	col_1	col_2
0	0	1	0
1	0	1	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	0	1	0
7	1	0	0
8	0	1	0
9	1	0	0

We can use different hashing methods using the *hash_method* option. Any method from hashlib will function (import hashlib):

```
encoder_purpose = ce.HashingEncoder(n_components=3, hash_method="sha256")
```

Hashing encoding is well suited for categorical variables with a large number of levels and scales very well with low memory usage.

2.2.12 Backward Difference Encoding

In the backward difference encoding method, which is similar to Helmert encoding, the mean of the dependent variable for a level is compared with the mean of the dependent variable for the prior level. Backward difference encoding falls under the classification of contrast encoders for categorical features. This method may be useful for both nominal and ordinal variables. In addition, in contrast to the examples of dummy encoding, we see regressed continuous values as outputs.

To implement backward difference encoding in Python, we can use the *category_encoders* library. Below, we transform the feature “Size.”

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
                 'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

import category_encoders as ce
encoder = ce.BackwardDifferenceEncoder(cols=['Size'])
df_encoded = encoder.fit_transform(df['Size'])
print(df_encoded)
```

Output:

	intercept	Size_0	Size_1
0	1	-0.666667	-0.333333
1	1	-0.666667	-0.333333
2	1	0.333333	-0.333333
3	1	0.333333	0.666667
4	1	0.333333	-0.333333
5	1	0.333333	-0.333333
6	1	-0.666667	-0.333333
7	1	0.333333	0.666667
8	1	-0.666667	-0.333333
9	1	0.333333	0.666667

2.2.13 Leave-One-Out Encoding

Leave-one-out encoding, a target-based encoder, excludes the current row’s target in calculating the mean target for a level to reduce the effect of outliers. In other words, it involves taking the mean target value of all data points in the category except the current row:

$$\hat{x}_i^k = \frac{\sum_{j \neq i} (y_j * (x_j == k)) - y_i}{\sum_{j \neq i} (x_j == k)}$$

The variables x_i and y_i are the i th value of categories and target, respectively. The overall calculates the mean target of category k for observation j if observation j is removed from the dataset.

For the test dataset, a category is replaced with the mean target of the category k in the training dataset:

$$\hat{x}_i^k = \frac{\sum y_j * (x_j == k)}{\sum (x_j == k)}$$

As usual, implementation can be performed with the *category_encoders* library. An example is shown below for the “Color” feature.

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
               'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

import category_encoders as ce
encoder = ce.LeaveOneOutEncoder(cols=['Color'])
df_encoded = encoder.fit_transform(df['Color'], df['Target'])
print(df_encoded)
```

Output:

Color
0 0.333333
1 0.000000
2 1.000000
3 0.500000
4 0.500000
5 0.666667
6 0.000000
7 1.000000
8 0.333333
9 0.666667

2.2.14 James-Stein Encoding

The target-based James-Stein encoder, only defined for normal distributions, was inspired by the James-Stein estimator.

For the feature value i , the James-Stein estimator returns a weighted average. The mean target estimation for category k can be calculated with the following formula:

$$\hat{x}^k = (1 - B) * \frac{n^+}{n} + B * \frac{y^+}{y}$$

The variables y and y^+ are the total number of observations and the total number of positive observations, respectively ($y = 1$); n and n^+ are the number of observations and the number of positive observations ($y = 1$), respectively, for a given value of a categorical column; B is the weight.

The first part of the formula is the mean target value for the observed feature value, and the second part is the mean target value regardless of the feature value. Depending on the weight we put on the conditional or global mean value, we may be in

an overfit or underfit situation. A large value of B will result in a larger weight of the global mean (underfit), while a low value of B will result in a larger weight of the condition mean (overfit).

A way to select B is to tune it like a hyperparameter. Charles Stein devised the following solution:

$$B = \frac{\text{var}(y^k)}{\text{var}(y^k) + \text{var}(y)}$$

If we cannot rely on the estimation of categories mean to target (y_i has a high variance), we need to put more weight on $\text{mean}(y)$, the global mean. We also need to assume that variance is the same among all categories and equal to the global variance of y (which might be a good estimation if we do not have too many unique categorical values). It is called pooled variance or pooled mode. We could also use an independent model by replacing the variances with squared standard errors, which will penalize small observation counts. In addition, the fact that we need to use it on a normal distribution is a serious limitation in classification task. A possible solution is to use a beta distribution or convert binary targets with log odds.

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
               'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

import category_encoders as ce
encoder = ce.JamesSteinEncoder(cols=['Color'])
df_encoded = encoder.fit_transform(df['Color'], df['Target'])

print(df_encoded)
```

Output:

Color	
0	0.5
1	0.0
2	1.0
3	0.0
4	1.0
5	0.5
6	0.0
7	1.0
8	0.5
9	0.5

2.2.15 M-Estimator Encoding

M-estimator encoding, a more general Bayesian approach, has only one hyperparameter (m) that represents the power of regularization and is generally useful for data with high cardinality. The default value of m is 1. The recommended values are in the range of 1–100 and higher is m stronger shrinking. M-estimator encoding can be calculated using the following formula:

$$\hat{x}^k = \frac{n^+ + \text{prior} * m}{y^+ + m}$$

where y^+ [count(category)] is the total number of positive observations ($y = 1$), n^+ is the number of positive observations ($y = 1$) for a given value of a categorical column [count(category) \times mean(category)], and prior [mean(target)] is an average value of target.

The implementation can be performed using the *category_encoders* library (with an example shown below for the “Color” feature).

Input:

```
data = {'Size': ['small', 'small', 'large', 'medium', 'large', 'large', 'small',
               'medium', 'small', 'medium'],
        'Color': ['red', 'green', 'black', 'white', 'blue', 'red', 'green', 'black',
                  'red', 'red'],
        'Target': [1, 0, 1, 0, 1, 0, 0, 1, 1, 0]}

df = pd.DataFrame(data, columns = ['Size', 'Color', 'Target'])

encoder = ce.MEstimateEncoder(cols=['Color'])
df_encoded = encoder.fit_transform(df['Color'], df['Target'])
print(df_encoded)
```

Output:

	Color
0	0.500000
1	0.166667
2	0.833333
3	0.250000
4	0.750000
5	0.500000
6	0.166667
7	0.833333
8	0.500000
9	0.500000

2.2.16 Using Hephaistos to Encode Categorical Data

As described in Chapter 1, we can include in the pipeline of hephaistos one of the methods described to encode categorical data. The following options are available: “ordinal_encoding,” “one_hot_encoding,” “label_encoding,” “helmert_encoding,” “binary_encoding,” “frequency_encoding,” “mean_encoding,” “sum_encoding,” “weightofevidence_encoding,” “probability_ratio_encoding,” “hashing_encoding,” “backward_difference_encoding,” “leave_one_out_encoding,” “james_stein_encoding,” and “m_estimator_encoding.” Different encoding methods can be combined.

Here is a simple example. From an insurance dataset, we want to encode three features:

- Two features (smoker and sex) with binary encoding.
- One feature (region) with label encoding.

We have included in the pipeline the row removal method to handle missing values. We go to the hephaistos folder, create a new Python file, and code the pipeline as shown below.

Input:

```
from ml_pipeline_function import ml_pipeline_function
import pandas as pd
import numpy as np

from data.datasets import insurance
df = insurance()
df = df.rename(columns={"charges": "Target"})

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['binary_encoding', 'label_encoding'],
features_binary = ['smoker', 'sex'], features_label = ['region'])
```

2.3 Time-Related Features Engineering

As we have seen, feature engineering requires the use of different methods to rescale continuous data and encode categorical data. If we add a time component, feature engineering can be more complex to understand, yet time-related features are found in many fields such as finance, weather forecasting, and healthcare. A time series is a sequence of numerical values representing the evolution of a specific quantity over time. The data are captured at equal intervals. Time series are useful to understand the behavior of a variable in the past and use this knowledge to predict the future behavior of the variable through probabilistic and statistical concepts. With the time variable, we can predict a stock price based on what happened yesterday, predict the evolution of a disease based on past experiences, or predict the road traffic in a city if we have data from the last few years. Time series may also reflect seasonality or trends that we can model mathematically.

To study some techniques for handling time-related features, we will use a dataset used to train models on weather forecasting in the Indian climate. This dataset provides data from 2013 to 2017 in the city of Delhi, India, with four parameters: meantemp, humidity, wind_speed, and meanpressure.

The Jupyter Notebook for the examples below is located in hephaistos/Notebooks/Time_related_transformation.ipynb. Let us load some data to illustrate time-related features engineering.

Input:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load data
csv_data = '../data/datasets/DailyDelhiClimateTrain.csv'
df = pd.read_csv(csv_data, delimiter=',')
print(df.head())
```

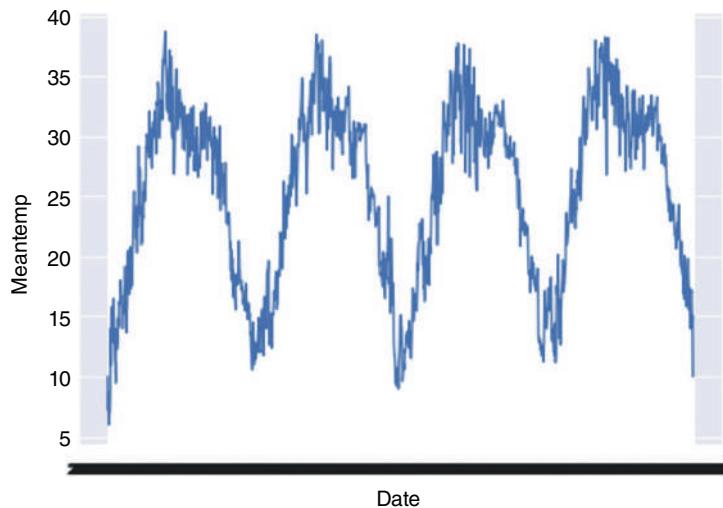
Output:

	date	meantemp	humidity	wind_speed	meanpressure
0	2013-01-01	10.000000	84.500000	0.000000	1015.666667
1	2013-01-02	7.400000	92.000000	2.980000	1017.800000
2	2013-01-03	7.166667	87.000000	4.633333	1018.666667
3	2013-01-04	8.666667	71.333333	1.233333	1017.166667
4	2013-01-05	6.000000	86.833333	3.700000	1016.500000

Input:

```
# Plot "meantemp" and "date"
sns.set_theme(style="darkgrid")
sns.lineplot(x="date", y="meantemp", data=df)
plt.show()
```

Output:



In this chapter, we will examine some scenarios that involve time-related data.

2.3.1 Date-Related Features

For many companies, the task of forecasting the sales of a product on a specific month or day can be useful. An initial process in addressing time-related data is to split the time information by day, month, year, hours, seconds, milliseconds, or other units. For instance, in the data above, the variable “date” gives us the day, month, and year of the different measures.

Input:

```
# Split the variable date
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')
df['year']=df['date'].dt.year
df['month']=df['date'].dt.month
df['day']=df['date'].dt.day
print(df.head())
```

Output:

	date	meantemp	humidity	wind_speed	meanpressure	year	month	day
0	2013-01-01	10.000000	84.500000	0.000000	1015.666667	2013	1	1
1	2013-01-02	7.400000	92.000000	2.980000	1017.800000	2013	1	2
2	2013-01-03	7.166667	87.000000	4.633333	1018.666667	2013	1	3
3	2013-01-04	8.666667	71.333333	1.233333	1017.166667	2013	1	4
4	2013-01-05	6.000000	86.833333	3.700000	1016.500000	2013	1	5

We have now loaded the dataset with pandas and have created a DataFrame with new columns (year, month, and day) for each observation in the series. Of course, we can adjust the time variables to more than just year, month, or day alone. We can combine time information with other features, such as the season of the month or semester, to improve the performance of our models. For instance, if our time stamp has hours and we want to study the road traffic, we can add variables such as business hours and non-business hours or the name of the day in the week. *DatetimeIndex* from pandas provides many attributes.

2.3.2 Lag Variables

Let us say we are predicting the stock price of a company. To make a prediction, we consider past values. The prediction of a value at time t will be impacted by the value at time $t - 1$. We need to create features to represent these past values. We call the past values lags, such that $t - 1$ is lag 1, $t - 2$ is lag 2, and so on. We can use the *shift()* method in pandas to create the lags.

Input:

```
df['lag_1'] = df['meantemp'].shift(1)
df = df[['date', 'lag_1', 'meantemp']]
print(df.head())
```

Output:

	date	lag_1	meantemp
0	2013-01-01	NaN	10.000000
1	2013-01-02	10.000000	7.400000
2	2013-01-03	7.400000	7.166667
3	2013-01-04	7.166667	8.666667
4	2013-01-05	8.666667	6.000000

In this example, we have generated a lag 1 feature for our variable meantemp. We do not really have a justification to do it here, but if we have data in which we wish to identify a weekly trend, for instance, we can create lag features for one week (Monday to Monday). We also can create multiple lag features (the sliding window approach). Let us say we desire lag 1 ($t - 1$) to lag 5 ($t - 5$).

Input:

```
df['lag_1'] = df['meantemp'].shift(1)
df['lag_2'] = df['meantemp'].shift(2)
df['lag_3'] = df['meantemp'].shift(3)
df['lag_4'] = df['meantemp'].shift(4)
df['lag_5'] = df['meantemp'].shift(5)

df = df[['date', 'lag_1', 'lag_2', 'lag_3', 'lag_4', 'lag_5', 'meantemp']]
print(df.head(10))
```

Output:

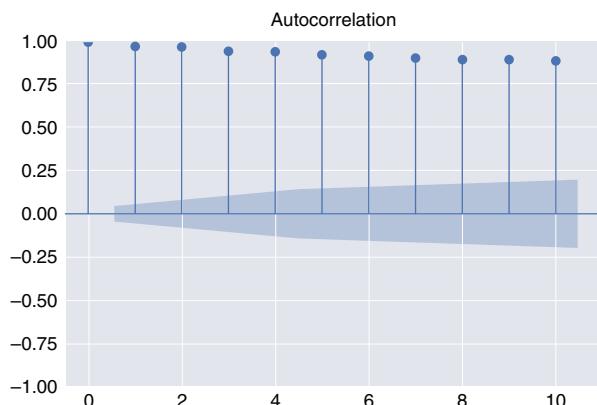
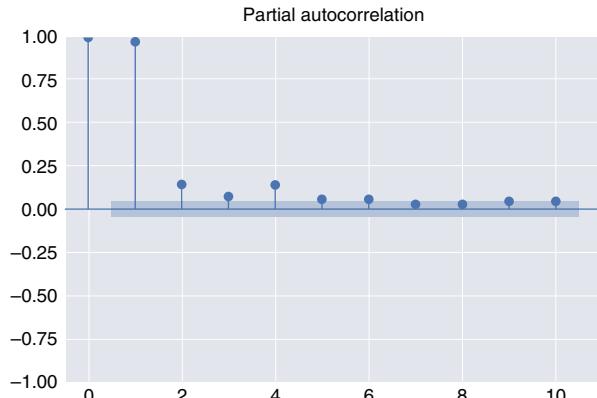
	date	lag_1	lag_2	lag_3	lag_4	lag_5	meantemp
0	2013-01-01	NaN	NaN	NaN	NaN	NaN	10.000000
1	2013-01-02	10.000000	NaN	NaN	NaN	NaN	7.400000
2	2013-01-03	7.400000	10.000000	NaN	NaN	NaN	7.166667
3	2013-01-04	7.166667	7.400000	10.000000	NaN	NaN	8.666667
4	2013-01-05	8.666667	7.166667	7.400000	10.000000	NaN	6.000000
5	2013-01-06	6.000000	8.666667	7.166667	7.400000	10.000000	7.000000
6	2013-01-07	7.000000	6.000000	8.666667	7.166667	7.400000	7.000000
7	2013-01-08	7.000000	7.000000	6.000000	8.666667	7.166667	8.857143
8	2013-01-09	8.857143	7.000000	7.000000	6.000000	8.666667	14.000000
9	2013-01-10	14.000000	8.857143	7.000000	7.000000	6.000000	11.000000

As we can see, the DataFrame contains “NaN,” which means Not a Number. We should discard the first rows of the data to train the models. We can perform a sensitivity analysis and use different numbers of lags or use lag values from the last month or last year. If we train a linear regression model, the model will assign weights to the lag features. We can also use an autocorrelation function (ACF), which measures the correlation between the time series and the lagged version of itself, and a partial autocorrelation function (PACF), which measures the correlation between the time series with a lagged version of itself after the elimination of the variations already explained by the intervening comparisons to determine the lag at which the correlation is significant.

Input:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(df['meantemp'], lags=10)
plot_pacf(df['meantemp'], lags=10)
```

Output:



The partial autocorrelation shows a high correlation with the first and second lag. The ACF shows a slow decrease, meaning that the future values have a very high correlation with past values. As we have seen, lag features are used to understand the behavior of a target value relative to the past, which may be a day, a week, or a month before. However, we must pay attention to the use of lag features, which can lead to overfitting if not used properly.

Aggregating features through statistics such as average, standard deviation, maximum, minimum, or skewness might be valuable additions to predict future behavior. Pandas provides the *aggregate* method to perform these calculations.

Input:

```
# Load data
csv_data = './data/DailyDelhiClimateTrain.csv'
df = pd.read_csv(csv_data, delimiter=',')

df['lag_1'] = df['meantemp'].shift(1)
df['lag_2'] = df['meantemp'].shift(2)
df['lag_3'] = df['meantemp'].shift(3)
df['lag_4'] = df['meantemp'].shift(4)
df['lag_5'] = df['meantemp'].shift(5)
```

```

lagged_feature_cols = ['lag_1', 'lag_2', 'lag_3', 'lag_4', 'lag_5']

# Create df_lagged_features to use for aggregation calculations
df_lagged_features = df.loc[:, lagged_feature_cols]

# Create aggregated features
df['max'] = df_lagged_features.aggregate(np.max, axis=1)
df['min'] = df_lagged_features.aggregate(np.min, axis=1)
df['mean'] = df_lagged_features.aggregate(np.mean, axis=1)
df['Standard Deviation'] = df_lagged_features.aggregate(np.std, axis=1)

# Drop first 5 rows due to NaNs
df = df.iloc[5: , :]

print(df.head(10))

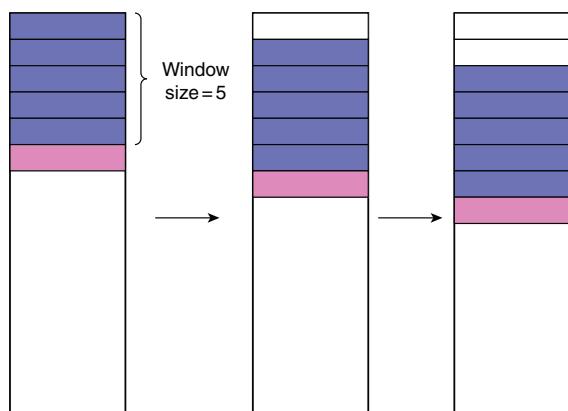
```

Output:

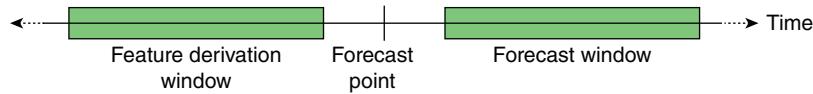
	date	meantemp	humidity	wind-speed	meanpressure	lag_1	lag_2	lag_3	lag_4	lag_5	max	min	mean	Standard Deviation
5	2013-01-06	7.000000	82.800000	1.480000	1018.000000	6.000000	8.666667	7.166667	7.400000	10.000000	10.000000	6.000000	7.846667	1.531448
6	2013-01-07	7.000000	78.600000	6.300000	1020.000000	7.000000	6.000000	8.666667	7.166667	7.400000	8.666667	6.000000	7.246667	0.956731
2	2013-01-08	8.857143	63.714286	7.142857	1018.714284	7.000000	7.000000	6.000000	8.666667	7.166667	8.666667	6.000000	7.166667	0.957427
8	2013-01-09	14.000000	51.250000	12.500000	1017.000000	8.857143	7.000000	7.000000	6.000000	8.666667	8.857143	6.000000	7.504762	1.219922
9	2013-01-10	11.000000	62.000000	7.400000	1015.666667	14.000000	8.857143	7.000000	7.000000	6.000000	14.000000	6.000000	8.571429	3.205544
10	2013-01-11	15.714286	51.285714	10.571429	1016.142857	11.000000	14.000000	8.857143	7.000000	7.000000	14.000000	7.000000	9.571429	2.974380
11	2013-01-12	14.000000	74.000000	13.228571	1015.571429	15.714286	11.000000	14.000000	8.857143	7.000000	15.714286	7.000000	11.314286	3.581984
12	2013-01-13	15.833333	75.166667	4.633333	1013.333333	14.000000	15.714286	11.000000	14.000000	8.857143	15.714286	8.857143	12.714286	2.744196
13	2013-01-14	12.833333	88.166667	0.616667	1015.166667	15.833333	14.000000	15.714286	11.000000	14.000000	15.833333	11.000000	14.109524	1.951916
14	2013-01-15	14.714286	71.857143	0.528571	1015.857143	12.833333	15.833333	14.000000	15.714286	11.000000	15.833333	11.000000	13.876190	2.036195

2.3.3 Rolling Window Feature

Rolling window features calculate statistics by selecting a window size, averaging the values in the selected window, and using the result as a feature. It is called rolling window (a new feature generated by the method) because the selected window slides with every next point.



To implement this method, we need to define the feature derivation window, which is a rolling window relative to a forecast point, and a forecast window, which is a range of the future values we want to predict.



We can calculate the mean of the previous seven values (*meantemp* in our data) and use that to predict the next value.

Input:

```
df['rolling_window_mean'] = df['meantemp'].rolling(window=7).mean()
df = df[['date', 'rolling_window_mean', 'meantemp']]
print(df.head(20))
```

Output:

	date	rolling_window_mean	meantemp
0	2013-01-01	NaN	10.000000
1	2013-01-02	NaN	7.400000
2	2013-01-03	NaN	7.166667
3	2013-01-04	NaN	8.666667
4	2013-01-05	NaN	6.000000
5	2013-01-06	NaN	7.000000
6	2013-01-07	7.604762	7.000000
7	2013-01-08	7.441497	8.857143
8	2013-01-09	8.384354	14.000000
9	2013-01-10	8.931973	11.000000
10	2013-01-11	9.938776	15.714286
11	2013-01-12	11.081633	14.000000
12	2013-01-13	12.343537	15.833333
13	2013-01-14	13.176871	12.833333
14	2013-01-15	14.013605	14.714286
15	2013-01-16	13.989796	13.833333
16	2013-01-17	14.775510	16.500000
17	2013-01-18	14.506803	13.833333
18	2013-01-19	14.292517	12.500000
19	2013-01-20	13.642857	11.285714

The use of the mean is not required, as we can also consider other metrics such as the sum, minimum value, or maximum value as features for the selected window. It is also possible to adjust data by using weights that can depend on the time of observation (e.g., giving higher weights to recent values).

2.3.4 Expanding Window Feature

Expanding window is a type of rolling window with the difference that the selected size of the window increases by one at every step as it considers a new value. The result of this process is that the size of the window expands and takes all past values into account.

Input:

```
# Load data
csv_data = './data/DailyDelhiClimateTrain.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

df['expanding_mean'] = df['meantemp'].expanding(7).mean()
df = df[['date', 'meantemp', 'expanding_mean']]
print(df.head(20))
```

Output:

	date	meantemp	expanding_mean
0	2013-01-01	10.000000	NaN
1	2013-01-02	7.400000	NaN
2	2013-01-03	7.166667	NaN
3	2013-01-04	8.666667	NaN
4	2013-01-05	6.000000	NaN
5	2013-01-06	7.000000	NaN
6	2013-01-07	7.000000	7.604762
7	2013-01-08	8.857143	7.761310
8	2013-01-09	14.000000	8.454497
9	2013-01-10	11.000000	8.709048
10	2013-01-11	15.714286	9.345887
11	2013-01-12	14.000000	9.733730
12	2013-01-13	15.833333	10.202930
13	2013-01-14	12.833333	10.390816
14	2013-01-15	14.714286	10.679048
15	2013-01-16	13.833333	10.876190
16	2013-01-17	16.500000	11.207003
17	2013-01-18	13.833333	11.352910
18	2013-01-19	12.500000	11.413283
19	2013-01-20	11.285714	11.406905

2.3.5 Understanding Time Series Data in Context

Alongside the methods described above, it is very important to understand domain-specific features to increase the performance of the models. This understanding is what will help us to define lag values and window sizes more accurately. We may also consider adding more external data to increase the value of the machine learning models. For instance, in road traffic, vacation periods are certainly a factor that influences the results. We need to consider trends, seasonality, cyclical, and irregularity.

Trends and seasonal components of a time series can also be added as individual features depending on context. To differentiate a selected seasonal trend, the `diff()` method can be used by setting the `periods` parameter to the length of the selected seasonal trend.

Input:

```
# Load data
csv_data = './data/DailyDelhiClimateTrain.csv'
df = pd.read_csv(csv_data, delimiter=',')

# 1 month difference
df['1month_diff'] = df['meantemp'].diff(periods=1)

# 24 months difference
df['24month_diff'] = df['meantemp'].diff(periods=24)

print(df.head(10))
```

Output:

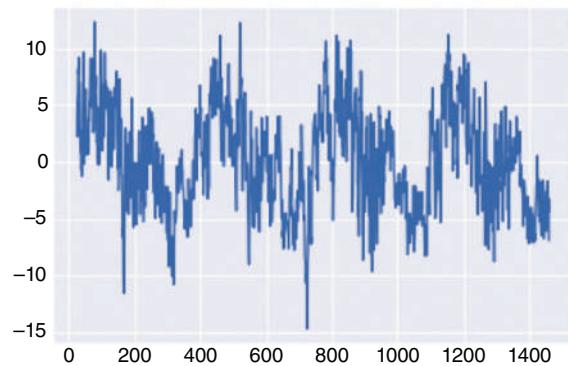
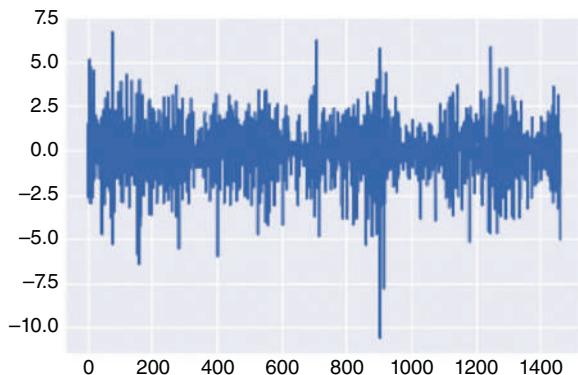
	date	meantemp	humidity	wind-speed	meanpressure	1month_diff	24month_diff
0	2013-01-01	10.000000	84.500000	0.000000	1015.666667	NaN	NaN
1	2013-01-02	7.400000	92.000000	2.980000	1017.800000	-2.600000	NaN
2	2013-01-03	7.166667	87.000000	4.633333	1018.666667	-0.233333	NaN
3	2013-01-04	8.666667	71.333333	1.233333	1017.166667	1.500000	NaN
4	2013-01-05	6.000000	86.833333	3.700000	1016.500000	-2.666667	NaN
5	2013-01-06	7.000000	82.800000	1.480000	1018.000000	1.000000	NaN
6	2013-01-07	7.000000	78.600000	6.300000	1020.000000	0.000000	1NaN
7	2013-01-08	8.857143	63.714286	7.142857	1018.714286	1.851743	NaN
8	2013-01-09	14.000000	51.250000	12.500000	1017.000000	5.142857	NaN
9	2013-01-10	11.000000	62.000000	7.400000	1015.166667	-3.000000	NaN

Input:

```
df['1month_diff'] = df['meantemp'].diff(periods=1)
plt.plot(df['1month_diff'])
plt.show()

df['24month_diff'] = df['meantemp'].diff(periods=24)
plt.plot(df['24month_diff'])
plt.show()
```

Output:



Applying square root, power, or log transformations to create a more normal distribution is also a possibility. If the data are stationary, which means that mean, variance, and autocorrelation structure do not change over time, we can consider each point as having been drawn from a normal distribution. We can use the square root to transform our dataset into a linear trend if the data increase quadratically. If the dataset grows exponentially, log (natural) transformation will make it linear.

Input:

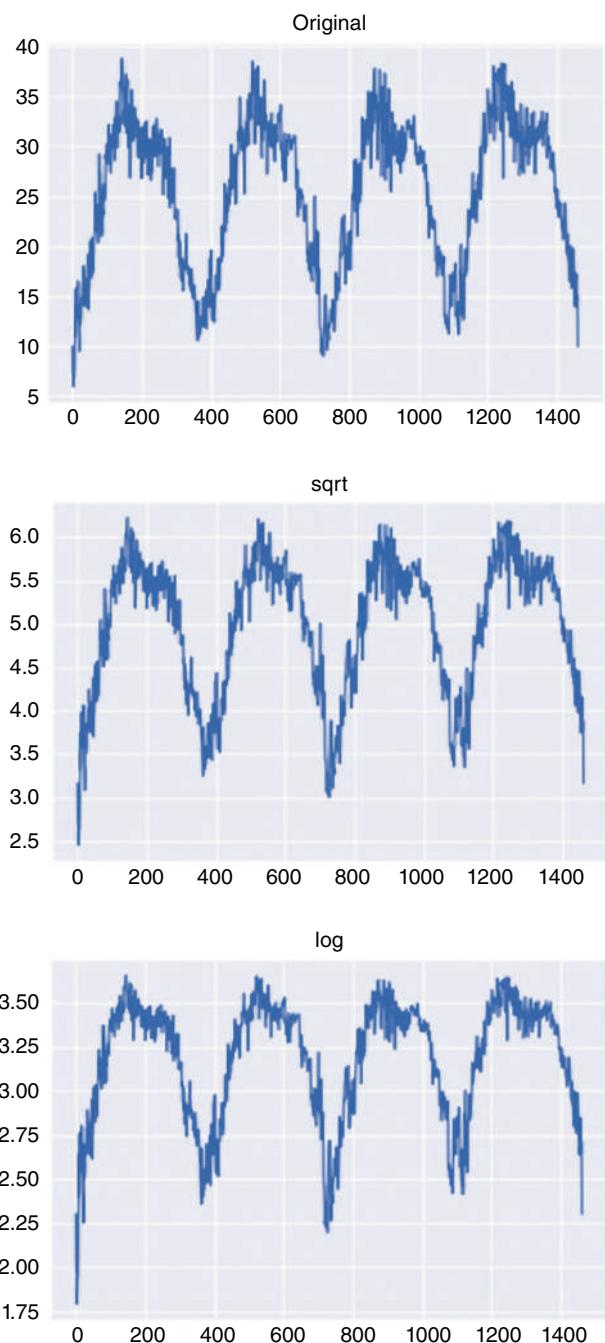
```
# Load data
csv_data = './data/DailyDelhiClimateTrain.csv'
df = pd.read_csv(csv_data, delimiter=',')

plt.plot(df['meantemp'])
plt.title("Original")
plt.show()

df['sqrt'] = np.sqrt(df['meantemp'])
plt.plot(df['sqrt'])
plt.title("sqrt")
plt.show()

df['log'] = np.log(df['meantemp'])
plt.plot(df['log'])
plt.title("log")
plt.show()
```

Output:



The transformation does not really change the shape of the data because the data points neither quadratically nor exponentially increase.

The techniques above allow us to convert our time series problem into a supervised machine learning problem. However, we need to pay attention when creating the validation and test sets for time series. Usually, the validation and test subsets are randomly selected. In time series, a data point is dependent on past values, meaning that if we select our subsets randomly, we might train our model on future data and predict past values (look-ahead bias). We should split the time variable to avoid this situation. It is recommended to select training, validation, and testing procedures in that exact order according to time, with the test set using the most recent data. Recurrent neural networks are the most traditional and accepted algorithms for problems based on time series forecasting.

To transform time series with hephAlistos, we need to provide inputs to the following parameters:

- **time_transformation:** To transform time series data, we can use different techniques such as lag, rolling window, or expending window. For example, to use lag, we need to set the time_transformation as follows: time_transformation = "lag."
- If lag is selected, we need to add the following parameters:
 - **number_of_lags:** An integer defining the number of lags we desire
 - **lagged_features:** The features for which we want to apply lag
 - **lag_aggregation:** The aggregation method. For aggregation, the following options are available: "min," "max," "mean," "std," or "no." Several options can be selected at the same time.
- If rolling_window is selected:
 - **window_size:** An integer indicating the size of the window
 - **rolling_features:** The features for which to apply rolling window.
- If expending_window is selected:
 - **expending_window_size:** An integer indicating the size of the window
 - **expending_features:** To select the features we want to apply expending rolling window

We can go to hephAlistos main folder and create a new Python file to insert the code below.

Input:

```
from ml_pipeline_function import ml_pipeline_function

from data.datasets import DailyDelhiClimateTrain
df = DailyDelhiClimateTrain()
df = df.rename(columns={"meantemp": "Target"})

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method = 'row_removal',
test_time_size = 365, time_feature_name = 'date', time_format = "%Y-%m-%d",
time_split = ['year','month','day'], time_transformation='lag',number_of_lags=2,
lagged_features = ['wind_speed', 'meanpressure'], lag_aggregation
= ['min', 'mean'])
```

2.4 Handling Missing Values in Machine Learning

The issue of missing data in machine learning has been largely overlooked, but it affects data analysis across a wide range of domains. The handling of missing values is also a key step in the preprocessing of a dataset, as many machine learning algorithms do not support missing values. In addition, making the correct decision regarding missing values can generate robust data models. Missing values are due to different causes such as incomplete extraction, data corruption, or issues in loading the dataset.

There is a large set of methods to address missing values, ranging from simple ones such as deleting the rows containing missing values or imputing them for both continuous and categorical variables to more complex ones such as the use of machine and deep learning algorithms to impute missing values.

Let us create a class called *missing* in a file named *missing.py* to address missing values.

Input:

```
# Importing all packages required for imputation
import sys
import numpy as np
import pandas as pd
import inputs
```

```
# Class for handling missing values algorithm module
class missing:
    """
    df : original dataframe
    """
    def __init__(self, read_dataframe, write_dataframe):
        """
        Args:
            input_dataframe : dataframe to be read and perform missing values removal
            algorithm
            output_dataframe : dataframe to be written into after performing the algorithm
        """
        self.df = read_dataframe
        print('Data read successfully !', f'Shape of original file : {self.df.shape}')
        self.input_dataframe = read_dataframe
        self.output_dataframe = write_dataframe
```

2.4.1 Row or Column Removal

A simple method that is commonly used to address missing values is to delete the corresponding row. We can also delete a complete feature if we observe that a large portion of the data is missing or incorrect. The size of the dataset will impact the output of this method. Obviously, the drawback of deleting information is that it works poorly if the size of the dataset is small or if the percentage of missing values is high. The loss of information needs to be assessed. On the other hand, an advantage is that we will generate more robust and accurate models if we remove the maximum number of incorrect variables and missing values. The more we can delete features that do not provide specific information, the better it will be for the model, as it does not really have a high weightage. Python's pandas library provides all the necessary functions to remove rows and columns from a data frame that contains missing values.

To detect missing values, we can add the following function to the *class missing*:

Input:

```
# function for detecting missing values and reporting it
def detect_missing(self):
    # checking missing values
    null_series = self.df.isnull().sum()
    print()
    null_column_list = []
    if sum(null_series):
        print('Following columns contains missing values : ')
        total_samples = self.df.shape[0]
        for i, j in null_series.items():
            print(j)
```

```

if j:
    print("{} : {:.2f} %".format(i, (j/total_samples)*100))
    null_column_list.append(i)
else:
    print("None of the columns contains missing values !")
return null_column_list

```

To remove rows or columns containing missing values, we can use `pandas.DataFrame.dropna`, which determines whether rows or columns that contain missing values are removed; “axis = 0” drops rows whereas “axis = 1” drops columns.

Input:

```

# using row removal
def row_removal(self):
    original_row, original_col = self.df.shape[0], self.df.shape[1]
    print()
    print('Using row removal algorithm...')
    # removing rows
    df_row = self.df.dropna(axis=0)
    print(f"Shape of new dataframe : {df_row.shape}")
    print(f"Total {original_row - df_row.shape[0]} rows removed")
    return df_row

# using column removal
def column_removal(self):
    original_row, original_col = self.df.shape[0], self.df.shape[1]
    print()
    print('Using column removal algorithm...')
    print('Warning : Features may be reduced, introducing inconsistency when
Testing !')
    # removing columns
    df_col = self.df.dropna(axis=1)
    print(f"Shape of new dataframe : {df_col.shape}")
    print(f"Total {original_col - df_col.shape[1]} columns removed")
    return df_col

```

2.4.2 Statistical Imputation: Mean, Median, and Mode

Replacing missing values by the mean, median, or mode can be applied to features that have numeric data. The mean imputation will replace any missing value from a variable with the mean of that variable for all other cases, which has the great advantage of not changing the sample mean of the variable. The drawback of using the mean is that it will attenuate any correlation involving the imputed variables. Mode is frequently used to impute missing values; it works with categorical features by replacing the missing value with most frequent category within the variable. These are obviously approximations, but they can improve model performance by avoiding data removal. There are different possibilities for optimizing the results such as using the deviation of neighboring values when the data are linear.

Input:

```

# using statistical imputation
def stats_imputation(self, null_column_list):
    print()

```

```

print('Using Statistical imputation algorithm...')

# extracting columns for numerical columns
valid_cols = [column for column in null_column_list if self.df[column].dtype != 'object']

# extracting columns for categorical columns
categorical_cols = [column for column in null_column_list if self.df[column].dtype == 'object']

numeric_cols = valid_cols

df_stats_mean, df_stats_median, df_stats_mode = self.df.copy(), self.df.copy(), self.df.copy()

# Imputing mean for numeric values and then imputing median and mode for categorical values
print(f'Imputing following columns with mean, median and mode : {numeric_cols}')
print(f'Imputing following columns with mode : {categorical_cols}')

if len(numeric_cols):
    for i in numeric_cols:
        df_stats_mean.fillna({i : self.df[i].mean()}, inplace=True)
        df_stats_median.fillna({i : self.df[i].median()}, inplace=True)
        df_stats_mode.fillna({i : self.df[i].mode()[0]}, inplace=True)

if len(categorical_cols):
    for i in categorical_cols:
        df_stats_mean.fillna({i : self.df[i].mode()[0]}, inplace=True)
        df_stats_median.fillna({i : self.df[i].mode()[0]}, inplace=True)
        df_stats_mode.fillna({i : self.df[i].mode()[0]}, inplace=True)

return df_stats_mean, df_stats_median, df_stats_mode

```

2.4.3 Linear Interpolation

Interpolation works well for a time series with some type of trend but is not suitable for seasonal data. This technique attempts to estimate values from other observations within the range of a discrete set of known data points. In other words, it will adjust a function to the data and will use this function to extrapolate missing data. The simplest type of interpolation is linear interpolation, which is a method of approximating value by joining points in increasing order along a straight line. It takes a mean between the values before the missing data and the values after it.

Input:

```

# using linear interpolation
def linear_interpolation(self, null_column_list):
    print()
    print('Using Linear Interpolation imputation algorithm...')

    # extracting columns for numerical columns
    valid_cols = [column for column in null_column_list if self.df[column].dtype != 'object']

    # extracting columns for categorical columns
    categorical_cols = [column for column in null_column_list if self.df[column].dtype == 'object']

    numeric_cols = valid_cols

```

```

df_linear_interpolation = self.df.copy()
# Linear interpolation for numeric values
print(f'Imputing following columns with linear interpolation :
{numeric_cols}')

if len(numeric_cols):
    for i in numeric_cols:
        df_linear_interpolation[numeric_cols] = df_linear_interpolation[numeric_cols].interpolate(method='linear', limit_direction='forward', axis=0)

if len(categorical_cols):
    for i in categorical_cols:
        df_linear_interpolation[numeric_cols] = df_linear_interpolation[numeric_cols].interpolate(method='linear', limit_direction='forward', axis=0)

return df_linear_interpolation

```

2.4.4 Multivariate Imputation by Chained Equation Imputation

In multiple imputation, several different completed sets of data are generated and each missing value is replaced by several different values. The procedure has different steps:

- The dataset with missing values is duplicated several times.
- The missing values of each duplicate are replaced with imputed values. Due to random variation across copies, slightly different values are imputed.
- Each multiple imputed dataset is analyzed, and the results are combined.

The use of multiple imputations, as opposed to single imputations, accounts for statistical uncertainty in the imputations (Figure 2.11).

Multivariate imputation by chained equation (MICE), also known as fully conditional specification, sequential regression multiple imputation, or Gibbs sampling, has emerged in the statistical literature as one appealing method to replace missing values via multiple imputations. To make it concrete, let us imagine we have a dataset composed of the variables age, gender, and income. Each of the variables has missing values. The MICE method assumes that these data points are missing at random (MAR). This assumption considers that the probability of the missing data for a variable is related to other measured variables but unrelated to the variable with the missing value itself. In our example, this means that the probability that the income variable is missing depends on other observed values but not on unobserved values for income. In other words, the income data values are missing perhaps because a certain gender is less likely to respond to a survey. The missing data points are not related to the level of income itself. We also have other categories, such as missing completely at random (MCAR) when the data are missing values completely at random, unrelated to any other variables (including the variable with missing value itself). Values may be missing because we have lost information during transportation. There is also the missing not at random (MNAR) category, which indicates that the missing values for a variable are related to the variable with the missing values itself.

The following steps would be an approach to apply MICE:

- The missing variables are imputed using, for instance, mean imputation (any missing value is replaced by the mean observed value for that specific variable).
- The imputed mean values for age are set back to missing.
- A linear regression is run for age predicted by income and gender, using all cases in which age was observed.
- The results of this analysis and the prediction of the missing age values are imputed. Age should not have any missing values at this stage.

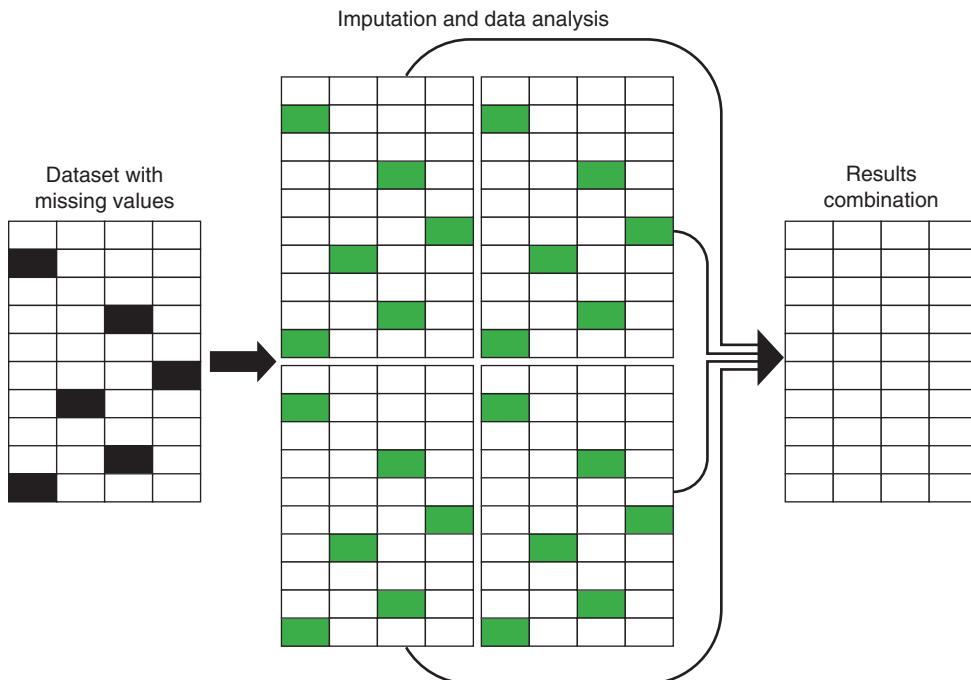


Figure 2.11 Multiple imputation.

- The previous three steps are repeated for the income variable. The linear regression is performed with income predicted by age and gender. The same steps are performed for the gender variable with a logistic regression of gender on age and income.
- The entire process is repeated until convergence. The final set of imputed values constitutes a complete dataset.

The properties of MICE make it particularly useful for large imputation procedures. The technique is highly flexible, as it can process variables of varying types such as binary or continuous, as well as complexities such as bounds or survey skip patterns. Depending on the tools we use, the implementation of MICE can vary; we can model each variable according to its distribution with a multinomial logit model for categorical variables, a Poisson model for count variables, logistic regression for binary variables, or linear regression for continuous variables. The regression models use information from all other variables. Residual error is added to create the imputed values and add sampling variability to the imputations. Residual variance can also be added to the parameter estimates of the regression.

As an example, we could use the *fancyimpute* library to apply the MICE method on continuous variables.

Input:

```
def mice(self):
    from fancyimpute import IterativeImputer
    mice_imputer = IterativeImputer()
    df_mice = mice_imputer.fit_transform(self.df)
    return df_mice
```

2.4.5 KNN Imputation

The KNN imputation method uses the k-nearest neighbors algorithm to replace missing values by identifying the neighboring points through a measure of distance, by default the Euclidean distance. The missing values can be estimated using

completed values of neighboring observations. For example, we can use the mean value from the nearest k neighbors (`n_neighbors`) in the dataset. This method can be used for continuous, discrete, and categorical data.

To implement it, we can use either `fancyimpute` or `scikit-learn`.

Input:

```
from fancyimpute import KNN
knn_imputer = KNN()
df = knn_imputer.fit_transform(df)
```

Let us continue our `class missing`, created at the beginning of the chapter.

Input:

```
# using KNN
def knn(self):
    print()
    print('Using KNN imputation algorithm...')
    from sklearn.impute import KNNImputer
    KNN_imputer = KNNImputer(n_neighbors=5)
    df_knn = self.df.copy(deep=True)
    df_knn.iloc[:, :] = KNN_imputer.fit_transform(df_knn)
    return df_knn
```

We can use many methods, including machine learning algorithms such as XGBoost, LightGBM, or random forests, to address missing values. We need to explore what is the most appropriate for our context.

To finish our class `missing`, we can add the following lines to our code:

```
# main executing functions
def missing_main(self):

    # printing missing detected values
    try:
        null_column_list = self.detect_missing()
    except:
        print("Something went wrong with null_column_list: Please Check")

    if inputs.selected_method[0] == 'row_removal':
        # applying row removal
        try:
            df_row = self.row_removal()
            print(df_row)
            return df_row
        except:
            print("Something went wrong with df_row: Please Check")

    if inputs.selected_method[0] == 'column_removal':
        # applying column removal
        try:
            df_col = self.column_removal()
            print(df_col.shape)
        except:
            print("Something went wrong with df_col: Please Check")
```

```

        return df_col
    except:
        print("Something went wrong with df_col: Please Check")

    if inputs.selected_method[0] == 'stats_imputation_mean':
        # applying statistical imputation Mean + Mode
        try:
            df_stats_mean = self.stats_imputation_mean(null_column_list)
            return df_stats_mean
        except:
            print("Something went wrong with stats_imputation_mean: Please Check")

    if inputs.selected_method[0] == 'stats_imputation_median':
        # applying statistical imputation Median + Mode
        try:
            df_stats_median = self.stats_imputation_median(null_column_list)
            return df_stats_median
        except:
            print("Something went wrong with stats_imputation_median: Please Check")

    if inputs.selected_method[0] == 'stats_imputation_mode':
        # applying statistical imputation Median + Mode
        try:
            df_stats_mode = self.stats_imputation_mode(null_column_list)
            return df_stats_mode
        except:
            print("Something went wrong with stats_imputation_mode: Please Check")

    if inputs.selected_method[0] == 'linear_interpolation':
        # applying linear interpolation
        try:
            df_interpolation_linear = self.linear_interpolation(null_column_list)
            return df_interpolation_linear
        except:
            print("Something went wrong with interpolation_linear: Please Check")

    if inputs.selected_method[0] == 'mice':
        # applying MICE
        try:
            df_mice = self.mice()
            return df_mice
        except:
            print("Something went wrong with MICE: Please Check")

    if inputs.selected_method[0] == 'knn':
        # applying KNN
        try:
            df_knn = self.knn()
            return df_knn

```

```

except:
    print("Something went wrong with KNN: Please Check")

# main driver function
if __name__ == '__main__':
    print('HANDLING MISSING VALUES')
    # runs if and only if it contains at least input file, output file
    if len(sys.argv) == 3:
        read_dataframe = sys.argv[1]
        write_dataframe = sys.argv[2]

        print(f"file given : {read_dataframe}")
        m = missing(read_dataframe, write_dataframe)
        m.missing_main()
    else:
        print('Something went wrong: Please Check')

```

We can then create a file called inputs.py as follows:

Input:

```

# Load data
input_file = './data/missing.csv'
df = pd.read_csv(input_file, delimiter=';')

"""
Selected_method: "row_removal", "column_removal", "stats_imputation_mean",
"stats_imputation_median", "stats_imputation_mode", "linear_interpolation", "mice",
"knn"
"""

selected_method = ['knn']

```

Then, we can launch the code (main.py).

Input:

```

import os
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import inputs
from missing import missing

df = inputs.df

# We split our data to y (Target) and X (features)

y = df.loc[:, df.columns == 'Target']

```

```

X = df.loc[:, df.columns != ('Target')]
# Split data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

m = missing(X_train, X_train)
X_train = m.missing_main()
print(X_train)

```

To address missing values in the sample with hephAistos, we can go to the hephAistos main folder, create a new Python file, and insert the code example below for a new pipeline. The following options are necessary:

- **missing_method:** Different methods can be selected. Options are “row_removal,” “column_removal,” “stats_imputation_mean,” “stats_imputation_median,” “stats_imputation_mode,” “linear_interpolation,” “mice,” and “knn.”

Input:

```

from ml_pipeline_function import ml_pipeline_function

# Import dataset
from data.datasets import neurons
df = neurons()

ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal')

```

2.5 Feature Extraction and Selection

In feature extraction, the objective is to reduce an initial set of raw data into more manageable groups for running our machine learning models. We speak about dimension reduction in this context. Feature selection is the process of identifying and selecting the most relevant subset of input features to explain the target variable. It is generally accepted that generating features in a model can be beneficial, but we also need to take care to exclude irrelevant features. For instance, in the case of weather data, the scores of the Portuguese or French soccer teams during the year are irrelevant for predicting the temperature. Both feature selection and feature extraction are used to dimensionally reduce initial raw data, allowing reduction of the complexity and overfitting of a model. As a data scientist, it is critically important to master these techniques to delete irrelevant features and reduce overfitting.

2.5.1 Feature Extraction

Large datasets can require more computing resources to process. In this book, we address the point of putting the machine and deep learning models into production. Taking care of computational resources is also an important topic. The advantage of feature extraction is that we can select or combine variables into features and reduce the amount of redundant data. This process will considerably reduce the amount of data that needs to be processed. Of course, the idea is not to compromise accuracy but rather to reduce data and at the same time maintain accuracy or even improve it. To summarize, feature extraction can avoid the risk of overfitting, speed up training, improve accuracy, and provide better data visualization and explainability of our models. Feature extraction is used in image processing to detect features in an image or video; it can be a shape or a motion. Another practical case is natural language processing with what is called a bag of words. The idea is to extract the words (features) in a target text and classify them by frequency of use.

Autoencoders, unsupervised learning of efficient data coding, is also a good application in which feature extraction can help identify important features for coding.

In this section, we will discuss some of the linear and nonlinear dimensionality reduction techniques that are widely used in a variety of applications, including PCA, independent component analysis (ICA), linear discriminant analysis (LDA), and locally linear embedding (LLE). Once we understand how these methods work, we can explore many more methods such as canonical correlation analysis (CCA), singular value decomposition (SVD), CUR matrix decomposition, compact matrix decomposition (CMD), non-negative matrix factorization (NMF), kernel PCA, multidimensional scaling (MDS), isomap, Laplacian Eigen map, local tangent space alignment (LTSA), or fast map.

All code examples provided to explain feature extraction can be found in hephaistos/Notebooks/Feature_extraction.ipynb or https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Feature_extraction.ipynb.

From a terminal in hephaistos/Notebooks, we can open Jupyter Notebook (the command line is jupyter notebook) and then open Feature_extraction.ipynb in a browser to run the different code examples.

2.5.1.1 Principal Component Analysis

Data transformation using unsupervised learning is a common method to better visualize, compress, and extract the most informative data. PCA is a well-known algorithm and is widely used to transform correlated features into features that are not statistically correlated. This transformation is followed by the selection of a subgroup of new features, classified by order of importance, that can best summarize the original data distribution. It allows us to reduce the original dimensions of a dataset. The basic idea is that PCA will study p variables measured for n individuals. When n and p are large, the aim is to reduce the number of features of a dataset while preserving as much information as possible. In the context of unidimensional or bidimensional studies, we can use graphical tools such as histograms or box plots and mean, variance, and correlation for numerical summaries. In a multidimensional context, this does not consider eventual relationships between variables. Even if we commonly use PCA for dimension reduction, we can also consider PCA as a method for multivariate outlier detection.

To summarize, PCA in the context of machine learning is a series of steps such as data rescaling, computation of the covariance matrix to identify correlations, computation of the eigenvectors and eigenvalues of the covariance matrix to identify the principal components, and creation of a feature vector to select the principal components to keep and recast the data along the principal component axes.

As PCA will calculate a new projection of our dataset, the new axis is based on the standard deviation of our variables. Therefore, we need to perform data rescaling to have the same standard deviation; thus, all variables will have the same weight, contributing equally, allowing calculation of the relevant axis. If we do not perform data rescaling, variables with larger ranges will take precedence over variables with smaller ranges, leading to biased results.

We can use standardization, which removes the mean and scales to unit variance:

$$z = \frac{x - \text{mean}(x)}{\text{standard deviation}(x)}$$

We then need to calculate the covariance matrix for the entire dataset. Let us first examine the difference between variance and covariance. Variance is a measure of the variation of a single random variable, for example, the weight of a person in a population. Covariance will measure how much two random variables vary together, for instance, the weight and the height of a person in a population.

The variance is calculated by the following formula:

$$\sigma_x^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

where x is the random variable, n is the number of samples, and \bar{x} is the mean.

The covariance is given by the following formula:

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

where x and y are two random variables.

The covariance matrix of a 2D case is described by the following:

$$C = \begin{pmatrix} \sigma(x,x) & \sigma(x,y) \\ \sigma(y,x) & \sigma(y,y) \end{pmatrix}$$

The diagonal entries are the variances, and the rest are the covariances. We then need to calculate the eigenvalues and eigenvectors of the covariance matrix. The objective of this step is to identify the principal components that are new, uncorrelated variables created from linear combinations of the initial variables. Most of the information is concentrated in the first components. Let us say we have 20 features; it will give us 20 principal components. The maximum information is in the first component (the largest possible variance), the maximum remaining information is in the second, and so on.

Eigenvectors constitute a set of vectors whose direction remains unchanged when we apply a linear transformation, and an eigenvalue is the factor by which the eigenvector is scaled. If v is an eigenvector of A and λ is the corresponding eigenvalue, we can write the following expression: $Av = \lambda v$. If we put all eigenvalues in a diagonal matrix L and all eigenvectors in a matrix V , we can state that $CV = VL$ where C is the covariance matrix, represented as $C = VLV^{-1}$.

The equation $Av = \lambda v$ can be stated equivalently as $(A - \lambda I)v = 0$, where I is the identity matrix (n by n) and 0 is the zero vector. Because v is a nonzero vector, the equation can be resolved if and only if the determinant of the matrix $(A - \lambda I)$ is zero: $|A - \lambda I|$.

Calculating the determinant will allow identification of the eigenvalues. We can then resolve the equation $(A - \lambda I)v = 0$ to find the eigenvectors using the eigenvalues. The next steps we need to perform are sorting the eigenvalues and their corresponding eigenvectors, choosing k eigenvalues, and forming a new matrix of eigenvectors to transform our data (features matrix $\times k$ eigenvectors).

For the purpose of this section, we will use the mushroom classification dataset coming from Kaggle (<https://www.kaggle.com/uciml/mushroom-classification>). This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the *Agaricus* and *Lepiota* genera, drawn from *The Audubon Society Field Guide to North American Mushrooms* (1981). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class has been combined with the poisonous one. The *Guide* clearly states that there is no simple rule (such as “leaflets three, let it be” for poison oak and ivy) for determining the edibility of a mushroom.

Now that we have seen an overview of the theory, let us code and use the mushroom data.

Input:

```
import pandas as pd
csv_data = '../data/datasets/mushrooms.csv'
df = pd.read_csv(csv_data, delimiter=',')
print(df)
```

Output:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	population	habitat
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	s	u
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	n	g
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	n	m
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	s	u
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	a	g
...
8119	e	k	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	b	c	l
8120	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	n	o	p	b	v	l
8121	e	f	s	n	f	n	a	c	b	n	...	s	o	o	p	o	o	p	b	c	l
8122	p	k	y	n	f	y	f	c	n	b	...	k	w	w	p	w	o	e	w	v	l
8123	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	o	c	l

To proceed, we need to encode the categorical variables. We will use label encoding. One very important thing to do is to perform feature rescaling, as PCA is highly affected by scale. Using StandardScaler will standardize the features into a unit scale and limit the effects.

Input:

```
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder

# Encode the data
enc = LabelEncoder()
df = df.apply(enc.fit_transform)

# Divide the data, y the variable to classify and X the features
y = df.loc[:, df.columns == 'class'].values.ravel()
X = df.loc[:, df.columns != 'class']

# Standardize the data
X = StandardScaler().fit_transform(X)
print(X)
```

Output:

```
array( [ [ 1.02971224,  0.14012794, -0.19824983, ..., -0.67019486,
          -0.5143892 ,  2.0.3002809], ,
        [ 1.02971224,  0.14012794,   1.76587407, ..., -0.2504706 ,
         -1.31310821, -0.29572966], ,
        [-2.08704716,  0.14012794,   1.37304929, ..., -0.2504706 ,
         -1.31310821,  0.86714922], ,
        ...,
        [-0.8403434 ,  0.14012794, -0.19824983, ..., -1.50964337,
         -2.11182722,  0.28570978], ,
        [-0.21699152,  0.95327039, -0.19824983, ...,  1.42842641,
         0.028432981,  0.28570978], ,
        [ 1.02971224,  0.14012794, -0.19824983, ...,  0.16925365,
         -2.11182722,  0.28570978] ] )
```

The original data have 22 columns that we will reduce to two dimensions with the following few lines, using scikit-learn:

Input:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(X)
PCA_df = pd.DataFrame(data = principalComponents, columns = ['principal component 1',
'principal component 2'])

# Concatenation of PCA data and Target
df_class = pd.DataFrame(y, columns = ["Target"])
PCA_df = pd.concat([PCA_df, df_class], axis = 1)
print(PCA_df)
```

Output:

principal	component 1	principal component 2	target
0	-0.574322	-0.975780	1
1	-2.282102	0.279064	0
2	-1.858036	-0.270974	0
3	-0.884780	-0.756470	1
4	0.689613	1.239266	0
...
8119	-2.244846	-0.918548	0
8120	-2.538760	-1.671722	0
8121	-1.621516	-0.757537	0
8122	3.670606	-1.032775	1
8123	-1.575203	-1.228580	0

Let us plot the data using matplotlib.

Input:

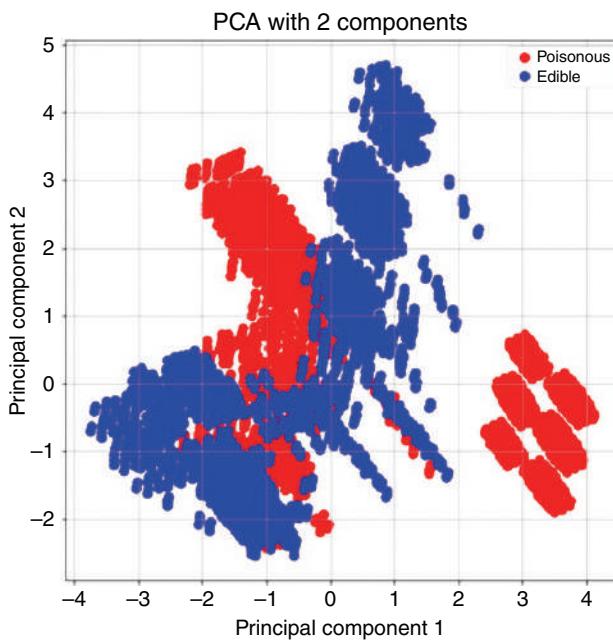
```
from matplotlib.pyplot import figure

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('PCA with 2 components', fontsize = 20)

targets = [1, 0]
colors = ['r', 'b']

for target, color in zip(targets,colors):
    indicesToKeep = PCA_df['Target'] == target
    ax.scatter(PCA_df.loc[indicesToKeep, 'principal component 1']
               , PCA_df.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(['Poisonous', 'Edible'])
ax.grid()
```

Output:



The metric to choose the number of components is called the explained variance ratio, which is the percentage of variance that is contributed by each of the selected components. Ideally, we need to reach a total of 80% to avoid overfitting.

```
pca.explained_variance_ratio_
```

The output of the line of code above shows that our first principal component contains 18.5% of the variance and the second principal component contains 12.4%. Together, the two components contain 30.9% of the information, which is not sufficient. More components need to be taken.

Although technically we can use PCA on label-encoded data or binary data, it does not perform well and can produce poor results; PCA is adapted to continuous variables. Squared deviation (minimize variance) is not significant on categorical data. Alternative methods should be used.

2.5.1.2 Independent Component Analysis

The difference between PCA and ICA is that PCA compresses information and ICA separates information. Both require autoscaling but ICA can benefit from first applying PCA. To describe ICA, statisticians use the example of the well-known “cocktail party problem.” Imagine we are at a cocktail party and that two people are speaking simultaneously. We have two microphones that we have placed in different locations. As voices are heard by both microphones at different volumes because of the distance, when we look at the records of the two microphones, the data are mixed and we have two recorded time signals, $x_1(t)$ and $x_2(t)$, where x_1 and x_2 are the amplitudes and t is the time index. The question is how to separate the recordings of each speaker. Each of the recorded signals is a weighted sum of the signals emitted by the two speakers, $s_1(t)$ and $s_2(t)$. All of this can be translated into a linear equation. To get there, we first need to make two assumptions to apply ICA. The independent components that are hidden and that we are trying to isolate must be statistically independent and non-Gaussian. Two events or distributions are statistically independent if their joint probabilities equal the product of their individual probabilities: $p(x,y) = p(x)p(y)$, where $p(x)$ is the probability distribution of x and $p(x,y)$ the joint probability distribution of x and y . In ICA, the distributions must be non-Gaussian.

ICA is used not only to isolate voices but also in many other applications such as electroencephalograms (EEGs), which are the records of brain electrical fields, magnetoencephalograms (MEGs), which are the records of brain magnetic fields, or functional magnetic resonance imaging (fMRI) analysis with the same objectives of the voice isolation

above to separate useful signals from unhelpful ones. Many biological artifacts reflect non-Gaussian processes. We can also consider improving Siri or Alexa or isolating sounds from others as use cases.

Let us generate some data as sinusoidal (s1), square (s2), or sawtooth (s3) signals. Then, we can standardize and mix the data with a mixing matrix.

Input:

```

import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from sklearn import preprocessing

np.random.seed(0) # set seed for reproducible results
n_samples = 3000
time = np.linspace(0, 10, n_samples)

# Sinusoidal signal
s1 = np.sin(2 * time)
# Square signal
s2 = np.sign(np.sin(2 * time))
# Sawtooth signal
s3 = signal.sawtooth(2 * np.pi * time)

S = np.c_[s1, s2, s3]
S += 0.15 * np.random.normal(size=S.shape) # Add noise

# Standardize the data
S = StandardScaler().fit_transform(S)

# Let's mix data with mixing matrix
M = np.array([[2, 1, 2], [1, 0.5, 3], [0.5, 1.5, 2]])
# Generate mixed observations
X = np.dot(S, M.T)

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

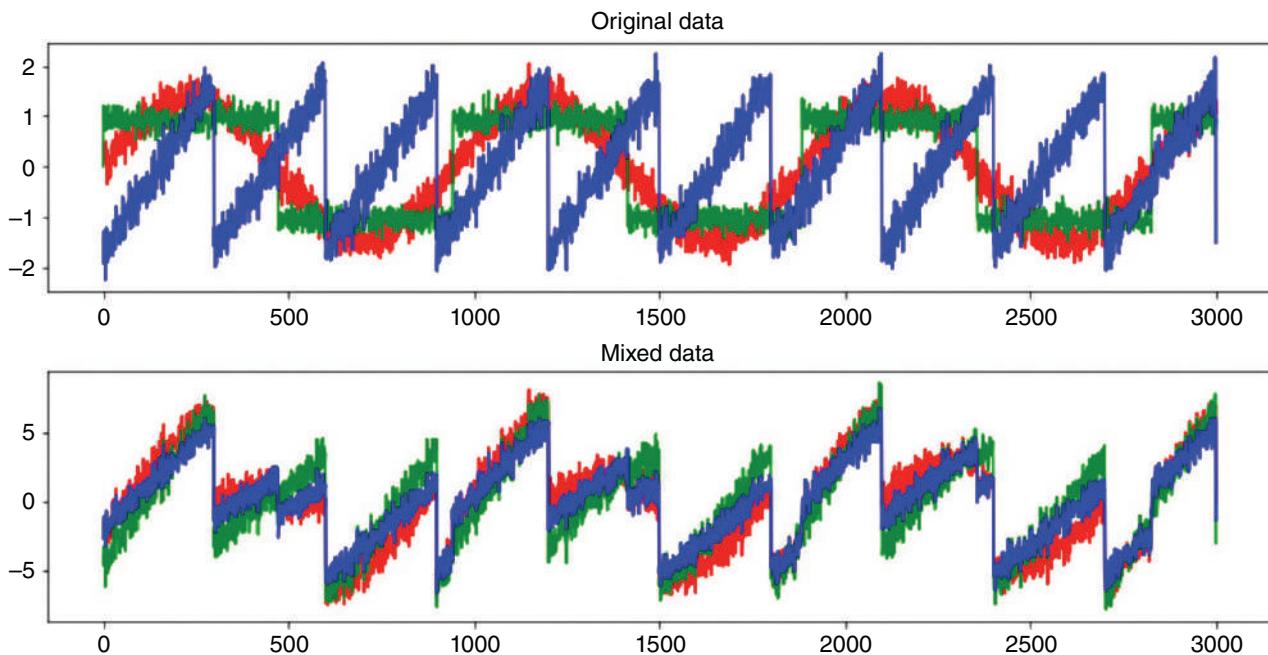
models = [S, X]
names = ['Orginal Data',
         'Mixed Data']
colors = ['red', 'green', 'blue']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

plt.tight_layout()

```

Output:



Let us now see how PCA behaves by modifying the code.

Input:

```
from sklearn.decomposition import PCA
# compute PCA
pca = PCA(n_components=3)
P = pca.fit_transform(X)

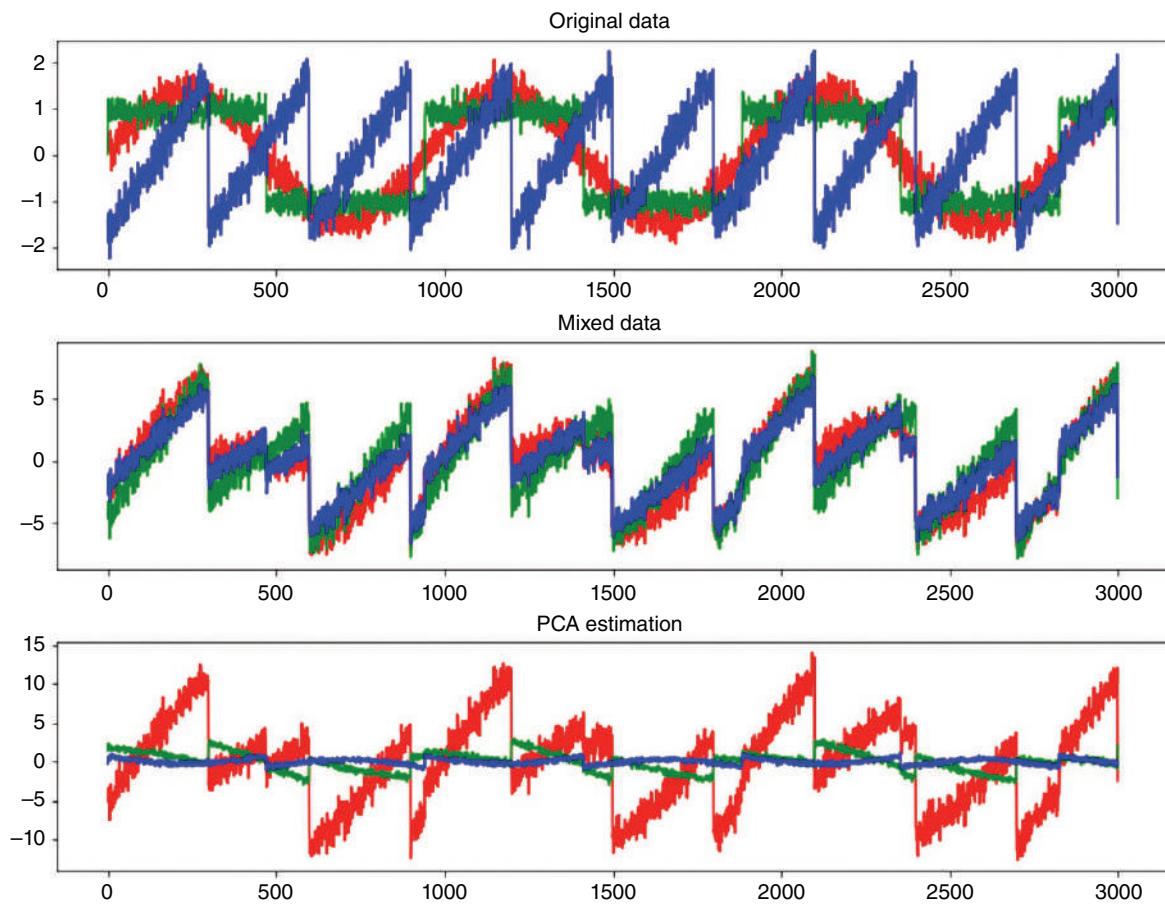
plt.figure(figsize=(10, 10))

models = [S, X, P]
names = ['Original Data',
         'Mixed Data',
         'PCA estimation']
colors = ['red', 'green', 'blue']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

plt.tight_layout()
```

Output:



As we can see, PCA does not perform well in isolating signals because signals are non-Gaussian processes. If the data only reflects Gaussian processes, ICA and PCA are equivalent. Let us see now how ICA behaves. One of the simplest ways to code ICA is to use FastICA from sklearn.decomposition.

Input:

```
from sklearn.decomposition import FastICA

# compute ICA
ica = FastICA(n_components=3)
I = ica.fit_transform(X) # Get the estimated sources

# compute PCA
pca = PCA(n_components=3)
P = pca.fit_transform(X) # estimate PCA sources

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

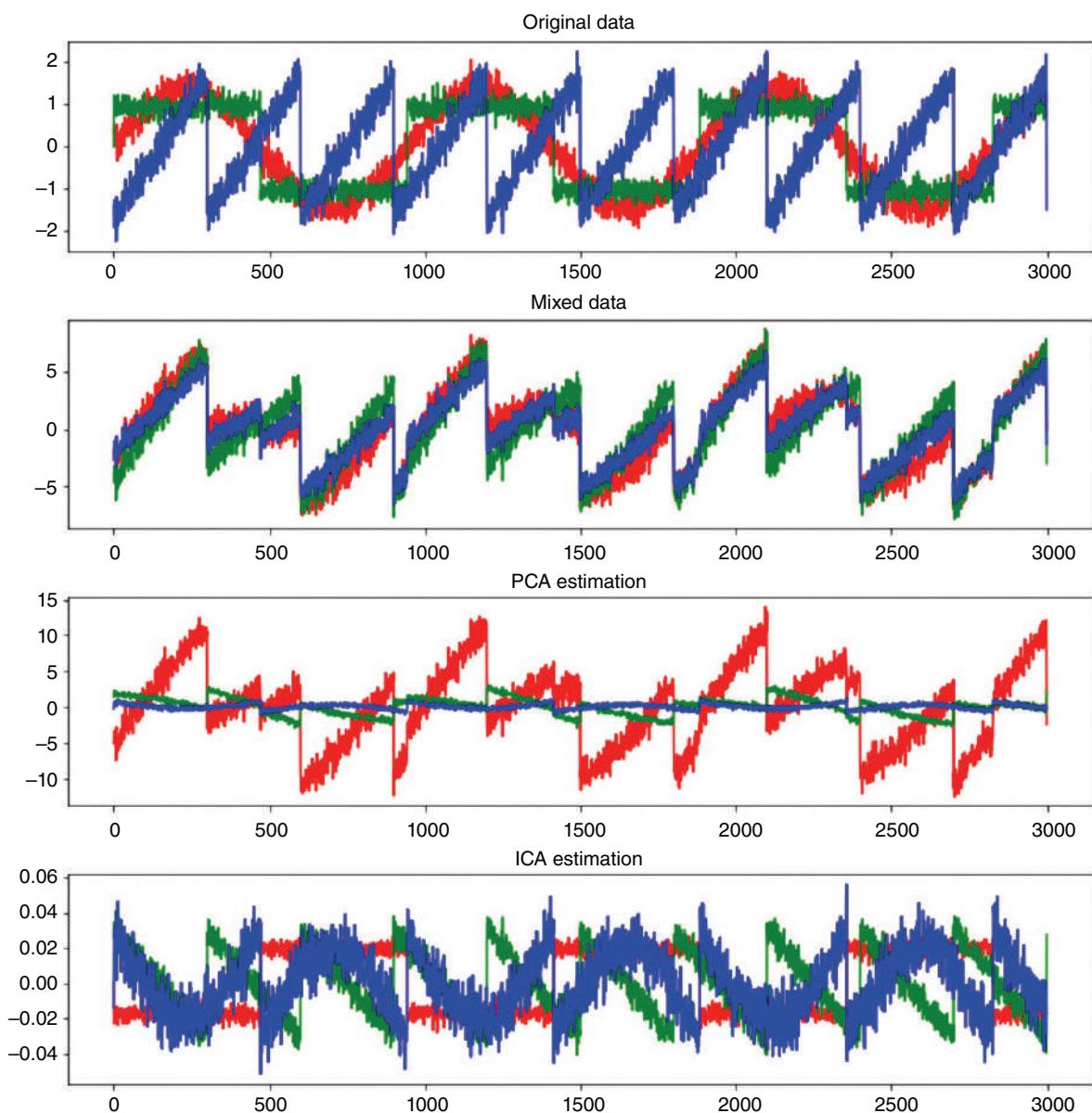
models = [S, X, P, I]
names = ['Original Data',
         'Mixed Data',
         'PCA estimation',
```

```
'ICA estimation']
colors = ['red', 'green', 'blue']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

plt.tight_layout()
```

Output:



As we can see, the signals are easier to isolate. We can also apply ICA after applying PCA to the data.

Input:

```

import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from sklearn import preprocessing
from sklearn.decomposition import FastICA, PCA

np.random.seed(0) # set seed for reproducible results
n_samples = 3000
time = np.linspace(0, 10, n_samples)

# Sinusoidal signal
s1 = np.sin(2 * time)
# Square signal
s2 = np.sign(np.sin(2 * time))
# Sawtooth signal
s3 = signal.sawtooth(2 * np.pi * time)

S = np.c_[s1, s2, s3]
S += 0.15 * np.random.normal(size=S.shape) # Add noise

# Standardize the data
S = StandardScaler().fit_transform(S)

# Let's Mix data with some matrix
M = np.array([[2, 1, 2], [1, 0.5, 3], [0.5, 1.5, 2]])
# Generating mixed observations
X = np.dot(S, M.T)

# compute PCA
pca = PCA(n_components=3)
P = pca.fit_transform(X) # estimate PCA sources

# compute ICA on PCA data
ica = FastICA(n_components=3)
I = ica.fit_transform(P)

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

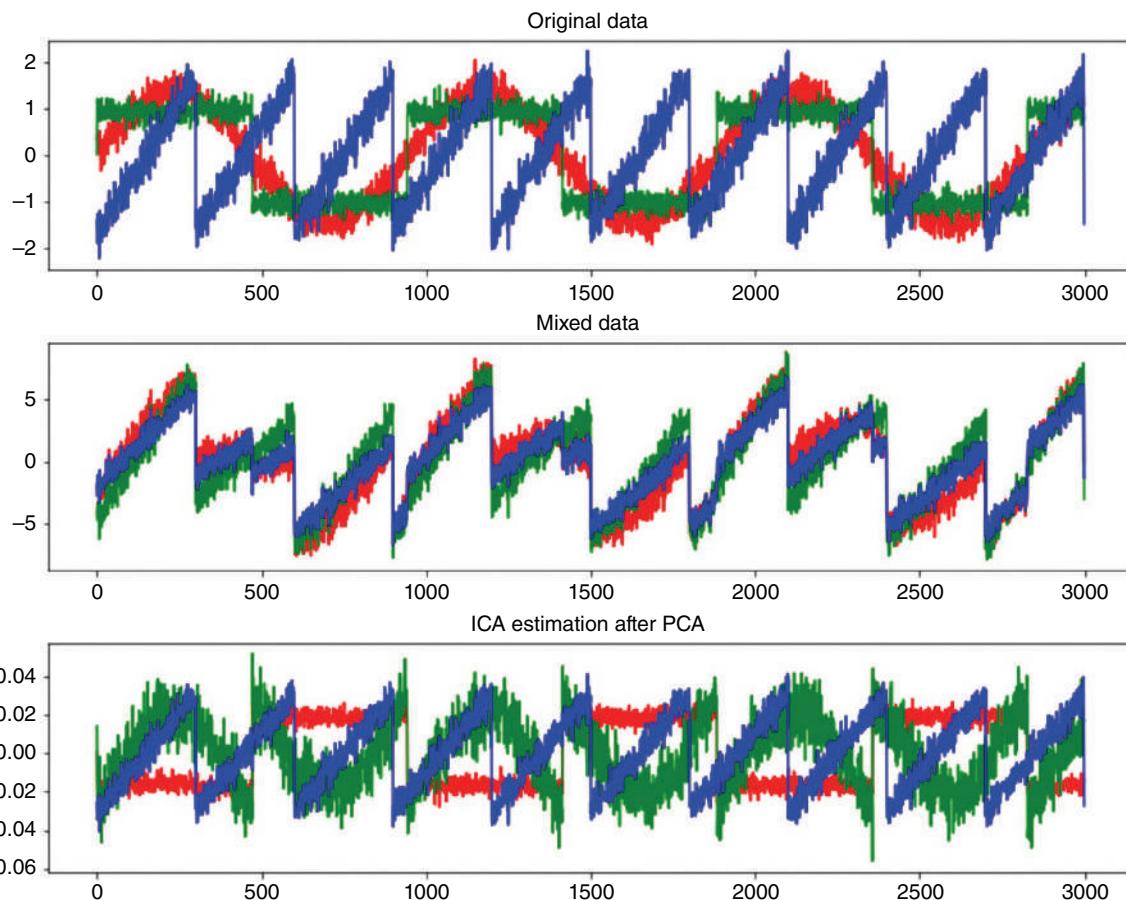
models = [S, X, I]
names = ['Original Data',
         'Mixed Data',
         'ICA estimation after PCA']
colors = ['red', 'green', 'blue']

for ii, (model, name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, ii)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color=color)

plt.tight_layout()

```

Output:



If we analyze the mushroom data by applying ICA with two components, we can see the difference from PCA.

Input:

```
import pandas as pd
csv_data = './data/mushrooms.csv'
df = pd.read_csv(csv_data, delimiter=',')

from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Encode the data
enc = LabelEncoder()
df = df.apply(enc.fit_transform)

# Divide the data, y the variable to classify and X the features
y = df.loc[:, df.columns == 'class'].values.ravel()
X = df.loc[:, df.columns != 'class']

# Standardize the data
X = StandardScaler().fit_transform(X)

from sklearn.decomposition import FastICA
# compute ICA on PCA data
```

```

ica = FastICA(n_components=2)
independent_components = ica.fit_transform(X)
ICA_df = pd.DataFrame(data = independent_components, columns = ['ICA 1', 'ICA 2'])

# Concatenation of ICA data and Target
df_class = pd.DataFrame(y, columns = ["Target"])
ICA_df = pd.concat([ICA_df, df_class], axis = 1)

import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

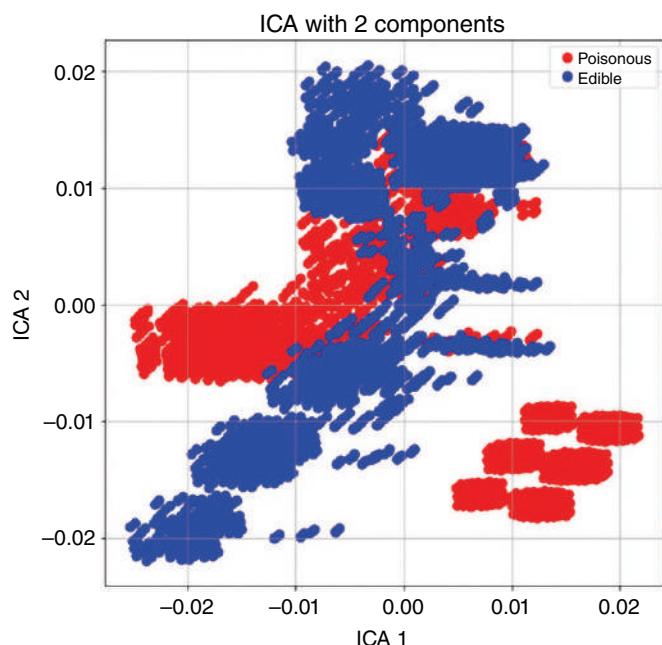
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('ICA 1', fontsize = 15)
ax.set_ylabel('ICA 2', fontsize = 15)
ax.set_title('ICA with 2 components', fontsize = 20)

targets = [1, 0]
colors = ['r', 'b']

for target, color in zip(targets,colors):
    indicesToKeep = ICA_df['Target'] == target
    ax.scatter(ICAgf.loc[indicesToKeep, 'ICA 1']
               , ICA_df.loc[indicesToKeep, 'ICA 2']
               , c = color
               , s = 50)
ax.legend(['Poisonous', 'Edible'])
ax.grid()

```

Output:



2.5.1.3 Linear Discriminant Analysis

LDA is a supervised machine learning technique used to classify data. It is also used as a dimensionality reduction technique to project the features from a higher dimensional space into a lower dimensional space with good class separability, avoiding overfitting and reducing computational costs. PCA and LDA are comparable in the sense that they are linear transformations. PCA is an unsupervised algorithm that can find the principal components, the directions, that maximize the variance in the dataset; it ignores class labels. LDA is not an unsupervised algorithm, as it considers class labels and computes the linear discriminants, the directions, that maximize the separation between multiple classes. In other words, LDA maximizes the distance between the means of each class and minimizes the spreading within the class itself (minimizing variation between each category). Thanks to the reduced overlap between the different classes, it can result in better accuracy when classifying data. Even though LDA considers class labels, that does not automatically mean that it performs better than PCA when used as a dimensionality reduction technique for a multi-class classification task for which class labels are known. As LDA is also a classifier, it is not uncommon to apply PCA for dimensionality reduction followed by LDA. A general conclusion when we compare LDA and PCA is that PCA can outperform LDA when the training dataset is small and that PCA is less sensitive to different training datasets. We can conclude that when we use LDA, we assume that the distribution of our data is Gaussian, features are statistically independent, and there are identical covariance matrices for every class. Ignoring these factors could lead to poor classification results. This especially applies to LDA as a classifier. When we use LDA for dimensionality reduction, LDA works reasonably well. Some papers have also provided concrete examples, for example, in object recognition, in which LDA has performed well as a classifier without respecting assumptions. In addition, if we use feature scaling such as standardization, it will not change the overall results of an LDA; it will only change the component axes.

Let us describe some steps of the LDA algorithm. In the literature, we can find different methods addressing class-dependent and class-independent LDA methods. In fact, class-independent LDA is traditionally associated with what is known as classic LDA; the class-dependent version of LDA more often goes by the name of quadratic discriminant analysis (QDA).

Let us consider a set of N samples $[x_i]$ with i from 1 to N and $X(N \times M)$, given by the following:

$$X = \begin{bmatrix} x_{(1,1)} & \cdots & x_{(1,M)} \\ \vdots & \ddots & \vdots \\ x_{(N,1)} & \cdots & x_{(N,M)} \end{bmatrix}$$

The first step is to compute the mean of each class $\mu_i(1 \times M)$, then compute the total mean of all data $\mu(1 \times M)$, and then calculate the between-class matrix $S_B(M \times M)$ as follows:

$$S_B = \sum_{i=1}^c n_i (\mu_i - \mu)(\mu_i - \mu)^T$$

We will need to compute the within-class matrix $S_W(M \times M)$:

$$S_W = \sum_{j=1}^c \sum_{i=1}^{n_j} (x_{ij} - \mu_j)(x_{ij} - \mu_j)^T$$

where x_{ij} represents the i th sample in the j th class.

After calculating S_B and S_W , the transformation matrix W of the LDA technique can be calculated using Fisher's formula:

$$\arg \max_W \frac{W^T S_B W}{W^T S_W W}$$

It can be reformulated as follows:

$$S_W W = \lambda S_B W$$

where λ represents the eigenvalues of W . The solution can be resolved by calculating the eigenvalues and eigenvectors of the following expression:

$$W = S_W^{-1} S_B$$

if S_W is non-singular.

We can then sort the eigenvectors with the k highest eigenvalues that will be used to construct a lower dimensional space (V_k). The other eigenvectors (V_{k+1}, \dots, V_M) are neglected.

Each sample, x_i in the M -dimensional space can be represented in a k -dimensional space by projecting them onto the lower dimensional space of LDA as follows: $y_i = x_i V_k$.

Let us apply it to a concrete example with the well-known “Iris” dataset deposited on the UCI machine learning repository (<https://archive.ics.uci.edu/ml/datasets/iris>).

The dataset contains three classes of 50 instances each and four features (sepal length, sepal width, petal length, petal width). Each class refers to a type of iris plant (setosa, versicolour, virginica).

Input:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import datasets

# Load Iris data and decompose it (X, y)
iris = datasets.load_iris()
col_names = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']
X = pd.DataFrame(data=iris.data, columns = col_names)
y = pd.DataFrame(data=iris.target, columns = ['class'])
target_names = iris.target_names
display(X)
display(y)
```

Output:

	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	class
0	5.1	3.5	1.4	0.2	0 0
1	4.9	3.0	1.4	0.2	1 0
2	4.7	3.2	1.3	0.2	2 0
3	4.6	3.1	1.5	0.2	3 0
4	5.0	3.6	1.4	0.2	4 0
...
145	6.7	3.0	5.2	2.3	145 2
146	6.3	2.5	5.0	1.9	146 2
147	6.5	3.0	5.2	2.0	147 2
148	6.2	3.4	5.4	2.3	148 2
149	5.9	3.0	5.1	1.8	149 2

150 rows × 4 columns

150 rows × 1 columns

The “class” column represents y , with 0 being setosa, 1 being versicolor, and 2 being virginica (we can print `target_names`). If we translate the DataFrame above into matrices, we can write the following:

$$X = \begin{bmatrix} x_{1\text{sepal length}} & x_{1\text{sepal width}} & x_{1\text{petal length}} & x_{1\text{petal width}} \\ x_{2\text{sepal length}} & x_{2\text{sepal width}} & x_{2\text{petal length}} & x_{2\text{petal width}} \\ \vdots & \vdots & \vdots & \vdots \\ x_{150\text{sepal length}} & x_{150\text{sepal width}} & x_{150\text{petal length}} & x_{150\text{petal width}} \end{bmatrix},$$

$$y = \begin{bmatrix} w_{setosa} \\ w_{setosa} \\ \vdots \\ w_{virginica} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 2 \end{bmatrix}$$

If we follow the steps above, we can first start to compute the mean vectors m_i of the three classes (setosa, versicolor, virginica):

$$m_0 = \begin{bmatrix} \mu_{w_0(\text{sepal length})} \\ \mu_{w_0(\text{sepal width})} \\ \mu_{w_0(\text{petal length})} \\ \mu_{w_0(\text{petal width})} \end{bmatrix}, m_1 = \begin{bmatrix} \mu_{w_1(\text{sepal length})} \\ \mu_{w_1(\text{sepal width})} \\ \mu_{w_1(\text{petal length})} \\ \mu_{w_1(\text{petal width})} \end{bmatrix}, m_2 = \begin{bmatrix} \mu_{w_2(\text{sepal length})} \\ \mu_{w_2(\text{sepal width})} \\ \mu_{w_2(\text{petal length})} \\ \mu_{w_2(\text{petal width})} \end{bmatrix}$$

The within-class S_w is calculated as follows:

$$S_w = \sum_{i=1}^c S_i$$

where

$$S_i = \sum_{x \in D_i}^n (x - m_i)(x - m_i)^T$$

and

$$m_i = \frac{1}{n_i} \sum_{x \in D_i}^n x_k$$

The between-class matrix S_B is computed as follows:

$$S_B = \sum_{i=1}^c N_i (m_i - m)(m_i - m)^T$$

where m is the overall mean, N_i is the sample size of the respective classes, and m_i is the sample mean of the respective classes.

As described above, we will need to solve the generalized eigenvalue problem for the matrix $W = S_w^{-1}S_B$. We can use NumPy or LAPACK for this purpose. Next, we will sort the eigenvectors by decreasing eigenvalues and choose the k eigenvectors with the largest eigenvalues for the new feature subspace. We could do all these steps manually and can find many examples on the internet if needed, but the simplest way is to use libraries that automatically perform the process for us such as scikit-learn:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=2)
X_r2 = lda.fit(X, y).transform(X)
```

To compare with PCA, let us plot both PCA and LDA transformations.

Input:

```

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Loading the data from Iris dataset
iris = datasets.load_iris()

# Splitting the data into X and y
X = iris.data
y = iris.target

from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Apply PCA with two components
pca = PCA(n_components=2)
X_pca = pca.fit(X).transform(X)

# Apply LDA with two components
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit(X, y).transform(X)

plt.figure()
colors = ["darkblue", "darkviolet", "darkturquoise"]
linewidth = 2

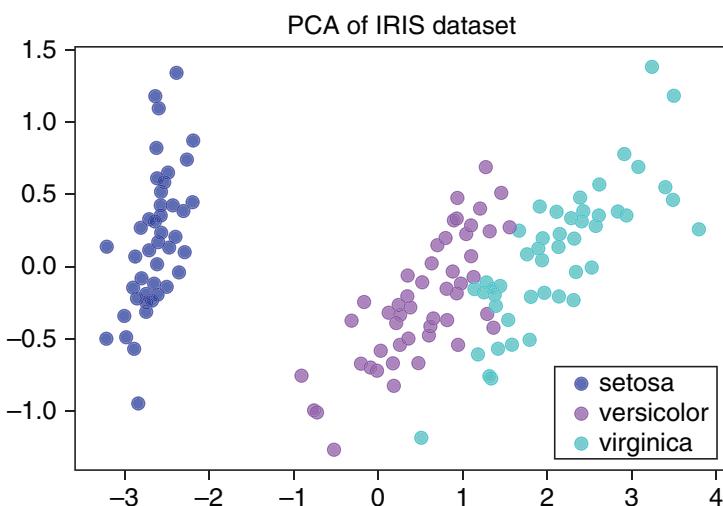
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(
        X_pca[y == i, 0], X_pca[y == i, 1], color=color, alpha=0.8, lw=linewidth,
        label=target_name
    )
plt.legend(loc="best", shadow=False, scatterpoints=1)
plt.title("PCA of IRIS dataset")

plt.figure()
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(
        X_lda[y == i, 0], X_lda[y == i, 1], alpha=0.8, color=color, label=target_name
    )
plt.legend(loc="best", shadow=False, scatterpoints=1)
plt.title("LDA of IRIS dataset")

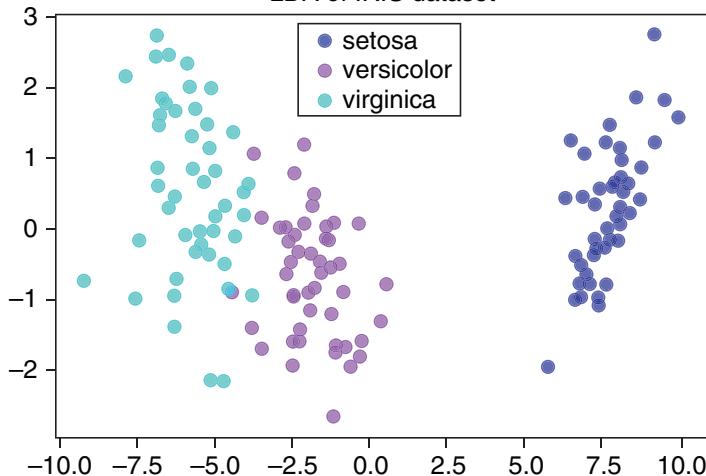
plt.show()

```

Output:



LDA of IRIS dataset



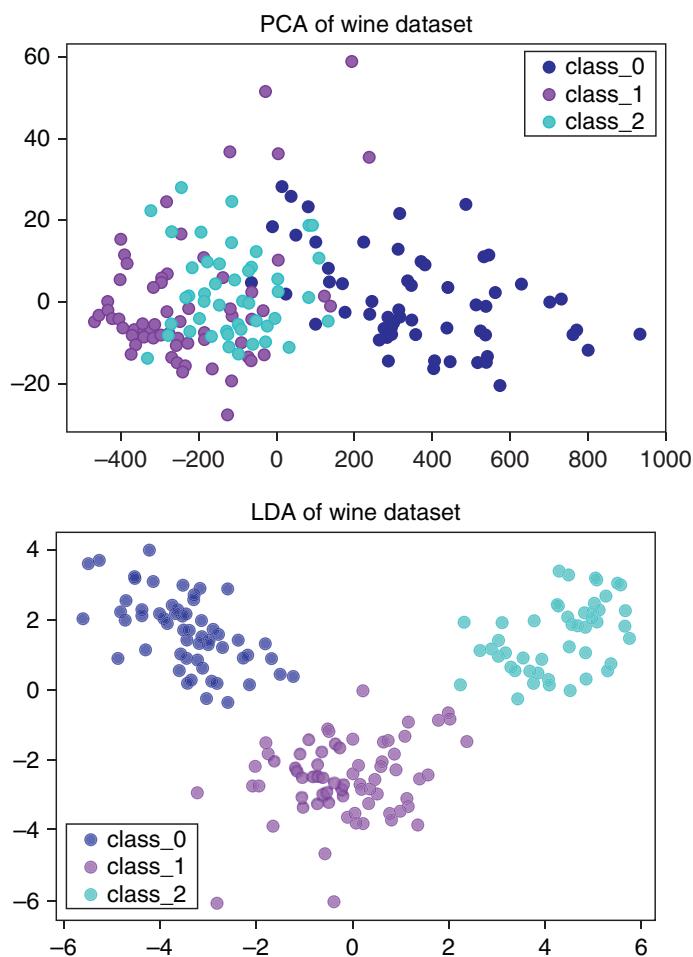
As we can see, both approaches can clearly separate the data. If we apply the same approach to another classic dataset (the wine dataset), we will see that LDA has a serious advantage. Let us only load the wine data, split it into X and y , and apply the same code as above.

Input:

```
from sklearn.datasets import load_wine
wine = load_wine()

# Splitting the data into X and y
X = wine.data
y = wine.target
```

Output:



2.5.1.4 Locally Linear Embedding

Before discussing the LLE method, we need to understand the nonlinear dimensionality reduction approach. As we have seen, we can use supervised and unsupervised linear dimensionality reduction techniques such as PCA, ICA, or LDA to perform linear data projection. The drawback of these methods is that they can miss important nonlinear structure in the data. In nonlinear dimensionality reduction, the goal is to retrieve a low-dimensional representation of a manifold (see below) that is embedded in a high-dimensional observation space. The linear techniques we have discussed mostly adopt concepts from linear algebra to perform dimensionality reduction. But sometimes, in fact very often, real-world applications generate nonlinear data. The difference between linear and nonlinear dimensionality reduction is that in the first case we transform the data to a low-dimension space as a linear combination of the original variables and in the second case we achieve lower dimensional representation of the data while preserving the original distances between the data points.

LLE is an unsupervised learning algorithm that computes low-dimensional, neighborhood-preserving embeddings of high-dimensional inputs. LLE is based on manifold learning, which is a class of unsupervised estimators designed to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces. We can consider a sheet of paper. The sheet of paper is a two-dimensional object that exists in our three-dimensional environment. In other words, the sheet of paper is a two-dimensional manifold embedded in three-dimensional space. If we perform a linear operation such as rotating, reorienting or stretching the paper in a three-dimensional space, we do not concretely change the flat geometry of the sheet of paper. However, if we crumple, bend, or curl the sheet of paper, we still have a two-dimensional manifold but its embedding into the three-dimensional space is no longer linear. Isomap, LLE, Hessian LLE, L-isomap, and manifold sculpting are some examples of manifold learning algorithms and are shown in Figure 2.12.

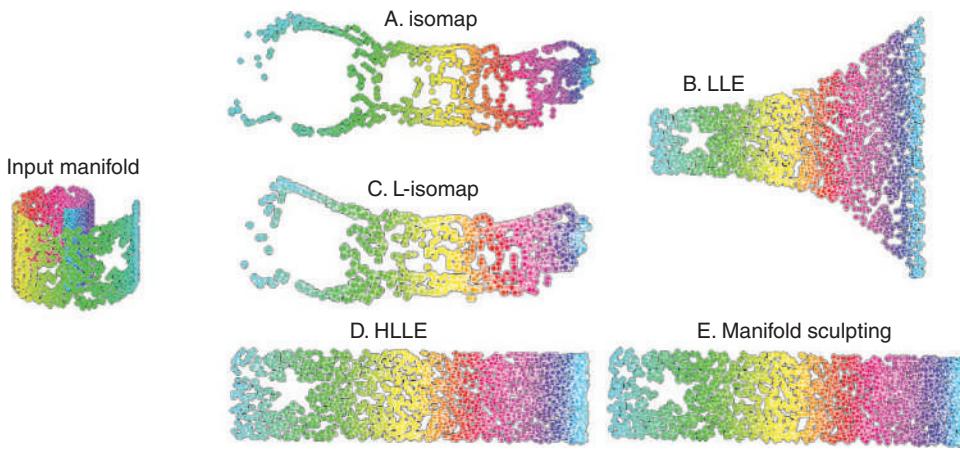


Figure 2.12 Comparison of several manifold learners on a Swiss Roll manifold. Color is used to indicate how points in the results correspond to points on the manifold. The Swiss Roll manifold is a typical example in which we are provided as input some data in a three-dimensional space with a distribution resembling one of a roll that we can unroll to reduce our data into two-dimensional space. Source: Gashler et al. (2008).

LLE leverages the local symmetries of linear reconstruction to discover the structure of nonlinear manifolds in high-dimensional data. To compute LLE and map high-dimensional data points, \vec{X}_i , to low-dimensional embedding vectors, \vec{Y}_i , we need to follow a few steps. First, we need to compute the neighbors of each data point \vec{X}_i and compute the weights W_{ij} that best reconstruct each data point \vec{X}_i from its neighbors, minimizing the cost of the following equation by constrained linear fits:

$$\epsilon(W) = \sum_i \left| \vec{X}_i - \sum_j W_{ij} \vec{X}_j \right|^2$$

where W_{ij} is the weight that describes the contribution of the j th data point to the i th reconstruction. In the simplest formulation, we identify K -nearest neighbors per data point, as measured by Euclidean distance. We can also use more sophisticated methodologies such as selecting points with a ball of fixed radius to select neighbors. The equation above sums the squared distances between all the data points and their reconstruction. To compute W_{ij} , we minimize the cost function using two constraints:

- Each data point \vec{X}_i is reconstructed only from its neighbors, meaning that if \vec{X}_j does not belong to this set, $W_{ij} = 0$.
- $\sum_j W_{ij} = 1$ (the row of the weight matrix sum).

The last step is to compute the vectors \vec{Y}_i best reconstructed by the weights W_{ij} , minimizing the quadratic form by its bottom nonzero eigenvectors in the following equation:

$$\phi(Y) = \sum_i \left| \vec{Y}_i - \sum_j W_{ij} \vec{Y}_j \right|^2$$

where the low-dimensional vector \vec{Y}_i represents the global internal coordinates on the manifold. The equation can be minimized by solving a sparse $N \times N$ eigenvector problem, whose bottom d (we minimize the embedding cost function by choosing d -dimensional coordinates of \vec{Y}_i) nonzero eigenvectors provide an ordered set of orthogonal coordinates centered on the origin.

To apply LLE and its variants, we will use the *digits* dataset, in which each data point is an 8×8 image of a numerical digit (1797 samples, 64 features, 180 samples per class). Let us first represent an extract of the *digits* dataset.

Input:

```
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
import numpy as np

digits = load_digits()
X = digits.data
y = digits.target
n_samples = 1797
n_features = 64
n_neighbors = 30

# Plot an extract of the data
fig, axs = plt.subplots(nrows=20, ncols=20, figsize=(10, 10))
for idx, ax0 in enumerate(axs.ravel()):
    # cmap=plt.cm.binary to display data in black (digits) and white (background)
    ax0.imshow(X[idx].reshape((8, 8)), cmap=plt.cm.binary)
    ax0.axis("off")

extract = fig.suptitle("Digits Dataset Selection", fontsize=15)
```

Output:

Digits dataset selection

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	
8	4	1	7	7	3	5	1	0	0	2	2	7	8	2	0	1	2	6	3	
3	7	3	3	4	6	6	4	9	1	5	0	3	5	2	8	4	0	0	0	
1	7	6	3	2	1	7	4	6	3	1	3	3	1	7	6	8	4	3	4	
4	0	5	3	6	3	6	1	7	5	4	5	7	2	8	2	2	5	7	9	
5	4	8	2	4	9	0	1	9	8	0	1	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	
0	3	5	5	6	5	5	0	3	8	3	2	4	1	7	7	3	5	1	0	0
2	2	7	8	1	0	4	2	6	3	3	7	3	3	4	6	6	6	4	9	
1	5	0	9	5	2	8	2	0	0	1	7	6	3	2	1	7	3	1	3	
9	4	7	6	8	4	3	1	4	0	5	7	6	9	6	4	7	5	4	4	
7	1	4	2	2	5	5	4	8	8	4	9	0	8	9	8	0	4	2	3	
4	5	6	7	8	9	0	4	2	3	4	5	6	7	8	9	0	4	2	3	
4	5	6	7	9	8	9	0	9	5	5	6	5	0	9	8	9	8	4	1	7
7	3	5	4	0	0	2	2	7	8	2	0	1	2	6	3	3	7	3	3	
4	6	6	6	4	9	1	5	0	9	5	2	8	2	0	0	4	4	6	3	
2	4	2	4	6	3	4	3	9	1	7	6	8	4	3	1	4	0	5	3	
6	9	6	1	7	5	4	4	7	2	8	2	2	5	7	9	5	4	8	8	
4	9	0	8	9	3	0	1	2	3	4	5	6	7	8	9	0	1	2	3	

We then define a *plot_embedding_function* to plot LLE, LLE variants, or other embedding techniques we will see in this chapter. This function will allow us to display and control whether the digits are grouped together or scattered in the embedding space.

Input:

```
# We define a plot_embedding_function in order to plot the LLE, future variants and
# other embedding techniques
# Display and control if digits are grouped together in the embedding space

import numpy as np
from matplotlib import offsetbox
from sklearn.preprocessing import MinMaxScaler

def plot_embedding_function(X, title, ax):
    # Rescaling Data
    X = MinMaxScaler().fit_transform(X)
    for digit in digits.target_names:
        ax.scatter(
            *X[y == digit].T,
            marker=f"${digit}$",
            s=60,
            color=plt.cm.Dark2(digit),
            alpha=0.425,
            zorder=2,
        )
    shown_images = np.array([[1.0, 1.0]])
    for i in range(X.shape[0]):
        # plot every digit on the embedding space
        # show an annotation box for a group of digits
        dist = np.sum((X[i] - shown_images) ** 2, 1)
        if np.min(dist):
            # don't show points that are too close
            continue
        shown_images = np.concatenate([shown_images, [X[i]]], axis=0)
        imagebox = offsetbox.AnnotationBbox(
            offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r), X[i]
        )
        imagebox.set(zorder=1)
        ax.add_artist(imagebox)

    ax.set_title(title)
    ax.axis("off")
```

LLE can be implemented using scikit-learn (*sklearn.manifold*).

Input:

```
from sklearn.manifold import LocallyLinearEmbedding

embeddings = {
    "Standard LLE": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="standard"
    ),
}
```

Once we have declared the LLE method, we can perform the projection of the original data. In addition, in the example provided by scikit-learn (manifold learning on handwritten digits) to illustrate various embedding techniques on the digits dataset, the computational time to perform each projection can be stored.

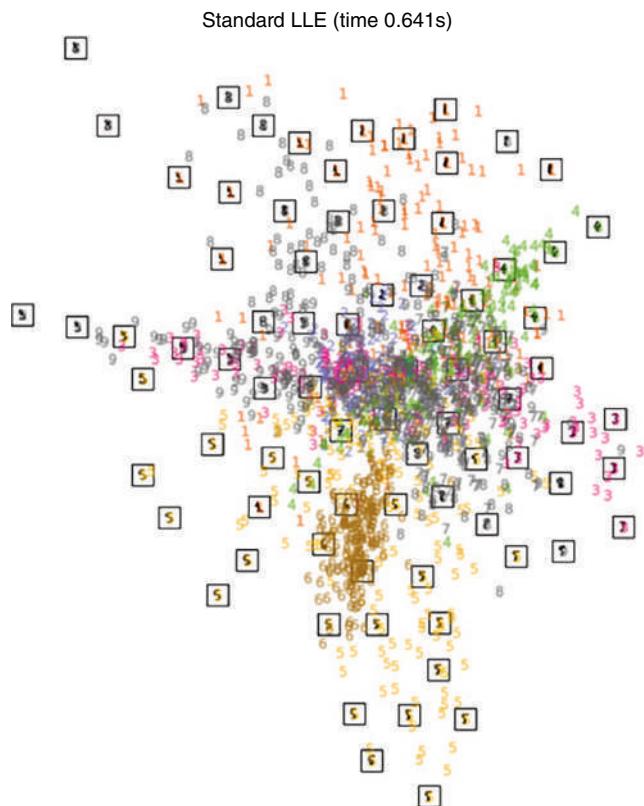
Input:

```
from time import time

projections, timing = {}, {}
for name, transformer in embeddings.items():
    if name.startswith("Standard LLE"):
        data = X.copy()
        data.flat[:: X.shape[1] + 1] += 0.01
    else:
        data = X

    print(f"{name}...")
    start_time = time()
    projections[name] = transformer.fit_transform(data, y)
    timing[name] = time() - start_time
```

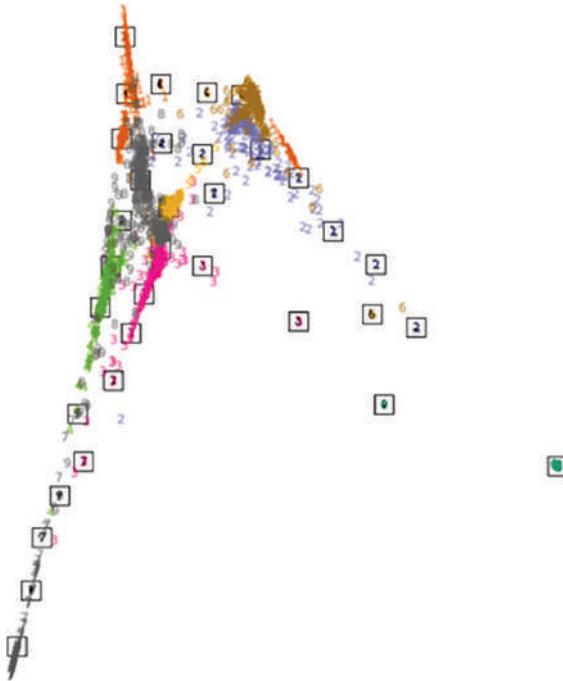
Output:



As we can see, data points are quite scattered; this process took 0.641 seconds on my personal computer. If we change the number of neighbors (`n_neighbors`), we can see important changes.

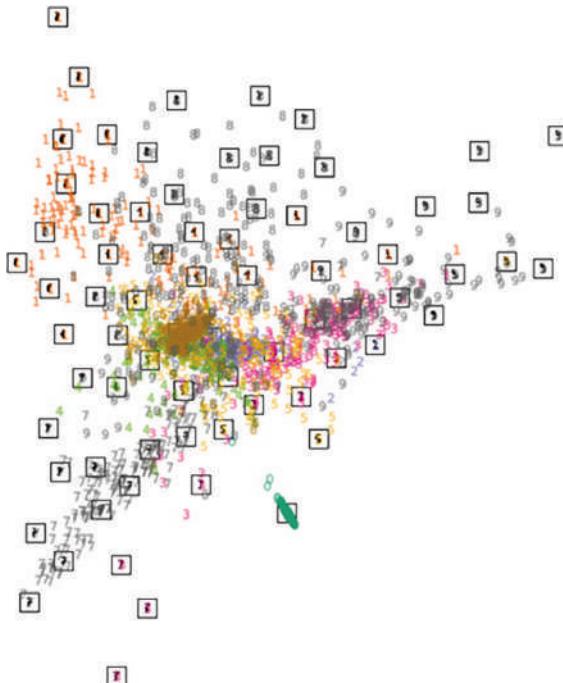
Output (n_neighbors = 10):

Standard LLE (time 0.363s)



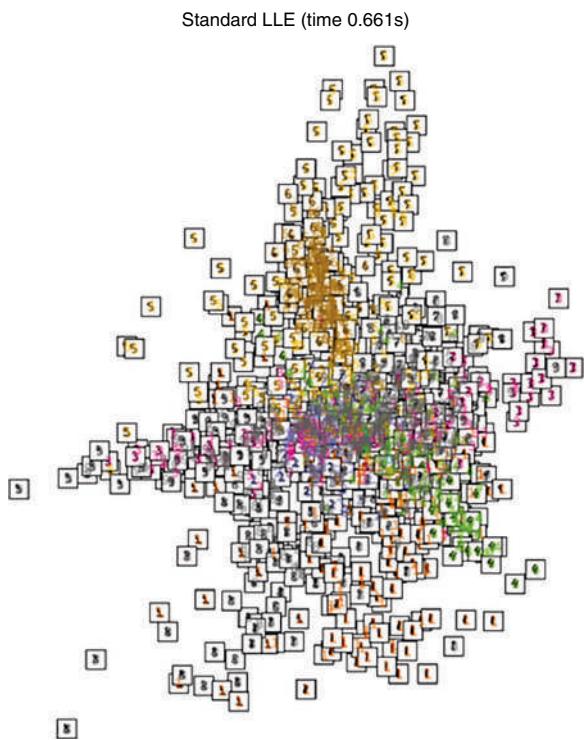
Output (n_neighbors = 20):

Standard LLE (time 0.432s)

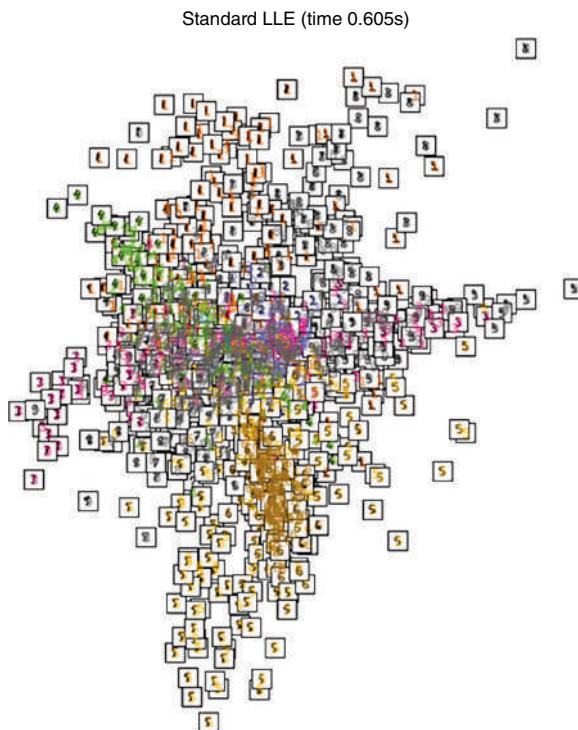


In the code above, we rescale the data using MinMaxScaler. If we return to n_neighbors = 30 and change the rescaling method, we also notice some changes.

Output (StandardScaler):



Output (RobustScaler):



In the literature, we can find variants of LLE algorithms such as Hessian locally linear embedding (HLLE), modified locally linear embedding (MLLE), or local tangent space alignment (LTSA). These variants can solve a well-known issue with LLE, which is the regularization problem. HLLE achieves linear embedding by minimizing the Hessian function on the manifold (a Hessian-based quadratic form at each neighborhood used to recover the locally linear structure). The scaling is not optimal with increased data size but tends to give higher quality results compared to standard LLE. MLLE addresses the regularization problem by using multiple weight vectors in each neighborhood. In LTSA, PCA is applied on the neighbors to construct a locally linear patch considered to be an approximation of the tangent space at the point. A coordinate

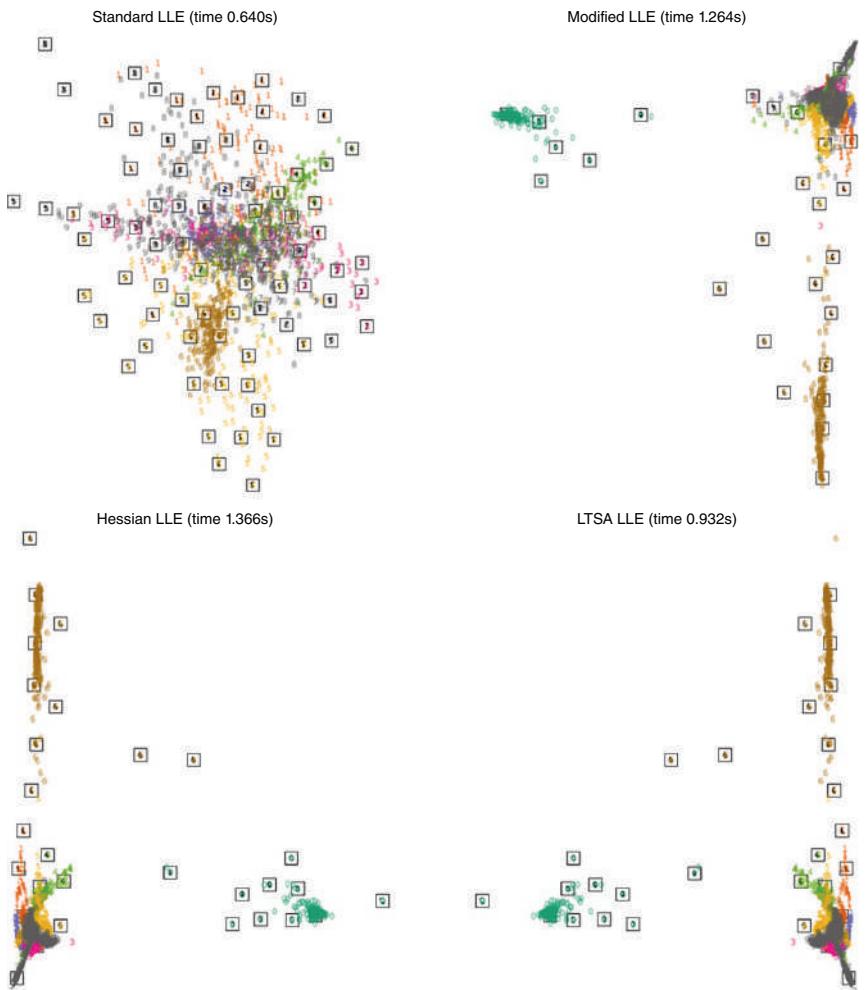
representation of the neighbors is provided by the tangent space and provides a low-dimensional representation of the patch.

Let us project the variants by modifying the code above.

Input:

```
embeddings = {
    "Standard LLE": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="standard"
    ),
    "Modified LLE": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="modified"
    ),
    "Hessian LLE": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="hessian"
    ),
    "LTSA LLE": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="ltsa"
    ),
}
```

Output:



The output demonstrates the differences among the variants. HLLE and MLLE provided better results in comparison to the standard LLE, although the standard LLE took less time to compute.

Algorithm 1: Simple version of t-Distributed Stochastic Neighbor Embedding.

Data: data set $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$,
cost function parameters: perplexity $Perp$,
optimization parameters: number of iterations T , learning rate η , momentum $\alpha(t)$.
Result: low-dimensional data representation $\mathcal{Y}^{(T)} = \{y_1, y_2, \dots, y_n\}$.

```

begin
  compute pairwise affinities  $p_{ji}$  with perplexity  $Perp$  (using Equation 1)
  set  $p_{ij} = \frac{p_{ji} + p_{ij}}{2n}$ 
  sample initial solution  $\mathcal{Y}^{(0)} = \{y_1, y_2, \dots, y_n\}$  from  $\mathcal{N}(0, 10^{-4})$ 
  for  $t = 1$  to  $T$  do
    compute low-dimensional affinities  $q_{ij}$  (using Equation 4)
    compute gradient  $\frac{\delta C}{\delta y_j}$  (using Equation 5)
    set  $\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta y_j} + \alpha(t)(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)})$ 
  end
end

```

Figure 2.13 A simple version of the t-SNE algorithm described by Laurens van der Maaten and Geoffrey Hinton, who introduced t-SNE. Source: van der Maaten and Hinton (2008).

2.5.1.5 The t-Distributed Stochastic Neighbor Embedding Technique

To visualize high-dimensional data, we can also use t-distributed stochastic neighbor embedding (t-SNE), which is an unsupervised and nonlinear technique that converts similarities between data points to joint probabilities. The Kullback–Leibler divergence is minimized between the joint probabilities of the low-dimensional embedding and the high-dimensional data. The t-SNE method models the dataset with a dimension-agnostic probability distribution to find a lower dimensional approximation with a closely matching distribution (Figure 2.13).

PCA attempts to preserve large pairwise distances to maximize variance, whereas t-SNE is concerned with preserving only small pairwise distances. If the number of features is high, it is recommended when applying t-SNE to use a dimensionality reduction method such as PCA to reduce computing time and suppress noise.

Let us apply PCA, LDA, and t-SNE on the digits dataset using scikit-learn. We have adapted the code used in the LDA section above to load the digits dataset and add t-SNE.

Input:

```

import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.manifold import TSNE

from sklearn.datasets import load_digits

digits = load_digits()
X = digits.data
y = digits.target
target_digits = digits.target_names

from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

```

```

# Apply PCA with two components
pca = PCA(n_components=2)
X_pca = pca.fit(X).transform(X)

# Apply LDA with two components
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit(X, y).transform(X)

# Apply t-SNE with two components
X_tsne = TSNE(n_components=2, learning_rate='auto', init='pca').fit_transform(X)

plt.figure(figsize=(8, 8))
colors = ["darkblue", "darkviolet", "darkturquoise", "black", "red", "pink",
"darkseagreen", "cyan", "grey", "darkorange"]
linewidth = 0.5

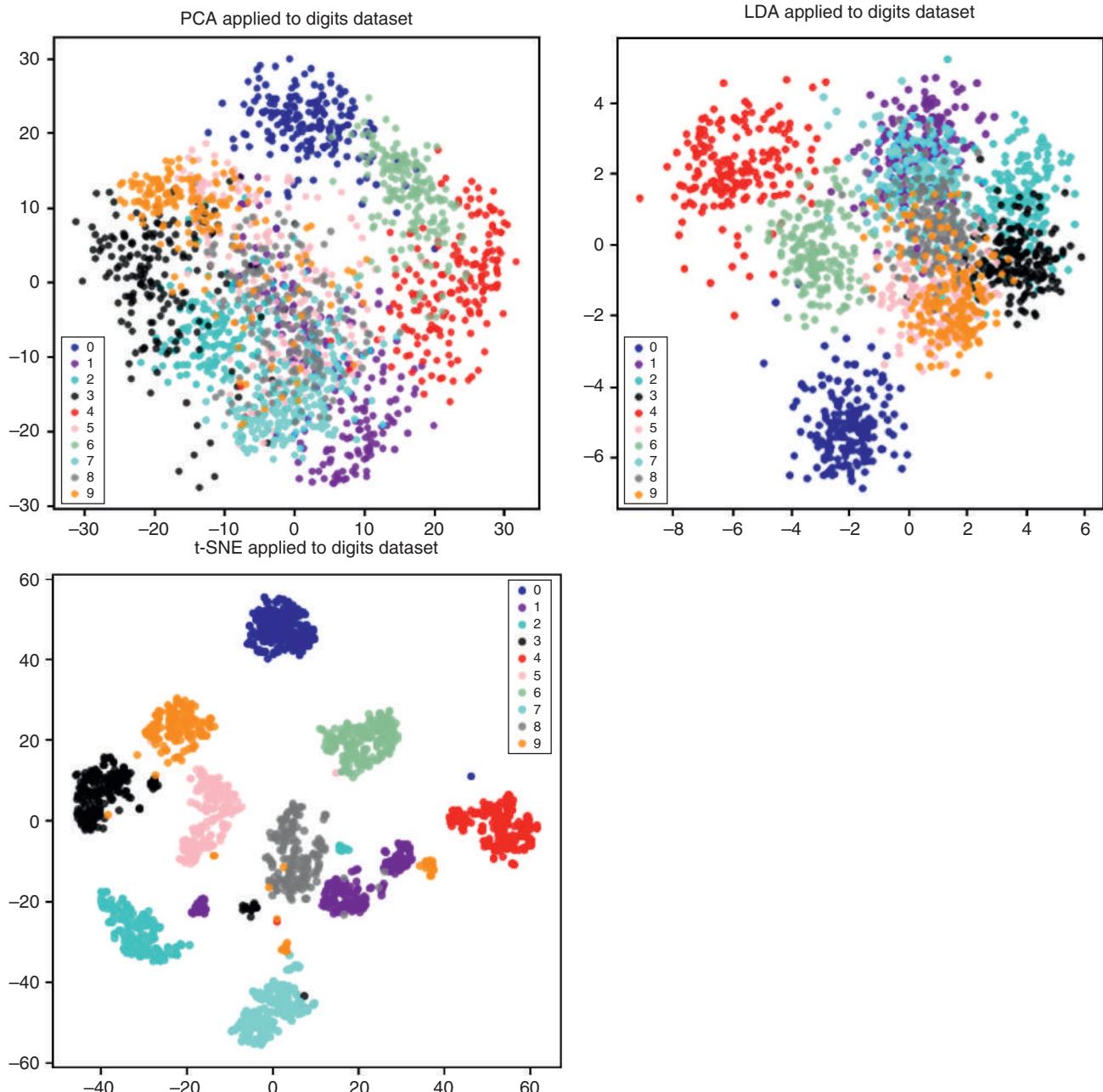
# Plot PCA data (X_pca)
for color, i, target_name in zip(colors, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
target_digits):
    plt.scatter(
        X_pca[y == i, 0], X_pca[y == i, 1], color=color, alpha=0.8, lw=linewidth,
label=target_name
    )
plt.legend(loc="best", shadow=False, scatterpoints=1)
plt.title("PCA applied to Digits Dataset")

# Plot LDA data (X_lda)
plt.figure(figsize=(8, 8))
for color, i, target_name in zip(colors, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
target_digits):
    plt.scatter(
        X_lda[y == i, 0], X_lda[y == i, 1], alpha=0.8, color=color, label=target_name
    )
plt.legend(loc="best", shadow=False, scatterpoints=1)
plt.title("LDA applied to Digits Dataset")

# Plot t-SNE data (X_tsne)
plt.figure(figsize=(8, 8))
for color, i, target_name in zip(colors, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
target_digits):
    plt.scatter(
        X_tsne[y == i, 0], X_tsne[y == i, 1], alpha=0.8, color=color, label=target_name
    )
plt.legend(loc="best", shadow=False, scatterpoints=1)
plt.title("t-SNE applied to Digits Dataset")
plt.show()

```

Output:



2.5.1.6 More Manifold Learning Techniques

If we read further about embedding techniques, we can find many more algorithms with many other variants such as truncated SVD, isomap, neighborhood component analysis, spectral embedding, multidimensional scaling, and others. We can examine some of these methods by modifying the code used in the LLE section and observing how the different algorithms behave with the digits dataset.

Input:

```

from sklearn.decomposition import TruncatedSVD
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomTreesEmbedding
from sklearn.manifold import (
    Isomap,
    LocallyLinearEmbedding,
    MDS,
    SpectralEmbedding,
    TSNE,
)
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from sklearn.pipeline import make_pipeline
from sklearn.random_projection import SparseRandomProjection
embeddings = {
    "Random projection embedding": SparseRandomProjection(
        n_components=2, random_state=42
    ),
    "Truncated SVD embedding": TruncatedSVD(n_components=2),
    "Linear Discriminant Analysis embedding": LinearDiscriminantAnalysis(
        n_components=2
    ),
    "Isomap embedding": Isomap(n_neighbors=n_neighbors, n_components=2),
    "Standard LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="standard"
    ),
    "Modified LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="modified"
    ),
    "Hessian LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="hessian"
    ),
    "LTSA LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="ltsa"
    ),
    "MDS embedding": MDS(n_components=2, n_init=1, max_iter=120, n_jobs=2),
    "Random Trees embedding": make_pipeline(
        RandomTreesEmbedding(n_estimators=200, max_depth=5, random_state=0),
        TruncatedSVD(n_components=2),
    ),
    "Spectral embedding": SpectralEmbedding(
        n_components=2, random_state=0, eigen_solver="arpack"
    ),
    "t-SNE embedding": TSNE(
        n_components=2,
        init="pca",
        learning_rate="auto",
        n_iter=500,
        n_iter_without_progress=150,
        n_jobs=2,
    )
}

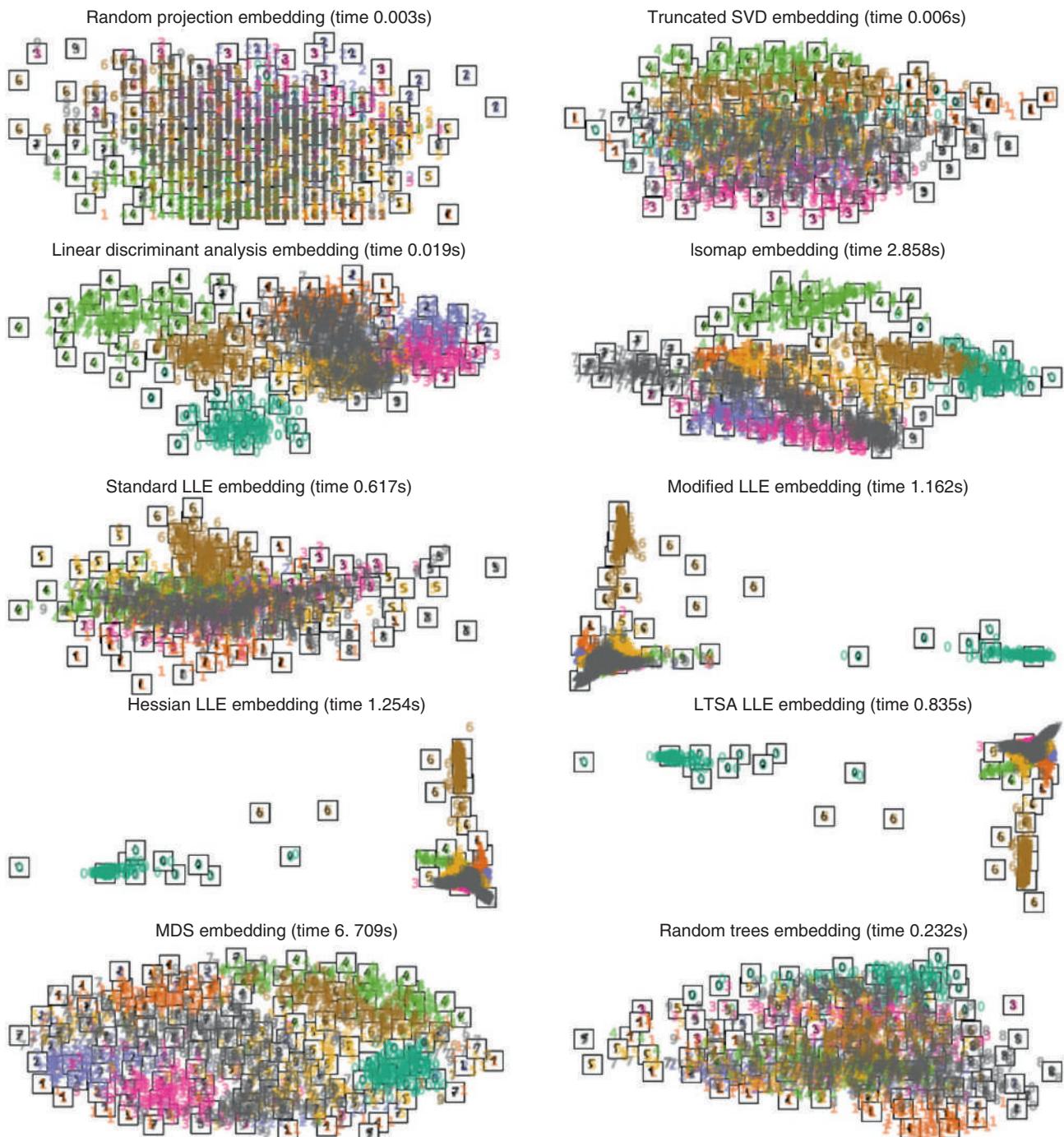
```

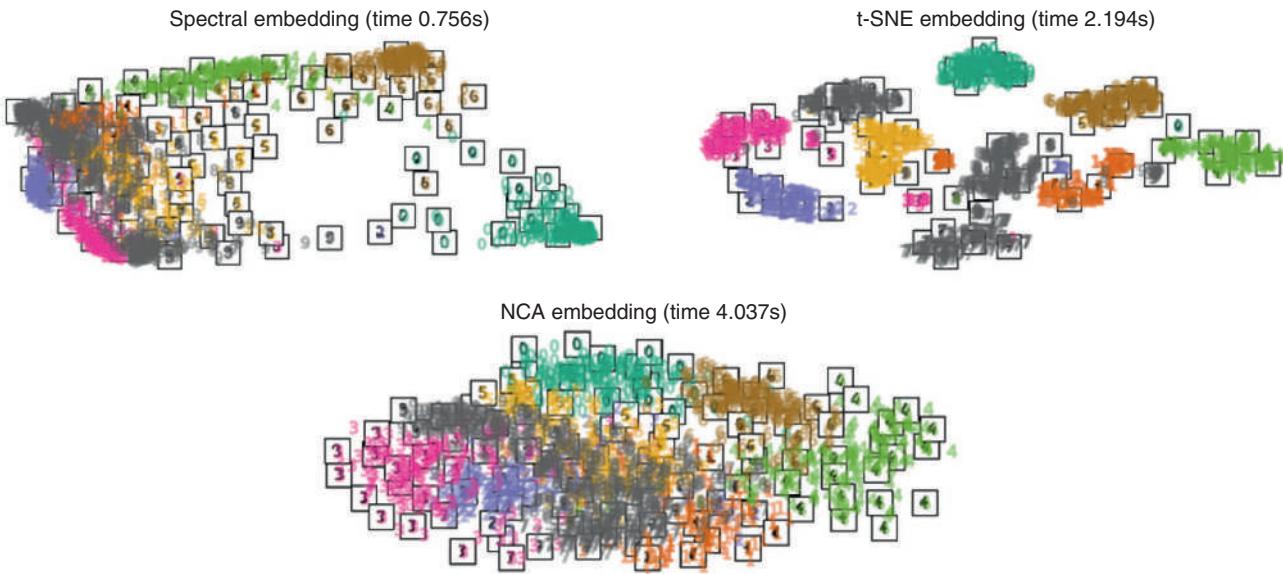
```

        random_state=0,
),
"NCA embedding": NeighborhoodComponentsAnalysis(
    n_components=2, init="pca", random_state=0
),
}
}

```

Output:





For practical reasons, we can also simply work with our variables without data visualization, as most of the time we are manipulating training and test data.

Input:

```
from sklearn import datasets
from sklearn.decomposition import PCA

from sklearn.decomposition import TruncatedSVD
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomTreesEmbedding
from sklearn.manifold import (
    Isomap,
    LocallyLinearEmbedding,
    MDS,
    SpectralEmbedding,
    TSNE,
)
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from sklearn.random_projection import SparseRandomProjection

from sklearn.datasets import load_wine
wine = load_wine()

# Splitting the data into X and y
X = wine.data
y = wine.target
target_wine_names = wine.target_names

# Apply PCA with two components
pca = PCA(n_components=2)
X_pca = pca.fit(X).transform(X)
```

```
# Apply LDA with two components
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit(X, y).transform(X)

# Apply Random projection with two components
ran = SparseRandomProjection(n_components=2, random_state=42)
X_ran = ran.fit_transform(X)

# Apply Truncated SVD with two components
trun = TruncatedSVD(n_components=2)
X_trun = trun.fit_transform(X)

# Apply Isomap with two components
isomap = Isomap(n_neighbors=n_neighbors, n_components=2)
X_isomap = isomap.fit_transform(X)

# Apply Standard LLE with two components
lle = LocallyLinearEmbedding(n_neighbors=n_neighbors, n_components=2,
method="standard")
X_lle = lle.fit_transform(X)

# Apply Modified LLE with two components
mllle = LocallyLinearEmbedding(n_neighbors=n_neighbors, n_components=2,
method="modified")
X_mllle = mllle.fit_transform(X)

# Apply Hessian LLE with two components
hlle = LocallyLinearEmbedding(n_neighbors=n_neighbors, n_components=2,
method="hessian")
X_hlle = hlle.fit_transform(X)

# Apply LTSA LLE with two components
ltsalle = LocallyLinearEmbedding(n_neighbors=n_neighbors, n_components=2,
method="ltsa")
X_ltsalle = ltsalle.fit_transform(X)

# Apply MDS with two components
mds = MDS(n_components=2, n_init=1, max_iter=120, n_jobs=2)
X_mds = mds.fit_transform(X)

# Apply Spectral with two components
spectral = SpectralEmbedding(n_components=2, random_state=0, eigen_solver="arpack")
X_spectral = spectral.fit_transform(X)

# Apply t-SNE with two components
tsne = TSNE(
    n_components=2,
    init="pca",
    learning_rate="auto",
    n_iter=500,
    n_iter_without_progress=150,
```

```

        n_jobs=2,
        random_state=0,
    )
X_tsne = tsne.fit_transform(X)

# Apply NCA with two components
nca = NeighborhoodComponentsAnalysis(n_components=2, init="pca", random_state=0)
X_nca = nca.fit(X,y).transform(X)

```

2.5.1.7 Feature Extraction with HephAlistos

HephAlistos also incorporates feature extraction techniques by including the features_extraction parameter:

- *features_extraction*: Selects the feature extraction method. The following options are available:
 - *pca*
 - *ica*
 - *icawithpca*
 - *lda_extraction*
 - *random_projection*
 - *truncatedSVD*
 - *isomap*
 - *standard_lle*
 - *modified_lle*
 - *hessian_lle*
 - *ltsa_lle*
 - *mds*
 - *spectral*
 - *tsne*
 - *nca*
- *number_components*: The number of principal components we want to keep for PCA, ICA, LDA, or other methods.
- *n_neighbors*: The number of neighbors to consider for manifold learning techniques.

If we go to the hephAlistos main folder and create a new Python file to create a new pipeline, we can copy and paste the following lines to test the routine:

```

from ml_pipeline_function import ml_pipeline_function

# Import Data
from data.datasets import neurons
df = neurons()

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', features_extraction = 'pca',
number_components = 2)

```

Here, we have decided to create a pipeline and to use row_removal to address missing data, to encode the “Target” feature with the label encoding method, to rescale the data using StandardScaler, and to use PCA with two components for feature extraction.

2.5.2 Feature Selection

There are many ways to increase the amount of data we have at our disposal and increase the number of features. We often tend to increase the dimensionality of our data, thinking that it will augment the performance of our models. Increasing dimensionality can have some negative consequences such as increasing complexity of the machine learning models, leading to overfitting, and increasing computational time while not always improving accuracy (due to misleading data), especially when we provide irrelevant features to linear algorithms such as linear and logistic regression. Overfitting will appear when we inject redundant data, adding noise. A concept similar to feature extraction, feature selection is a set of techniques to reduce the input variables to our machine learning models by only using relevant features that contribute most to the prediction variables. Feature selection is different from feature extraction in that feature selection returns a subset of the original features, whereas feature extraction creates new features from functions of the original features. Feature selection allows us to use the simplest models and generalize better by choosing a small subset of the relevant features from the original features by removing noisy, redundant, and irrelevant features.

We can use both supervised and unsupervised techniques and can classify feature selection methodologies as filter, wrapper, hybrid, or embedded. When we use the filter approach, it means that we simply select features based on statistical measures such as variance threshold, chi-squared test, Fisher score, correlation coefficient, or information gain. The idea is to filter out irrelevant features and redundant information. With filters, we take the intrinsic properties of the features via univariate statistics. In univariate tests, features are individually considered, meaning that a feature can be rejected if it is only informative when combined with another. The selected features are completely independent of the model we will use later. We can compute a score by features and simply select based on the score. In general, these methodologies require less computational time. The wrapper methodology uses a predictive model to evaluate combinations of features and assign model performance scores. In other words, wrapper methods evaluate using a specific machine learning algorithm to find optimal features. This means that, as opposed to filter methods, wrapper methods could need much more computation time if we are using high-dimensional data. In embedded-type feature selection, the selection of features is made within the model-learning process itself. In fact, the features are selected during the model-building process in each iteration of the model-training phase. If the number of observations is insufficient, there is a risk of overfitting when using wrapper methods (Figure 2.14).

Feature selection needs to be performed after splitting the data for training and testing, as there can be information leakage.

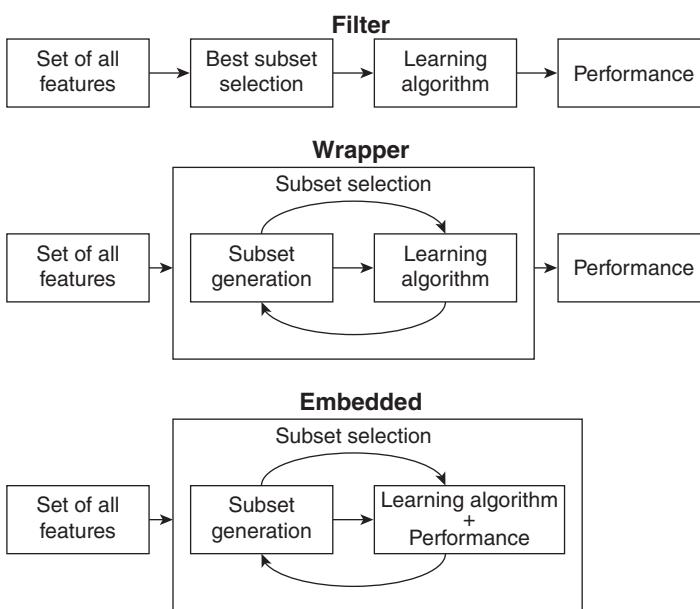


Figure 2.14 Flow of the different methods for feature selection.

All code examples provided to explain feature selection can be found in hephaistos/Notebooks/Features_selection.ipynb or https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Features_selection.ipynb. If we go to hephaistos/Notebooks in a terminal and open Jupyter Notebook (the command line is jupyter notebook), we can then open Feature_selection.ipynb in a browser and run the different code examples.

2.5.2.1 Filter Methods

Variance Threshold One simple idea to select features is to remove all features for which variance does not meet some threshold. For instance, if we calculate the variance of a feature and the result is zero, it means that the feature has the same value in all the samples. In that case, there is no need to take it into account. The hypothesis we form here is that a feature with higher variance contains more significant information. The drawback of this method is that we do not consider the correlation or any relationship between features. This filter method is called variance threshold. In this method, we can set a threshold, such as a percentage, to help identify the features to remove. Let us say that we have set the threshold at 80% and that we apply this threshold to a feature containing 0s and 1s; if we have more than 80% of 0s or more than 80% of 1s, the feature will be removed.

In the case of Boolean features (Bernoulli random variables), the variance is indicated by $\text{Var}(X) = p(1 - p)$, which means that the threshold (80%) is given by $0.8 \times (1 - 0.8)$.

To select features using variance threshold, we can simply employ the scikit-learn library *VarianceThreshold* from *sklearn.feature_selection*.

Input:

```
from sklearn.feature_selection import VarianceThreshold
# By default, it removes all zero-variance features
var = VarianceThreshold(threshold=0)
X_train = var.fit_transform(X_train)
print(X_train.shape)
```

Feature Selection Using Statistical Tests Statistical tests such as *z*-tests, *t*-tests, ANOVA, or correlation tests are commonly used to select features. Before applying statistical tests for feature selection, we need to understand some vocabulary associated with them. In statistics, hypothesis testing refers to the act of testing an assumption regarding a population feature by using sample data. In this case, we test the null hypothesis and the alternative hypothesis. In the null hypothesis, denoted H_0 , we hypothesize that there is no significant difference between sample and population or among different populations (for example, the mean of two samples is equal). In the alternative hypothesis, denoted H_1 , we hypothesize that there is a significant difference (for example, the mean of the two samples is not equal). In statistics, to decide whether we can reject the null hypothesis, we need to calculate a test statistic that provides a numerical value. There are two approaches: the critical value and the *p*-value. The critical value is a line on a curve splitting it into one or two sections that are the rejection regions. In other words, if the test statistic falls into one of the sections, we can reject the null hypothesis. On the contrary, if the test statistic does not fall into those regions (not extreme as the critical value), we cannot reject the null hypothesis. The critical value is calculated from a defined significance level α and the type of probability distribution of the ideal model.

In Figures 2.15–2.17, we can see a two-sided, a left-tailed, and a right-tailed test, respectively. In the examples shown in the figures, the idealized model is a normal probability distribution.

$$H_0 : \mu = \mu_0; H_1 : \mu \neq \mu_0$$

$$H_0 : \mu = \mu_0; H_1 : \mu < \mu_0$$

$$H_0 : \mu = \mu_0; H_1 : \mu > \mu_0$$

The probability value (*p*-value) of the test statistic is compared to the defined significance level (α). Smaller *p*-values provide stronger evidence to reject the null hypothesis ($p \leq 0.05$ is considered strong evidence). If the *p*-value is less than or equal to α , the null hypothesis H_0 is rejected. If the *p*-value is greater than α , H_0 is not rejected.

Another term to keep in mind is degrees of freedom, which is the number of independent variables and is used to calculate the *t*-statistic and the chi-squared statistic.

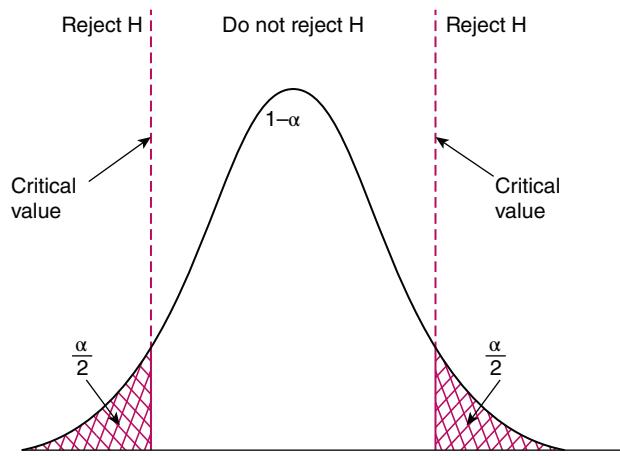


Figure 2.15 A two-sided test with a normal probability distribution.

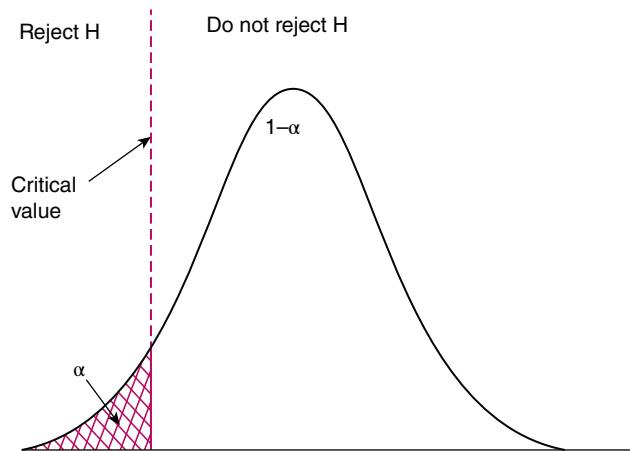


Figure 2.16 A left-tailed test with a normal probability distribution.

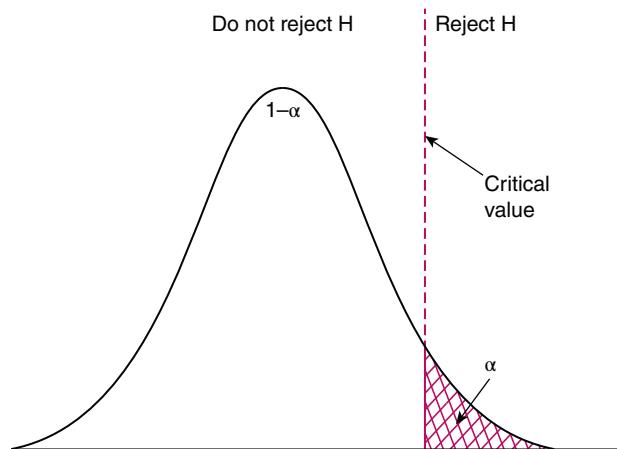


Figure 2.17 A right-tailed test with a normal probability distribution.

In summary, there are several operations we need to perform for statistical tests, starting with calculating a statistical value from a mathematical formula, then calculating the critical value using statistical tables, calculating the p -value, and checking whether $p \leq 0.05$ to accept or reject the null hypothesis.

For the filter methods, a classical way to begin is to consider the type of data we have. For instance, if both our input and output are categorical data, the most often used technique is the chi-squared test (`chi2()` in scikit-learn). We can also use mutual information (information gain). If our input is a continuous variable and our output is a categorical variable, ANOVA (`f_classif()` in scikit-learn), correlation coefficient (linear), or Kendall's rank coefficient (nonlinear) is appropriate. Pearson's (`r_regression()` in scikit-learn) correlation coefficient (linear) and Spearman's rank coefficient (nonlinear) can be used for both numerical input and output. SciPy library can also be used to implement all statistics.

To summarize, all these objects return univariate scores and p -values. For regression tasks, we can use `f_regression` and `mutual_info_regression`; for classification tasks, we can use `chi2`, `f_classif`, and `mutual_info_classif` options in scikit-learn.

Chi-Squared Test Let us now detail a feature selection method under the filter category, namely the chi-squared (χ^2) test, which is commonly used in statistics to test statistical independence between two or more categorical variables. A and B are two independent events if:

$$P(AB) = P(A)P(B) \text{ or, equivalently } P(A | B) = P(A) \text{ and } P(B | A) = P(B)$$

To correctly apply a chi-squared test, the data must be non-negative, sampled independently, and categorical, such as Booleans or frequencies (greater than 5, as chi-square is sensitive to small frequencies). The test is not appropriate for continuous variables or for comparison of categorical and continuous variables. In statistics, if we want to test the correlation (dependence) of two continuous variables, Pearson correlation is commonly used. When one variable is continuous and the other one is ordinal, Spearman's rank correlation is appropriate. It is possible to apply the chi-squared test on continuous variables after binning the variables. The idea behind the chi-squared test is to calculate a p -value in order to identify a high correlation (low p -value) between two categorical variables, indicating that the variables are dependent on each other ($p < 0.05$). The p -value is calculated from the chi-squared score and degrees of freedom.

In feature selection, we calculate chi-squared between each feature and the target. We can select the desired number of features based on the best chi-squared scores.

The test statistic for the chi-squared test is:

$$\chi_c^2 = \sum \left[\frac{(O_i - E_i)^2}{E_i} \right]$$

where O is the observed values, E is the expected values, and the degrees of freedom $c = k - r$, where k is the number of groups for which observed and expected frequencies are available and r is the number of restrictions or constraints imposed on the given comparison.

Let us be more concrete and calculate chi-squared in a concrete and simple example. Let us assume we have 3145 voters and that we want to assess whether gender influences the choice of a politician:

	Politician A	Politician B	Independent	Total
Women	630	855	100	1585
Men	795	675	90	1560
Total	1425	1530	190	3145

First, we need to define our hypotheses:

- **H0:** There is no link between gender and choosing a politician.
- **H1:** A link exists between gender and choosing a politician.

Now, we need to calculate the expected values:

$$\text{Expected Value} = \frac{(\text{Row Total}) \times (\text{Column Total})}{\text{Total number of observations}}$$

which means, in the case of women voting for politician A, $E_1 = (1425 \times 1585)/3145 = 718.2$

Applying this calculation to all the data gives the following expected values:

	Politician A	Politician B	Independent	Total
Women	718.163752	771.081081	95.7551669	1585
Men	706.836248	758.918919	94.2448331	1560
Total	1425	1530	190	3145

Now, we can apply the general formula $\chi^2_c = \sum \left[\frac{(O_i - E_i)^2}{E_i} \right]$:

	Politician A	Politician B	Independent	Total
Women	12.3378526	8.23670755	0.18018608	20.7547463
Men	9.77716624	10.4331629	0.20020675	20.4105359
Total	22.1150189	18.6698704	0.38039283	41.1652822

$$\chi^2_c = 41.1652822$$

As described above, we need to calculate the degrees of freedom. In our case, $c = (\text{number of columns} - 1) \times (\text{number of rows} - 1) = (3 - 1) \times (2 - 1) = 2$.

We can now check in the chi-squared distribution table (Figure 2.18). By comparing our chi-squared statistic (41.165) to the one in the table for an alpha level of 0.05 and two degrees of freedom (5.991), we can see that our obtained statistic is higher than the one in the table (critical value), meaning that we can reject the null hypothesis (H_0) and conclude that some evidence exists for an association between gender and choice of politician.

To implement chi-squared test for feature selection, let us use the mushroom dataset.

Input:

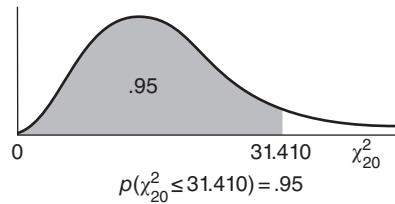
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

csv_data = '../data/datasets/mushrooms.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Encode the data
enc = LabelEncoder()
df = df.apply(enc.fit_transform)

# Divide the data, y the variable to classify and X the features
y = df.loc[:, df.columns == 'class'].values.ravel()
X = df.loc[:, df.columns != 'class']

# Splitting the data : training and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
print(X_train.shape)
```



d.f.	$\chi^2_{.005}$	$\chi^2_{.025}$	$\chi^2_{.05}$	$\chi^2_{.90}$	$\chi^2_{.95}$	$\chi^2_{.975}$	$\chi^2_{.99}$	$\chi^2_{.995}$
1	.0000393	.000982	.00393	2.706	3.841	5.024	6.635	7.879
2	.0100	.0506	.103	4.605	5.991	7.378	9.210	10.597
3	.0717	.216	.352	6.251	7.815	9.348	11.345	12.838
4	.207	.484	.711	7.779	9.488	11.143	13.277	14.860
5	.412	.831	1.145	9.236	11.070	12.832	15.086	16.750
6	.676	1.237	1.635	10.645	12.592	14.449	16.812	18.548
7	.989	1.690	2.167	12.017	14.067	16.013	18.475	20.278
8	1.344	2.180	2.733	13.362	15.507	17.535	20.090	21.955
9	1.735	2.700	3.325	14.684	16.919	19.023	21.66	23.589
10	2.156	3.247	3.940	15.987	18.307	20.483	23.209	25.188
11	2.603	3.816	4.575	17.275	19.675	21.920	24.725	26.757
12	3.074	4.404	5.226	18.549	21.026	23.336	26.217	28.300
13	3.565	5.009	5.892	19.812	22.362	24.736	27.688	29.819
14	4.075	5.629	6.571	21.064	23.685	26.119	29.141	31.319
15	4.601	6.262	7.261	22.307	24.996	27.488	30.578	32.801
16	5.142	6.908	7.962	23.542	26.296	28.845	32.000	34.267
17	5.697	7.564	8.672	24.769	27.587	30.191	33.409	35.718
18	6.265	8.231	9.390	25.989	28.869	31.526	34.805	37.156
19	6.844	8.907	10.117	27.204	30.144	32.852	36.191	38.582
20	7.434	9.591	10.851	28.412	31.410	34.170	37.566	39.997
21	8.034	10.283	11.591	29.615	32.671	35.479	38.932	41.401
22	8.643	10.982	12.338	30.813	33.924	36.781	40.289	42.796
23	9.260	11.688	13.091	32.007	35.172	38.076	41.638	44.181
24	9.886	12.401	13.848	33.196	36.415	39.364	42.980	45.558
25	10.520	13.120	14.611	34.382	37.652	40.646	44.314	46.928
26	11.160	13.844	15.379	35.563	38.885	41.923	45.642	48.290
27	11.808	14.573	16.151	36.741	40.113	43.194	46.963	49.645
28	12.461	15.308	16.928	37.916	41.337	44.461	48.278	50.993
29	13.121	16.047	17.708	39.087	42.557	45.722	49.588	52.336
30	13.787	16.791	18.493	40.256	43.773	46.979	50.892	53.672
35	17.192	20.569	22.465	46.059	49.802	53.203	57.342	60.275
40	20.707	24.433	26.509	51.805	55.758	59.342	63.691	66.766
45	24.311	28.366	30.612	57.505	61.656	65.410	59.957	73.166
50	27.991	32.357	34.764	63.167	67.505	71.420	76.154	79.490
60	35.535	40.482	43.188	74.397	79.082	83.298	88.3979	91.952
70	43.275	48.758	51.739	85.527	90.531	95.023	100.425	104.215
80	51.172	57.153	60.391	96.578	101.879	106.629	112.329	116.321
90	59.196	65.647	69.126	107.565	113.145	118.136	124.116	128.299
100	67.328	74.222	77.929	118.498	124.342	129.561	135.807	140.169

Figure 2.18 Percentiles of the chi-squared distribution. Source: Daniel and Cross (2018).

Output:

(5686, 22)

As we can see by printing the shape of the data, we have 22 features. We can perform a chi-squared test on the samples to retrieve only the two best features using *SelectKBest*, which will remove all but the two highest-scoring features.

Input:

```
# Perform a chi-square test to the samples to retrieve only the two best features

from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

X_training_new = SelectKBest(chi2, k=2).fit_transform(X_train, y_train)
print(X_training_new.shape)
```

Output:

(5686, 2)

It is also possible to use *SelectPercentile* to select features according to a percentile of the highest scores:

```
X_new = SelectPercentile(chi2, percentile=10).fit_transform(X, y)
```

ANOVA F-Value As we have seen, if the features are categorical, we can choose to examine chi-squared statistics between the features and the target vector. If the features are continuous variables and the target vector is categorical, the analysis of variance (ANOVA) F-statistic can be calculated to determine whether the means of each group (features by the target vector) are significantly different. When we run an ANOVA test or a regression analysis, we can compute an F-statistic (F-test) to statistically assess the equality of means. An F-test is similar to a *t*-test, which determines whether a single feature is statistically different; the F-test will determine whether the means of three or more groups are different. In fact, when we apply the ANOVA F-test to only two groups, $F = t^2$ where t is the student's *t*-statistic. Similar to other statistical tests, we will obtain an F-value, a critical F-value, and a *p*-value. For the one-way ANOVA F-test statistic, the formula is the following:

$$F = \frac{\frac{1}{K-1} \sum_{i=1}^K n_i (\bar{X}_i - \bar{X})^2}{\frac{1}{N-K} \sum_{i=1}^K \sum_{j=1}^{n_i} (X_{ij} - \bar{X}_i)^2}$$

The numerator is the explained variance or between-group variability, and the denominator is the unexplained variance or within-group variability. \bar{X}_i is the sample mean in the i th group, \bar{X} is the overall mean of our data, K is the number of groups, n_i is the number of observations in the i th group, X_{ij} is the j th observation in the i th out of K groups, and N is the overall sample size. The test statistic is compared to a quantile in the F-distribution. In the F-test, the data values need to be independent and normally distributed with a common variance.

In linear regression, the F-test can be used to determine whether we are able to improve our linear regression model by making it more complex (adding more linear regression variables) or if it would be better to swap our complex model with an intercept-only model (simple linear regression model).

The unrestricted model is the following:

$$\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_0 = y$$

The restricted model is the following:

$$\beta'_1 x_1 + \beta'_2 x_2 + \beta'_3 x_3 + \beta'_0 = y$$

The intercept-only model is the following:

$$\beta''_0 = y$$

If we consider two regression models (model 1 and model 2) for which model 1 has k_1 parameters and model 2 has k_2 parameters with $k_1 < k_2$, then model 1 (restricted model) is the simpler version of model 2 (unrestricted model).

We can then calculate the F-statistic as follows:

$$F = \frac{\frac{RSS_1 - RSS_2}{k_2 - k_1}}{\frac{RSS_2}{n - k_2}}$$

where RSS_1 is the residual sum of squares of fitted model 1, RSS_2 is the residual sum of squares of fitted model 2, and n is the number of data samples.

To use an ANOVA F-test to select features, let us use a well-known binary classification dataset that is a copy of the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset. It is composed of two classes (WDBC-Malignant, WDBC-Benign) and 30 numeric attributes, with 569 samples (212 for malignant and 357 for benign). The features have been computed from a digitized image of a fine needle aspirate (FNA) of a breast mass and describe characteristics of the cell nuclei present in the image.

Input:

```
# Import breast cancer from sklearn.datasets
from sklearn.datasets import load_breast_cancer

# Loading the data from breast cancer dataset
breast_cancer = load_breast_cancer()

# Splitting the data into X and y
X = breast_cancer.data
y = breast_cancer.target

# Splitting the data : training and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
print(X_train.shape)
```

Output:

(398, 30)

Let us now apply some Python code to select features according to the two best ANOVA F-values.

Input:

```
# Select Features with Best ANOVA F-Values
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

# Create an SelectKBest object to select features with two best ANOVA F-Values
fvalue_selector = SelectKBest(f_classif, k=2)

# Apply the SelectKBest object to the training features (X_train) and target (y_train)
X_train_f_classif = fvalue_selector.fit_transform(X_train, y_train)
print(X_train_f_classif.shape)
```

Output:

(398, 2)

For the regression task, we can use a medical cost personal dataset (<https://www.kaggle.com/mirichoi0218/insurance>), which can be used for insurance forecasts by linear regression. In the dataset, we will find costs billed by health insurance companies (insurance charges) and features (age, gender, BMI, children, smoking status).

Let us examine the head of the data:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520
...
1333	50	male	30.970	3	no	northwest	10600.54830
1334	18	female	31.920	0	no	northeast	2205.98080
1335	18	female	36.850	0	no	southeast	1629.83350
1336	21	female	25.800	0	no	southwest	2007.94500
1337	61	female	29.070	0	yes	northwest	29141.36030

1338 rows × 7 columns

Input:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

csv_data = '../data/datasets/insurance.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Encode the data and drop original column from df
enc = LabelEncoder()
df_encoded = df[['sex', 'smoker', 'region']].apply(enc.fit_transform)
df = df.drop(['sex', 'smoker', 'region'], axis = 1)

# Concatenate dataframes
df = pd.concat([df, df_encoded], axis=1)

# Divide the data
y = df.loc[:, df.columns == 'charges'].values.ravel()
X = df.loc[:, df.columns != 'charges']

print(X.shape)

# Splitting the data : training and test
from sklearn.model_selection import train_test_split
```

```

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

print(X_train.shape)

# Select Features With Best ANOVA F-Values
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression

# Create an SelectKBest object to select features with two best ANOVA F-Values
f_value = SelectKBest(f_regression, k=2)

# Apply the SelectKBest object to the training features (X_train) and target (y_train)
X_train_f_regression = f_value.fit_transform(X_train, y_train)
print(X_train_f_regression.shape)

```

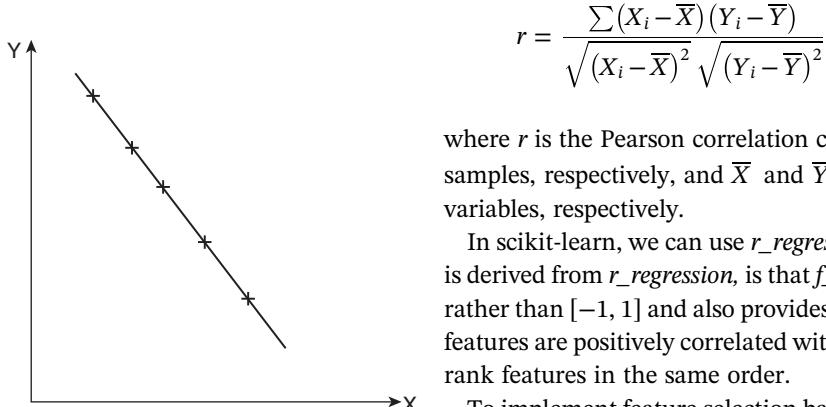
Output:

```
(1338, 6)
(936, 6)
(936, 2)
```

As we can see, from six features, we have kept only two.

Pearson Correlation Coefficient We can use the Pearson correlation coefficient (r) to measure the linear relationship between two or more variables or, in other words, how much we can predict one variable from another. We can use this number for feature selection with the idea that the variables to keep are those that are highly correlated with the target and uncorrelated among themselves. The Pearson correlation coefficient is a number between -1 and 1 . A value close to 1 indicates a strong positive correlation (if $r = 1$, there is a perfect linear correlation between two variables). A value close to -1 means a strong negative correlation ($r = -1$ indicates a perfect inverse linear correlation). Values close to 0 indicate weak correlation (0 means no linear correlation at all) (Figure 2.19).

To compute the linear coefficient of correlation r , the following formula can be used:



where r is the Pearson correlation coefficient, X_i and Y_i are the X and Y variable samples, respectively, and \bar{X} and \bar{Y} are the means of the values of the X and Y variables, respectively.

In scikit-learn, we can use $r_regression$. The difference from $f_regression$, which is derived from $r_regression$, is that $f_regression$ produces values in the range $[0, 1]$ rather than $[-1, 1]$ and also provides p -values, in contrast to $r_regression$. If all the features are positively correlated with the target, $f_regression$ and $r_regression$ will rank features in the same order.

To implement feature selection based on the coefficient of correlation r , we will use the Breast Cancer Dataset from the UCI ML Breast Cancer Wisconsin (Diagnostic).

Figure 2.19 Perfect inverse linear correlation ($r = -1$).

Input:

```

import pandas as pd
from sklearn.preprocessing import LabelEncoder

breastcancer = '../data/breastcancer.csv'
df = pd.read_csv(breastcancer, delimiter=';')

# Encode the data and drop original column from df + remove id variable
enc = LabelEncoder()
df_encoded = df[['diagnosis']].apply(enc.fit_transform)
df = df.drop(['diagnosis', "id"], axis = 1)

# Concatenate dataframes
df = pd.concat([df, df_encoded], axis=1)

# Divide the data
y = df.loc[:, df.columns == 'diagnosis'].values.ravel()
X = df.loc[:, df.columns != 'diagnosis']

# Convert feature matrix into DataFrame
df = pd.DataFrame(X)

# View the data frame
print(df)

```

Output:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	\
0	17.99	10.38	122.80	1001.0	0.11840	
1	20.57	17.77	132.90	1326.0	0.08474	
2	19.69	21.25	130.00	1203.0	0.10960	
3	11.42	20.38	77.58	386.1	0.14250	
4	20.29	14.34	135.10	1297.0	0.10830	
..
564	21.56	22.39	142.00	1479.0	0.11100	
565	20.13	28.25	131.20	1261.0	0.09780	
566	15.60	28.08	108.30	858.1	0.08455	
567	20.60	29.33	140.10	1265.0	0.11780	
568	7.76	24.54	47.92	181.0	0.05263	
	compactness_mean	concavity_mean	concave_points_mean	symmetry_mean	\	
0	0.27760	0.30010	0.14710	0.2419		
1	0.07864	0.08690	0.07017	0.1812		
2	0.15900	0.19740	0.12790	0.2069		
3	0.28390	0.24140	0.19520	0.2597		
4	0.13280	0.19800	0.10430	0.1809		
..		
564	0.11590	0.24390	0.13890	0.1726		
565	0.18340	0.14400	0.09791	0.1752		
566	0.10230	0.09251	0.05302	0.1590		
567	0.27700	0.35140	0.15200	0.2397		
568	0.04362	0.00000	0.00000	0.1587		
	fractal_dimension_mean	...	radius_worst	texture_worst	\	
0	0.07871	...	25.380	17.33		
1	0.05667	...	24.990	23.41		
2	0.05990	...	23.570	25.53		
3	0.09744	...	14.910	26.58		
4	0.05883	...	22.540	16.67		
..		
564	0.05623	...	25.450	26.48		
565	0.05533	...	23.690	38.25		
566	0.05648	...	18.980	34.12		
567	0.07016	...	25.740	39.42		
568	0.05884	...	9.456	30.37		

```

perimeter_worst    area_worst    smoothness_worst    compactness_worst    \n
0      184.60      2019.0       0.16220       0.66560
1      158.80      1956.0       0.12380       0.18660
2      152.50      1709.0       0.14440       0.42450
3      98.87       567.7        0.20980       0.86630
4      152.20      1575.0       0.13740       0.20500
..      ...
564     166.10      2027.0       0.14100       0.21130
565     155.00      1731.0       0.11660       0.19220
566     126.70      1124.0       0.11390       0.30940
567     184.60      1821.0       0.16500       0.86810
568     59.16       268.6        0.08996       0.06444

concavity_worst    concave_points_worst    symmetry_worst    \
0      0.7119       0.2654       0.4601
1      0.2416       0.1860       0.2750
2      0.4504       0.2430       0.3613
3      0.6869       0.2575       0.6638
4      0.4000       0.1625       0.2364
..      ...
564     0.4107       0.2216       0.2060
565     0.3215       0.1628       0.2572
566     0.3403       0.1418       0.2218
567     0.9387       0.2650       0.4087
568     0.0000       0.0000       0.2871

fractal_dimension_worst
0      0.11890
1      0.08902
2      0.08753
3      0.17300
4      0.07673
..      ...
564     0.07115
565     0.06637
566     0.07820
567     0.12400
568     0.07039

```

[569 rows x 30 columns]

Let us now display the correlation matrix (heatmap).

Input:

```

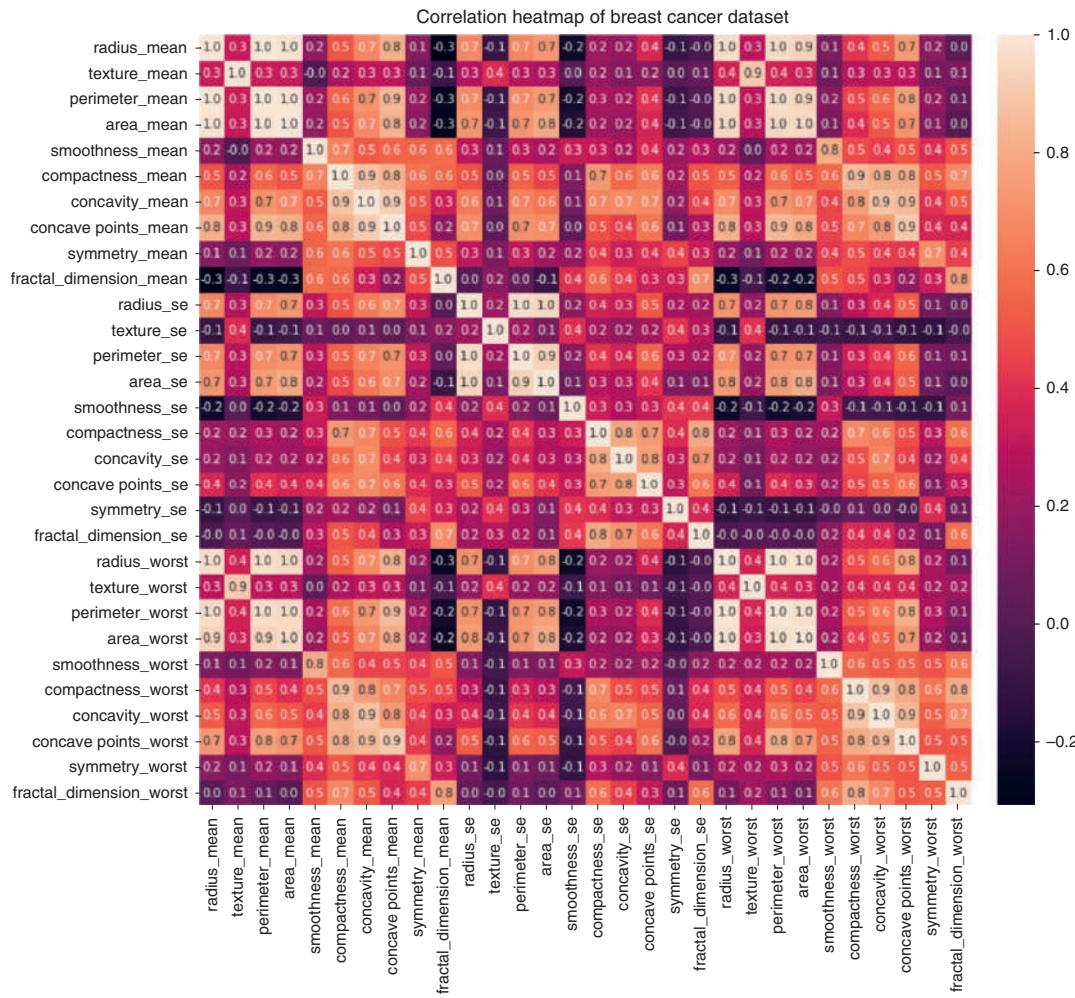
import matplotlib.pyplot as plt
import seaborn as sns

# Create correlation matrix
corr_matrix = df.corr()

# Create correlation heatmap
plt.figure(figsize=(16,12))
plt.title('Correlation Heatmap of Breast Cancer Dataset')
a = sns.heatmap(corr_matrix, square=True, annot=True, fmt=' .1f', linecolor='black')
a.set_xticklabels(a.get_xticklabels())
a.set_yticklabels(a.get_yticklabels())
plt.show()

```

Output:



As stated above, the main idea in feature selection is to retain the variables that are highly correlated with the target (“diagnosis” in our case) and keep features that are uncorrelated among themselves. For instance, we can search for the index of feature columns with correlation greater than 0.8.

Input:

```
import numpy as np
# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
# Find index of feature columns with correlation greater than 0.8
to_drop = [column for column in upper.columns if any(upper[column] > 0.8)]
print(to_drop)
```

Output:

```
['perimeter_mean', 'area_mean', 'concavity_mean', 'concave points_mean',
'perimeter_se', 'area_se', 'concavity_se', 'fractal_dimension_se', 'radius_worst',
'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst',
'compactness_worst', 'concavity_worst', 'concave points_worst',
'fractal_dimension_worst']
```

Now that we have identified the features to drop, we can drop them from the original dataset.

Input:

```
# Drop Marked Features
df_new = df.drop(to_drop, axis=1)
```

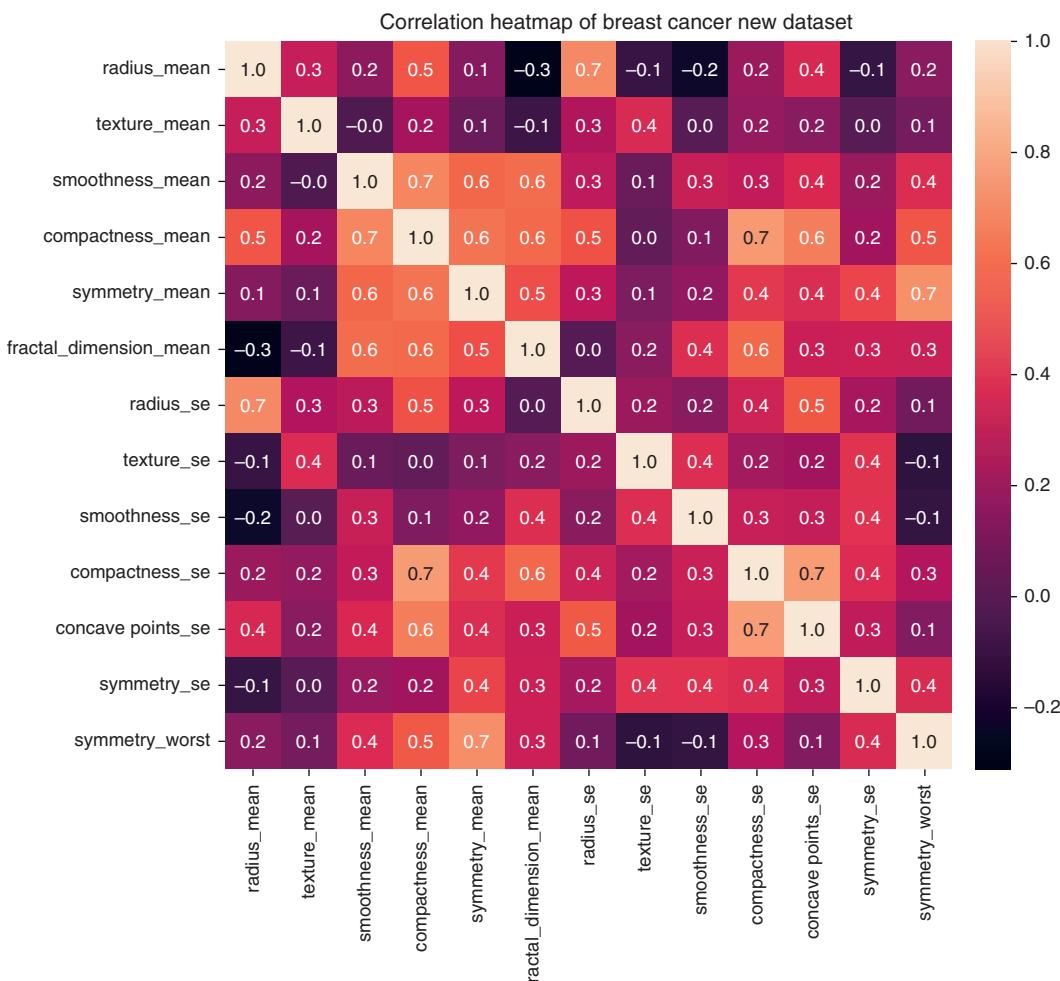
From the new generated dataset (*df_new*), we now have to set an absolute value for the threshold to select features that are correlated with the target. Let us choose 0.5.

Input:

```
# Create correlation matrix
corr_matrix_new = df_new.corr()

# Create correlation heatmap
plt.figure(figsize=(16,12))
plt.title('Correlation Heatmap of Breast Cancer New Dataset')
a = sns.heatmap(corr_matrix_new, square=True, annot=True, fmt='.1f',
linecolor='black')
a.set_xticklabels(a.get_xticklabels())
a.set_yticklabels(a.get_yticklabels())
plt.show()
```

Output:



Input:

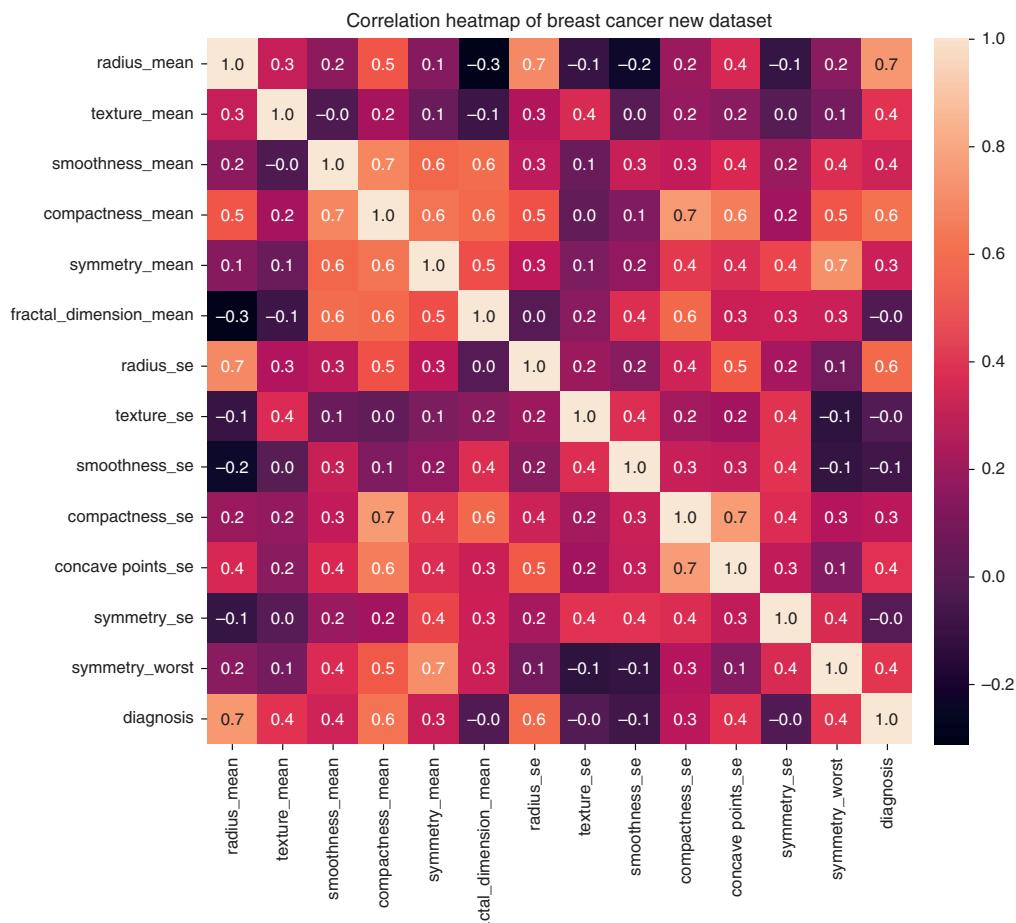
```
# Concatenate dataframes (X and the target y called diagnosis)
df_y = pd.DataFrame(y, columns = ["diagnosis"])
df_all = pd.concat([df_new, df_y], axis=1)

#Correlation with target variable
cor = df_all.corr()
cor_target = abs(cor["diagnosis"])

# Create correlation heatmap
plt.figure(figsize=(16,12))
plt.title('Correlation Heatmap of Breast Cancer New Dataset')
a = sns.heatmap(cor, square=True, annot=True, fmt='.1f', linecolor='black')
a.set_xticklabels(a.get_xticklabels())
a.set_yticklabels(a.get_yticklabels())
plt.show()

#Selecting highly correlated features
relevant_features = cor_target[cor_target>0.5]
print("Features to keep : \n")
print(relevant_features)
```

Output:



Features to retain:

radius_mean	0.730029
compactness_mean	0.596534
radius_se	0.567134
diagnosis	1.000000

As we can see from using the correlation coefficient r , the features to retain in our future model are radius_mean, compactness_mean, and radius_se.

We can also compute Pearson's r for each feature and the target using `r_regression` from scikit-learn.

```
from sklearn.feature_selection import r_regression
print(r_regression(X, y))
```

Filter Methods: Many More Possibilities As we have seen, the filter methodologies are based on univariate metrics that allow us to select features based on a ranking such as variance, chi-squared, correlation coefficient, or information gain (mutual information). We can also use other methods not described above such as missing value ratio in which we compute the number of missing values in each feature divided by the total number of observations. After defining a threshold, we can eliminate a column or not. Instead of using the variance, we can remove the square and alternatively calculate the mean absolute difference. Another possibility is to assess the dispersion ratio between the arithmetic mean and the geometric mean for a specific feature and retain features with higher dispersion ratios or to determine whether two variables are mutually dependent by calculating the amount of information that a feature contributes to making the prediction for the other feature such as the target.

There are many ways to select features with filter methods. The advantage of filter methods is that they are model-agnostic and fast to compute. The disadvantage is that they examine individual features only, which can be an issue. A feature may not be useful on its own but it may have much more influence on the target if combined with others.

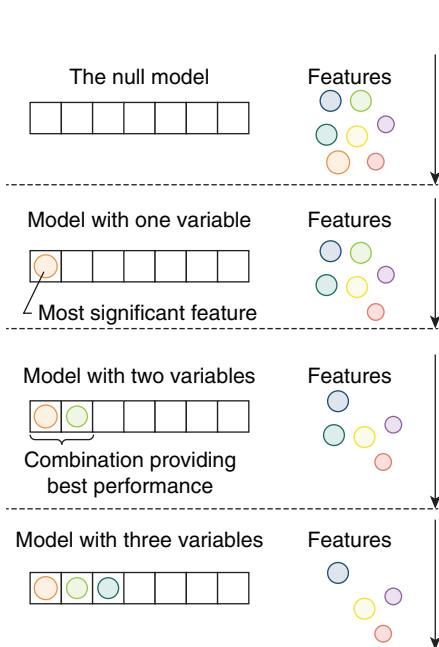


Figure 2.20 Forward stepwise selection starts with no feature in the model and iteratively adds the best performing features, one by one, against the target until a feature does not improve the model performance or there are no more features to add.

2.5.2.2 Wrapper Methods

In wrapper methods, we use a subset of features to train a machine learning algorithm to add or remove features based on inferences designed from the model. Wrapper methods are generally computationally expensive but often achieve better results than filter methods because they are tuned to the specific interaction between an induction algorithm and its training data. Wrapper methods are slower because we need to repeatedly call the induction algorithm and run it again when we use a different induction algorithm. There are different wrapper methods, such as forward stepwise selection, backward elimination, exhaustive feature selection, recursive feature elimination, and recursive feature elimination with cross-validation.

Forward Stepwise Selection Forward stepwise selection is an iterative method that starts with no features in the model and iteratively adds the best performing features, one by one, against the target (which best improves the model) until a feature does not improve the model performance or the features are exhausted. To follow this process, it is necessary to evaluate all features individually to select the one that provides the best performance in the algorithm based on preset evaluation criteria. We then combine the selected feature with another one that has been selected based on the performance of the algorithm (selecting the pair with the best score). As mentioned above regarding wrapper methods, forward stepwise selection can be computationally expensive because the selection procedure, usually called "greedy," evaluates all possible feature combinations (single, double, triple, etc.). If the feature space is large, sometimes it is simply not practical (Figure 2.20).

For classification, we can compute the area under the receiver operating characteristic curve (ROC AUC) from prediction scores and use it to check or visualize the performance of a classification problem. To summarize, the method indicates how well a model can distinguish between the different classes, with ROC being a probability curve and AUC a measure of separability (the higher the AUC, the better the model). We could also choose accuracy, precision, recall, f1-score, or another metric. For regression, we can compute R-squared, which indicates the strength of the relationship between our model and the dependent variable.

To implement wrapper methods and perform feature selection, we will analyze a dataset used to recognize fraudulent credit card transactions (<https://www.kaggle.com/mlg-ulb/creditcardfraud>). The dataset contains transactions made by credit cards in September 2013 by European cardholders over two days, in which we can find 492 frauds (0.172%) out of 284,807 transactions. The data have been transformed using PCA. The principal components (V1, V2, V3, ...) are the new features except for “Time” and “Amount,” which are the original ones. The feature “Class” is the response variable with a value of 1 in case of fraud and 0 otherwise.

Input:

```
import pandas as pd

csv_data = '../data/datasets/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()
```

Output:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339848	0.167170	-0.125895	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	-0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	-0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

5 rows × 31 columns

To implement forward selection, we can use the *mlxtend* library, which contains most of the feature selection techniques based on wrapper methods. Of note, the stopping criteria in mlxtend implementation are an arbitrarily set number of features.

Input:

```
# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Importing the necessary libraries for Forward stepwise feature selection
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.linear_model import LinearRegression

# SFS = Sequential Forward Selection
sfs = SFS(LinearRegression(),
          k_features=5,
```

```

        forward=True,
        floating=False,
        scoring = 'r2',
        cv = 0)

# Fit the model
sfs.fit(X_train, y_train)

# Print selected features (we could replace the two following lines with print(sfs.
k_feature_names_)
selected_feat= X.columns[list(sfs.k_feature_idx_)]
print(selected_feat)

```

Output:

```
Index(['V10', 'V12', 'V14', 'V16', 'V17'], dtype='object')
```

In the sequential forward selection (SFS) section of the code above, we have set some parameters. First, we have chosen the LinearRegression() estimator for the process. We could choose any scikit-learn classifier or regressor. We have also chosen to select the “best” five features from the dataset. This number can be any that we select, but we can also assess the optimal value by analyzing the scores for different numbers. Here, we set forward as true and floating as false for the forward selection technique. The evaluation criterion is provided by the parameter scoring. Here, we have chosen R-squared; we have not chosen k-fold cross-validation. We can read the selected feature indices as follows:

Input:

```
sfs.subsets_
```

Output:

```

{1: {'feature_idx': (17,), 
      'cv_scores': array([0.10679159]), 
      'avg_score': 0.10679159377776526, 
      'feature_names': ('V17',)}, 
2: {'feature_idx': (14, 17), 
      'cv_scores': array([0.20079793]), 
      'avg_score': 0.20079793094854692, 
      'feature_names': ('V14', 'V17')}, 
3: {'feature_idx': (12, 14, 17), 
      'cv_scores': array([0.27027216]), 
      'avg_score': 0.27027215811858596, 
      'feature_names': ('V12', 'V14', 'V17')}, 
4: {'feature_idx': (10, 12, 14, 17), 
      'cv_scores': array([0.31588019]), 
      'avg_score': 0.31588019129581957, 
      'feature_names': ('V10', 'V12', 'V14', 'V17')}, 
5: {'feature_idx': (10, 12, 14, 16, 17), 
      'cv_scores': array([0.35472495]), 
      'avg_score': 0.35472495313955954, 
      'feature_names': ('V10', 'V12', 'V14', 'V16', 'V17')}}

```

The prediction score for our five features is computed as follows:

Input:

```
sfs.k_score_
```

Output:

```
0.35472495313955954
```

Let us now change some parameters. For example, we can use KNN instead of linear regression and use cross-validation.

Input:

```
# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Importing the necessary libraries for Forward stepwise feature selection
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
#from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier

# SFS = Sequential Forward Selection
sfs = SFS(KNeighborsClassifier(), # we can use any scikit-learn classifier or
regressor.
           k_features=5,          # number of features we want to keep
           forward=True,
           floating=False,
           scoring = 'accuracy',
           cv = 5,                # Cross-validation
           n_jobs=-1)             # Run the cross-validation on all our available
                                   # CPU cores.

# Fit the model
sfs.fit(X_train, y_train)

# Print selected features (we could replace the two following lines with print(sfs.
k_feature_names_)
selected_feat= X.columns[list(sfs.k_feature_idx_)]
print(selected_feat)
```

Output:

```
Index(['V3', 'V4', 'V12', 'V14', 'V17'], dtype='object')
```

With this configuration, the process takes more time to compute when run on a personal computer. We have added the `n_jobs = -1` option in SFS to run the cross-validation on all our available CPU cores.

Let us read the selected feature indices:

Input:

```
sfs.subsets_
```

Output:

```
{1: {'feature_idx': (12,) ,
  'cv_scores': array([0.99918804, 0.99918804, 0.99892471, 0.9989686 , 0.9991661 ]),
  'avg_score': 0.9990870986855098,
  'feature_names': ('V12',)},
2: {'feature_idx': (12, 17),
  'cv_scores': array([0.99927582, 0.99938555, 0.99938555, 0.99929777, 0.99931971]),
  'avg_score': 0.9993328798086418,
  'feature_names': ('V12', 'V17')},
3: {'feature_idx': (4, 12, 17),
  'cv_scores': array([0.99934166, 0.9993636 , 0.99940749, 0.99947333, 0.99947333]),
  'avg_score': 0.9994118808839343,
  'feature_names': ('V4', 'V12', 'V17')},
4: {'feature_idx': (3, 4, 12, 17),
  'cv_scores': array([0.99945138, 0.99942944, 0.9993636 , 0.99951722, 0.99949527]),
  'avg_score': 0.9994513814215805,
  'feature_names': ('V3', 'V4', 'V12', 'V17')},
5: {'feature_idx': (3, 4, 12, 14, 17),
  'cv_scores': array([0.99942944, 0.99942944, 0.99945138, 0.99953916, 0.99951722]),
  'avg_score': 0.9994733261647173,
  'feature_names': ('V3', 'V4', 'V12', 'V14', 'V17')}}
```

Let us also compute the prediction score for our five features:

Input:

```
sfs.k_score_
```

Output:

0.9994733261647173

If we desire more details, we can input the following:

```
pd.DataFrame.from_dict(sfs.get_metric_dict()).T
```

Output:

	feature_idx	cv_scores	avg_score	feature_names	ci_bound	std_dev	std_err
1	(12,)	[0.999188044503939, 0.999188044503939, 0.99892...	0.999087	(V12,)	0.000149	0.000116	0.000058
2	(12, 17)	[0.9992758234764862, 0.9993855471921701, 0.999...	0.999333	(V12, V17)	0.000058	0.000045	0.000023
3	(4, 12, 17)	[0.9993416577058966, 0.9993636024490333, 0.999...	0.999412	(V4, V12, V17)	0.00007	0.000054	0.000027
4	(3, 4, 12, 17)	[0.9994513814215804, 0.9994294366784436, 0.999...	0.999451	(V3, V4, V12, V17)	0.000069	0.000054	0.000027
5	(3, 4, 12, 14, 17)	[0.9994294366784436, 0.9994294366784436, 0.999...	0.999473	(V3, V4, V12, V14, V17)	0.000059	0.000046	0.000023

Backward Elimination In backward elimination, contrary to forward stepwise, we start with the full model (all features including the independent ones). The objective is to eliminate the least significant feature (the worst feature with the highest p -value > significant level) at each iteration until no improvement of the performance model is observed.

To implement backward stepwise selection, we follow the same procedure as forward stepwise selection with the use of the *mlxtend* library.

Input:

```
import pandas as pd
import numpy as np

# Import data from a csv and create a DataFrame
csv_data = '../data/datasets/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Importing the necessary libraries for Forward stepwise feature selection
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.tree import ExtraTreeClassifier

# SFS = Sequential Forward Selection
sfs_back = SFS(ExtraTreeClassifier(), # we can use any scikit-learn classifier or
regressor.
               k_features=5,           # number of features we want to keep
               forward=False,
               floating=False,
               scoring = 'accuracy',
               cv = 5,                 # Cross-validation
               n_jobs=-1)              # Run the cross-validation on all our available
                                         # CPU cores.

# Fit the model
sfs_back.fit(X_train, y_train)

# Print selected features
print(sfs_back.k_feature_names_)
```

Output:

```
Index(['Time', 'V2', 'V6', 'V10', 'V17'], dtype='object')
```

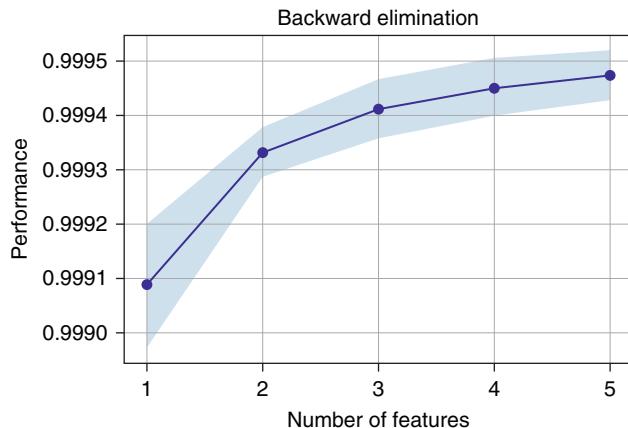
Here, we have chosen the *ExtraTreeClassifier()* estimator for the process. We have also set forward as false and floating as false for the backward elimination technique.

We can also plot the performance versus the number of features:

Input:

```
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
import matplotlib.pyplot as plt
fig = plot_sfs(sfs_back.get_metric_dict(), kind='std_dev')
plt.title('Backward Elimination')
plt.grid()
plt.show()
```

Output:



Let us now use RandomForestClassifier as another example. The script below can take quite a bit of time to execute.

Input:

```
import pandas as pd
import numpy as np

# Import data from a csv and create a DataFrame
csv_data = '../data/datasets/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Importing the necessary libraries for Forward stepwise feature selection
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.ensemble import RandomForestClassifier

# SFS = Sequential Forward Selection
sfs_back = SFS(RandomForestClassifier(), # we can use any scikit-learn classifier or
regressor.
```

```

        k_features=5,           # number of features we want to keep
        forward=False,
        floating=False,
        scoring = 'accuracy',
        cv = 5,                 # Cross-validation
        n_jobs=-1)              # Run the cross-validation on all our available
                                # CPU cores.

# Fit the model
sfs_back.fit(X_train, y_train)

# Print selected features
print(sfs_back.k_feature_names_)

```

Output:

```
Index(['V4', 'V7', 'V14', 'V15', 'V17'], dtype='object')
```

Exhaustive Feature Selection Exhaustive feature selection is a brute-force evaluation of feature subsets that evaluates model performance (such as classification accuracy) with all feature combinations. For instance, if there are three features, the model will be tested with feature 0 only, then feature 1 only, feature 2 only, features 0 and 1, features 0 and 2, features 1 and 2, and features 0, 1, and 2. Like the other wrapper methods, the method is computationally expensive (a greedy algorithm) due to its search for all combinations. We can use different approaches, such as reducing the search space, to reduce this time.

To implement this method, we can also use the *ExhaustiveFeatureSelector* function from the *mlxtend.feature_selection* library. As we can see in the script below, the class has *min_features* and *max_features* attributes to specify the minimum and maximum number of features desired in the combination.

Input:

```

import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

```

```

knn = KNeighborsClassifier(n_neighbors=3)

# Let's create our feature selector
efs1 = EFS(knn,
            min_features=2,
            max_features=4,
            scoring='accuracy',
            print_progress=True,
            cv=5)

# Call the fit method on our feature selector and pass it the training set
efs1 = efs1.fit(X_train, y_train)

# Print the results

print('Best accuracy score: %.2f' % efs1.best_score_)
print('Best subset (indices):', efs1.best_idx_)
print('Best subset (corresponding names):', efs1.best_feature_names_)

```

Output:

```

Best accuracy score: 1.00
Best subset (indices): (4, 10, 14, 16)
Best subset (corresponding names): ('V4', 'V10', 'V14', 'V16')

```

2.5.2.3 Embedded Methods

Performing feature selection with embedded methods can be a good alternative to both filter and wrapper methods. As the name of the method indicates, the feature selection process is performed during execution of the modeling algorithm (embedded in the algorithm). Wrapper and embedded methods both select features based on the learning procedure of the machine learning model with the difference that embedded methods compute feature selection and training in parallel, whereas wrapper methods use evaluation metrics to consider unimportant features iteratively. As mentioned above, both wrapper and filter methods retain or discard features in their discrete processes, which can result in high variance. In embedded methods, it is common to use algorithms such as multinomial logistic regression or decision tree algorithms such as random forest, CART, or C4.5. In embedded methods, regularization methods (also called penalization methods) are also widely used to minimize fitting errors and keep feature coefficients small or near zero. Regularization methods that penalize a feature that does not contribute to the model given a coefficient threshold are based on lasso regression (L1 regularization), ridge regression (L2 regularization), or elastic net (L1/L2 regularization) and usually work with linear classifiers. It is deployed to reduce overfitting. In other words, if we perform a regression analysis, the features are estimated using coefficients during the modeling process. To limit the impact of insignificant features, we need to reduce their weight to avoid high variance with a stable fit. This goal can be achieved if the estimates are restricted, reduced, or regularized toward zero.

Least Absolute Shrinkage and Selection Operator (Lasso) Lasso is a shrinkage method. It performs L1 regularization as follows:

$$\text{L1 Regularization} = \sum_{i=0}^N \left(y_i - \sum_{j=0}^M x_{ij} w_j \right)^2 + \alpha \sum_{j=0}^M |w_j|$$

As we can see, L1 regularization adds a penalty to the cost based on the complexity of the model. In the equation above, instead of calculating the cost with a loss function (the first term of the equation), there is an additional element (the second term of the equation) called the regularization term that is used to penalize the model. L1 regularization adds the absolute value of the magnitude of coefficient (the weights w). The hyperparameter α is a complexity parameter that is non-negative

and controls the amount of shrinkage. This is a hyperparameter that we should tune. A larger value produces a greater amount of shrinkage, resulting in a more simplified model. If α is 0, there is no elimination of the parameters; increasing α leads to increased bias, while decreasing α will increase the variance.

Let us take the dataset recorded from European cardholders that we have previously used and select features using lasso.

Input:

```
import pandas as pd
import numpy as np

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

from sklearn.linear_model import Lasso
sel_ = SelectFromModel(Lasso(alpha=0.01))
sel_.fit(X_train, y_train)

# Build a list of selected features and print it
selected_feat = X_train.columns[(sel_.get_support())]
print(selected_feat)

# Print total features, selected features and features with coefficients shrunk to
zero
print('total features: {}'.format((X_train.shape[1])))
print('selected features: {}'.format(len(selected_feat)))
print('features with coefficients shrunk to zero: {}'.format(
    np.sum(sel_.estimator_.coef_ == 0)))
```

Output:

```
Index(['V3', 'V12', 'V14', 'V17'], dtype='object')
total features: 30
selected features: 4
features with coefficients shrunk to zero: 24
```

As can be seen in the script, we have set α to 0.01. It is important to be careful with the α hyperparameter because the penalty can highly impact the performance of the model. If it is set too high, it can encourage the removal of important features.

It could also be important to print the weight values. We can do this with *eli5*.

Input:

```

import pandas as pd
import numpy as np

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

from sklearn import linear_model
regressor = linear_model.Lasso(alpha=0.000185,
                               positive=True,
                               fit_intercept=False,
                               max_iter=1000,
                               tol=0.0001)
regressor.fit(X_train, y_train)

# Print the results: the greater the weight the more important the feature
import eli5
eli5.show_weights(regressor, top=-1, feature_names = X_train.columns.tolist())

```

Output:

y top features

Weight?	Feature
+0.006	V11
+0.004	V4
+0.003	V2
+0.002	V21
+0.002	V19
+0.001	V8
+0.001	V27
+0.000	V25
+0.000	Amount
+0.000	Time

L2 Regularization (Ridge Regression) L2 regularization (ridge regression) adds a penalty that is equal to the square of the magnitude of coefficients:

$$L2 \text{ Regularization} = \sum_{i=0}^N \left(y_i - \sum_{j=0}^M x_{ij} w_j \right)^2 + \alpha \sum_{j=0}^M w_j^2$$

Ridge regression and lasso regression were built to make use of regularization for prediction by penalizing the magnitude of coefficients and minimizing the errors between actual values and predictions. In contrast to lasso regression, ridge regression cannot nullify the impact of an irrelevant feature, which is an effective way to reduce the variance when we have many insignificant features. In other words, use of ridge regression cannot reduce the coefficients to absolute zero (it does not eliminate features but only minimizes them), which means that if we have data with a very large number of features out of which only few are significant for our model, the model risks having poor accuracy.

Elastic Net Combination of L1 and L2 regularization produces the elastic net method of adding a hyperparameter:

$$\text{Elastic Net} = \sum_{i=0}^N \left(y_i - \sum_{j=0}^M x_{ij}w_j \right)^2 + \lambda \left[\alpha \sum_{j=0}^M |w_j| + (1-\alpha) \sum_{j=0}^M w_j^2 \right]$$

where the second component is the penalty function of the elastic net regression. If $\alpha = 0$, we can recognize ridge regression; if $\alpha = 1$, we can recognize lasso regression. Similar to L1 and L2 regularization, cross-validation can be used to tune the hyperparameter α . The elastic net method balances between lasso, which eliminates features and reduces overfitting, and ridge, which reduces the impact of features that are not significant in predicting target values.

Selecting Features with Regularization Embedded into Machine Learning Algorithms In embedded methods, we can select an algorithm for classification or regression and choose the penalty we wish to apply. Let us say we want to build a model using, for example, a linear support vector classification algorithm (LinearSVC in scikit-learn) using an L1 penalty (lasso for regression tasks and LinearSVC for classification).

Input:

```
import pandas as pd
import numpy as np

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Lasso for Regression tasks, and LinearSVC for classification
from sklearn.linear_model import Lasso
from sklearn.svm import LinearSVC
from sklearn.feature_selection import SelectFromModel

# using LinearSVC with penalty l1.
selection = SelectFromModel(LinearSVC(C=0.05, penalty='l1', dual=False))
selection.fit(X_train, y_train)

# see the selected features.
selected_features = X_train.columns[(selection.get_support())]
```

```

print("Selected Features: \n")
print(selected_features)

# see the deleted features.
removed_features = X_train.columns[(selection.estimator_.coef_ == 0).ravel().tolist()
()]
print("Removed Features: \n")
print(removed_features)

```

Output:

Selected Features:

```

Index(['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11',
       'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20', 'V21',
       'V22', 'V23', 'V24', 'V25', 'V27', 'V28', 'Amount'],
      dtype='object')

```

Removed Features:

```
Index(['V26'], dtype='object')
```

Another example could be to use logistic regression with an L2 penalty.

Input:

```

import pandas as pd
import numpy as np
from matplotlib import pyplot

import pandas as pd
from sklearn.preprocessing import LabelEncoder

breastcancer = '../data/breastcancer.csv'
df = pd.read_csv(breastcancer, delimiter=';')

# Encode the data and drop original column from df + remove id variable
enc = LabelEncoder()
df_encoded = df[['diagnosis']].apply(enc.fit_transform)
df = df.drop(['diagnosis', "id"], axis = 1)

# Concatenate dataframes
df = pd.concat([df, df_encoded], axis=1)

# Divide the data
y = df.loc[:, df.columns == 'diagnosis'].values.ravel()
X = df.loc[:, df.columns != 'diagnosis']

# Splitting the data : training and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

```

```

# Get features name
feature_names = [f"{{i}}" for i in X.columns]

from sklearn.linear_model import LogisticRegression
# define the model
model = LogisticRegression(penalty="l2")
# fit the model
model.fit(X_train, y_train)
# get importance
importance = model.coef_[0]

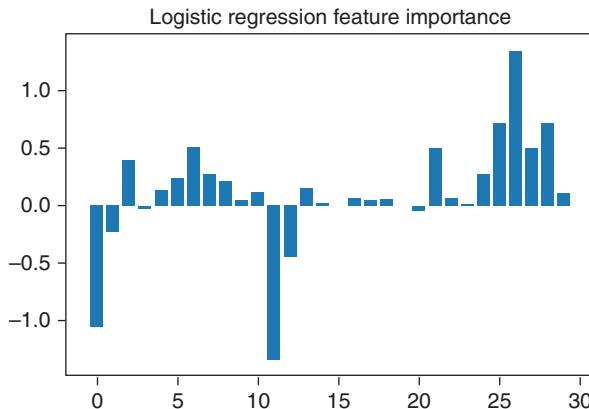
# create a data frame to visualize features importance
final_df = pd.DataFrame({"Features": feature_names, "Importances":importance})
final_df.set_index('Importances')
print(final_df)

# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.title('Logistic Regression Feature Importance')
pyplot.show()

```

Output:

	Features	Importances
0	radius_mean	-1.064278
1	texture_mean	-0.230833
2	perimeter_mean	0.391031
3	area_mean	-0.026048
4	smoothness_mean	0.136829
5	compactness_mean	0.239398
6	concavity_mean	0.515873
7	concave points_mean	0.274763
8	symmetry_mean	0.220043
9	fractal_dimension_mean	0.038494
10	radius_se	0.118969
11	texture_se	-1.348197
12	perimeter_se	-0.457078
13	area_se	0.145482
14	smoothness_se	0.018263
15	compactness_se	-0.005684
16	concavity_se	0.067884
17	concave points_se	0.035053
18	symmetry_se	0.045293
19	fractal_dimension_se	-0.000522
20	radius_worst	-0.042753
21	texture_worst	0.504892
22	perimeter_worst	0.060300
23	area_worst	0.011025
24	smoothness_worst	0.274080
25	compactness_worst	0.718200
26	concavity_worst	1.339619
27	concave points_worst	0.495012
28	symmetry_worst	0.716176
29	fractal_dimension_worst	0.099935



For regression problems, we could also use, for example, ordinary least squares linear regression to fit a linear model with coefficients (w_1, w_2, \dots, w_n) to minimize the residual sum of squares between the predicted and observed targets. Let us analyze in the following example the medical cost personal dataset (<https://www.kaggle.com/mirichoi0218/insurance>), which is used for insurance forecasts, by using linear regression. In the dataset, we will find costs billed by health insurance companies (insurance charges) and features (age, gender, BMI, children, smoking status).

Input:

```

import pandas as pd
from sklearn.preprocessing import LabelEncoder

csv_data = '../data/insurance.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Encode the data and drop original column from df
enc = LabelEncoder()
df_encoded = df[['sex', 'smoker', 'region']].apply(enc.fit_transform)
df = df.drop(['sex', 'smoker', 'region'], axis = 1)

# Concatenate dataframes
df = pd.concat([df, df_encoded], axis=1)

# Divide the data
y = df.loc[:, df.columns == 'charges'].values.ravel()
X = df.loc[:, df.columns != 'charges']

# Splitting the data : training and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

from matplotlib import pyplot
from sklearn.linear_model import LinearRegression

# Get features name
feature_names = [f"{i}" for i in X.columns]

# define the model
model = LinearRegression()

```

```

# fit the model
model.fit(X_train, y_train)
# get importance
importance = model.coef_

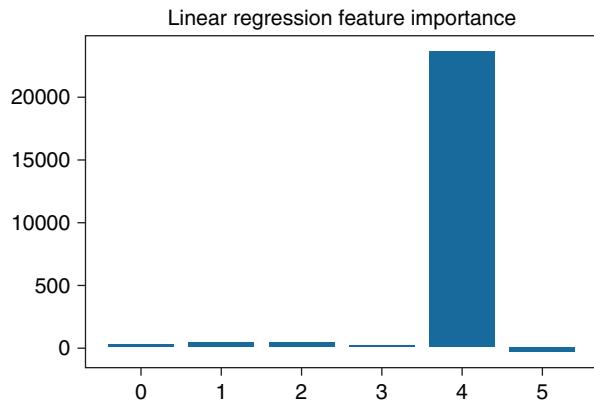
# create a data frame to visualize features importance
final_df = pd.DataFrame({"Features": feature_names, "Importances":importance})
final_df.set_index('Importances')
print(final_df)

# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.title('Linear Regression Feature Importance')
pyplot.show()

```

Output:

	Features	Importances
0	age	261.625690
1	bmi	344.544831
2	children	424.370166
3	sex	109.647196
4	smoker	23620.802521
5	region	-326.462625



Tree-Based Feature Importance Tree-based algorithms such as random forest, XGBoost, decision tree, or extra tree are also commonly used for prediction. They can also be an alternative method to select features by indicating which of them are more important or the most used in making predictions for our target variable (classification). For the example of random forest, a machine learning technique used to solve regression and classification consisting of many decision trees, each tree of the random forest can calculate the importance of a feature. The random forest algorithm can calculate the importance of a feature because of its ability to increase the “pureness” of the leaves. In other words, when we train a tree, feature importance is determined as a decrease in node impurity weighted in a tree (the higher the increment in leaf purity, the more important the feature). We call a situation “pure” when the elements belong to a single class. After a normalization, the sum of the calculated importance scores is 1. The mean decrease impurity that we call the Gini index (between 0 and 1), used by random forest to estimate a feature’s importance, measures the degree or probability that a variable has been wrongly classified when randomly chosen. The index is 0 when all elements belong to a certain class, 1 when the elements are randomly distributed across various classes, and 0.5 when the elements are equally distributed among classes.

The Gini index is calculated as follows:

$$Gini = 1 - \sum_{i=1}^n (p_i)^2$$

where p_i is the probability that an element has been classified in a distinct class.

Let us code a simple example to calculate feature importance using random forest.

Input:

```

import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Get features name
feature_names = [f"{i}" for i in X.columns]

# Model fitting + feature selection altogether
# n_estimators, default=100, is the number of trees in the forest that we can vary
forest = RandomForestClassifier(n_estimators = 100)
forest.fit(X_train, y_train)

# Importance of the resulting features
# Feature importance's are provided by the fitted attribute feature_importances_
# Computed as the mean and standard deviation of accumulation of the impurity decrease
# within each tree.
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)

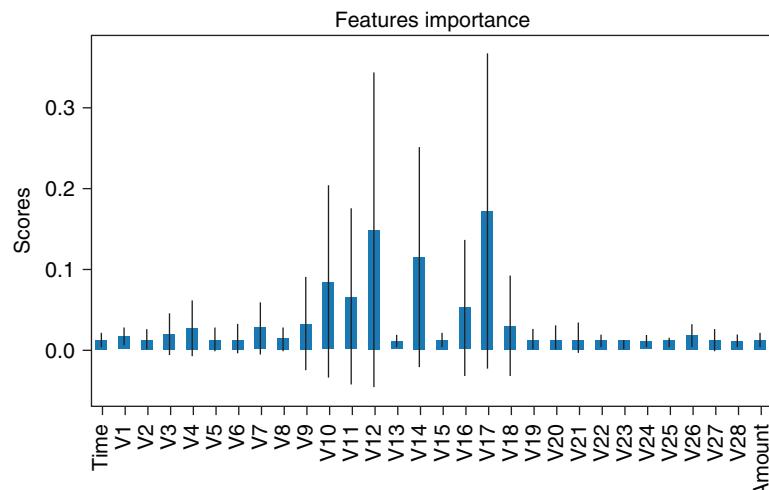
# Create a data frame to visualize features importance
final_df = pd.DataFrame({"Features": X_train.columns, "Importances":importances})
final_df.set_index('Importances')
print(final_df)

# Plot the feature importances in bars.
forest_importances = pd.Series(importances, index=feature_names)
fig, ax = plt.subplots()
forest_importances.plot.bar(yerr=std, ax=ax)
ax.set_title("Feature importances using MDI")
ax.set_ylabel("Mean decrease in impurity")
fig.tight_layout()

```

Output:

	Features	Importances
0	Time	0.011951
1	V1	0.015791
2	V2	0.011951
3	V3	0.019299
4	V4	0.026344
5	V5	0.012594
6	V6	0.014220
7	V7	0.026682
8	V8	0.012779
9	V9	0.031650
10	V10	0.083980
11	V11	0.065454
12	V12	0.147052
13	V13	0.009819
14	V14	0.114699
15	V15	0.011963
16	V16	0.052669
17	V17	0.171931
18	V18	0.029313
19	V19	0.012876
20	V20	0.014186
21	V21	0.015796
22	V22	0.010009
23	V23	0.006625
24	V24	0.010271
25	V25	0.008583
26	V26	0.017892
27	V27	0.011208
28	V28	0.010213
29	Amount	0.012203



Random forest has some limitations. For example, if two features are correlated, they will be given similar and lowered importance. In addition, as a set of decision trees, it gives preference to features with high cardinality.

As stated above, we can perform feature selection with tree-based algorithms such as decision tree (nonparametric, supervised) for both classification and regression. For classification, we can use DecisionTreeClassifier from scikit-learn.

Input:

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

from sklearn.feature_selection import SelectFromModel

# Import data from a csv and create a DataFrame
csv_data = '../data/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()
```

```

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Get features name
feature_names = [f"{i}" for i in X.columns]

from sklearn.tree import DecisionTreeClassifier
# define the model
model = DecisionTreeClassifier()
# fit the model
model.fit(X_train, y_train)
# get importance
importance = model.feature_importances_

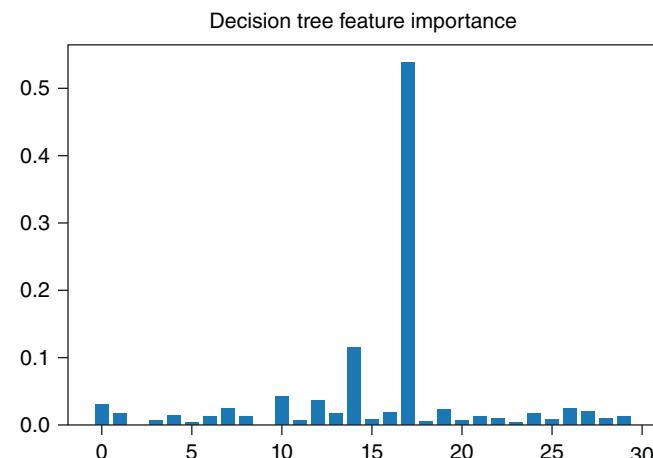
# create a data frame to visualize features importance
final_df = pd.DataFrame({"Features": feature_names, "Importances":importance})
final_df.set_index('Importances')
print(final_df)

# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.title('Decision Tree Feature Importance')
pyplot.show()

```

Output:

	Features	Importances
0	Time	0.028703
1	V1	0.015377
2	V2	0.000000
3	V3	0.004238
4	V4	0.011141
5	V5	0.001695
6	V6	0.009737
7	V7	0.022080
8	V8	0.010224
9	V9	0.000000
10	V10	0.040428
11	V11	0.005098
12	V12	0.036830
13	V13	0.015404
14	V14	0.112317
15	V15	0.007372
16	V16	0.017439
17	V17	0.536157
18	V18	0.001907
19	V19	0.020087
20	V20	0.005494
21	V21	0.009470
22	V22	0.008294
23	V23	0.002796
24	V24	0.014243
25	V25	0.006329
26	V26	0.022797
27	V27	0.016909
28	V28	0.006952
29	Amount	0.010481



For regression, we simply need to replace `DecisionTreeClassifier()` with `DecisionTreeRegressor()`. We can employ many tree-based algorithms for feature selection using both regression, for example, `RandomForestRegressor()`, `GradientBoostingRegressor()`, or `ExtraTreesRegressor()`, and classification, for example, `RandomForestClassifier()`, `GradientBoostingClassifier()`, or `ExtraTreesClassifier()`. It is important to take time to explore which technique will give a model the best performance.

Permutation Feature Importance The idea of permutation feature importance was introduced by Breiman (2001) for random forests. It measures the importance of a feature by computing the increase in the model's prediction error after permuting the values of the feature. If randomly shuffling the values of a feature increases the model error, it means that the feature is "important." By contrast, if shuffling the values of a feature leaves the model error unchanged, the feature is not important because the model has ignored the feature for the prediction. In other words, if we destroy the information contained in a feature by randomly shuffling the feature values, the accuracy of our models should decrease. In addition, if the decrease is substantial, it means that the information contained in the feature is important for our predictions.

Let us say we have trained a model and have measured its quality through MSE, log-loss, or another method. For each feature in the dataset, we randomly shuffle the data in the feature while keeping the values of other features constant. We then generate a new model based on the shuffled values, re-evaluate the quality of the newly trained model, and calculate the feature importance based on the change in the quality of the new model relative to the original one. We perform this process for all features, allowing us to rank all the features in terms of their predictive usefulness.

Let us use permutation feature importance with KNN for classification.

Input:

```
import pandas as pd
import numpy as np

# permutation feature importance with knn for classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.inspection import permutation_importance
from matplotlib import pyplot

# Import data from a csv and create a DataFrame
csv_data = '../data/datasets/creditcard.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Divide the data
y = df.loc[:, df.columns == 'Class'].values.ravel()
X = df.loc[:, df.columns != 'Class']

# Split data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,
random_state=42)

# Get features name
feature_names = [f"{{i}}" for i in X.columns]

# define the model
model = KNeighborsClassifier()
# fit the model
model.fit(X_train, y_train)
# perform permutation importance
```

```

results = permutation_importance(model, X_train, y_train, scoring='accuracy')
# get importance
importance = results.importances_mean

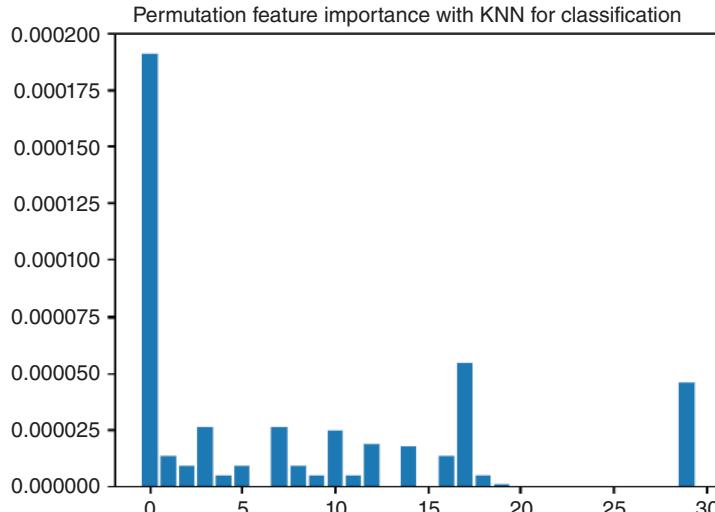
# create a data frame to visualize features importance
final_df = pd.DataFrame({"Features": feature_names, "Importances":importance})
final_df.set_index('Importances')
print(final_df)

# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.title('Permutation Feature Importance with KNN for Classification')
pyplot.savefig('permutation.png')

```

Output:

	Features	Importances
0	Time	1.904804e-04
1	V1	1.316685e-05
2	V2	8.777897e-06
3	V3	2.633369e-05
4	V4	4.388949e-06
5	V5	8.777897e-06
6	V6	0.000000e+00
7	V7	2.633369e-05
8	V8	8.777897e-06
9	V9	4.388949e-06
10	V10	2.457811e-05
11	V11	4.388949e-06
12	V12	1.843358e-05
13	V13	0.000000e+00
14	V14	1.755579e-05
15	V15	0.000000e+00
16	V16	1.316685e-05
17	V17	5.442296e-05
18	V18	4.388949e-06
19	V19	8.777897e-07
20	V20	0.000000e+00
21	V21	0.000000e+00
22	V22	0.000000e+00
23	V23	0.000000e+00
24	V24	0.000000e+00
25	V25	0.000000e+00
26	V26	0.000000e+00
27	V27	0.000000e+00
28	V28	0.000000e+00
29	Amount	4.564507e-05



We can also use permutation feature importance with KNN for regression using the same lines of code and replacing `KNeighborsClassifier` with `KNeighborsRegressor`.

Permutation feature importance can be computationally expensive due to the necessary iteration through each predictor. We also need to pay attention to the potential presence of multicollinearity and consider the context of our model, as scores can be relative.

As we have seen in this entire chapter, there are many ways to select features. Only a few of the large number of methods in the literature have been described here. In summary, filter methods do not incorporate a specific machine learning algorithm and are much faster compared to wrapper methods and less prone to overfitting. Wrapper methods evaluate based on a specific machine learning algorithm to determine the most important features; their drawbacks are the computation time needed and the high chances of overfitting. In embedded methods, feature selection is performed by observing each iteration of a model's training phase; they are effective in reducing overfitting through penalization techniques and represent a good trade-off regarding computation time.

Feature selection can also be performed by managing features as hyperparameters to fine-tune. In this case, we can create a pipeline object to assemble the data transformation and apply an estimator. As we have seen, selection of the k -best variables according to a given correlation metric can be performed by the *SelectKBest* module of scikit-learn. We can combine this module with a supervised model in a *Pipeline* object. *GridSearchCV* can then be used to perform tuning of the hyperparameter considering k as a hyperparameter of the pipeline.

2.5.2.4 Feature Importance Using Graphics Processing Units (GPUs)

If one were to test all the scripts above, they would certainly experience that some of them are computationally time consuming. Sometimes, if we have a large amount of data and we use pandas and sklearn, the computational burden can be quite heavy as these packages do not support GPUs. GPUs are an excellent way to improve computing time. Even if they are often dedicated to the training of neural networks, thanks to frameworks such as TensorFlow, Keras, and PyTorch, we can also consider using them to measure feature importance.

To execute pipelines on GPUs, we can use *RAPIDS*, which is a suite of packages developed by NVIDIA. In RAPIDS, we can find cuDF, which is a data frame manipulation library similar to pandas, and cuML, which is a collection of GPU-accelerated machine learning libraries (providing GPU versions of machine learning algorithms available in scikit-learn). We can also find cuGRAPH, which contains a collection of libraries for graph analytics. When using RAPIDS, we will find many similarities with pandas or scikit-learn.

In pandas, we can read a .csv file as follows:

```
x = pd.read_csv('data.csv')
```

In cuDF, it is quite similar:

```
x_cudf = cudf.read_csv('data.csv')
```

For a linear regression, we would use a script as follows with scikit-learn:

```
reg_sk = sklGLM.LinearRegression(fit_intercept=fit_intercept, normalize=normalize)
result_sk = reg_sk.fit(x, y)
```

In cuML, it is a similar approach:

```
reg_cuml = cumlOLS(fit_intercept=fit_intercept, normalize=normalize,
algorithm=algorithm)
result_cuml = reg_cuml.fit(x_cudf, y_cudf)
```

Another way to leverage GPUs is to use Keras and TensorFlow, in which it is very straightforward to incorporate a regularization such as L2 or L1:

```
tf.keras.layers.Dense(32, kernel_regularizer='l2')
```

The value of bias can also be indicated, as in this example:

```
tf.keras.layers.Dense(32, kernel_regularizer=l2(0.01),
bias_regularizer=l2(0.01))
```

The *kernel_regularizer* will apply a penalty on the kernel of the layer, and the *bias_regularizer* will apply a bias penalty to the layer. It is also possible to use L1 and L2 at the same time by adding “*kernel_regularizer=l1_l2(l1=0.01, l2=0.01)*.”

We can also explore PyTorch in applying regularization. For example, it is possible to choose the α value and sum the weights squared:

```
l2_alpha = 0.001
l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
```

The bias can also be added after calculating the loss function:

```
loss = loss + l2_alpha * l2_norm
```

We will explore GPUs further in the description of machine and deep learning algorithms, as they are crucial in several use cases to accelerate the computing time.

2.5.2.5 Feature Selection Using HephaIstos

HephaIstos also provides feature selection techniques. We can go to the hephaIstos main folder and create a new Python file to create a pipeline. The feature_selection parameter should be included in the pipeline:

- **feature_selection:** Here, we can select a feature selection method (filter, wrapper, or embedded):

- Filter options:
 - **variance_threshold:** Apply a variance threshold. If we choose this option, we also need to indicate the features we want to process (features_to_process= ['feature_1', 'feature_2', ...]) and the threshold (var_threshold=0 or any number).
 - **chi2:** Perform a chi-squared test on the samples and retrieve only the k-best features. We can define k with the k_features parameter.
 - **anova_f_c:** Create a SelectKBest object to select features with the k-best ANOVA F-values for classification. We can define k with the k_features parameter.
 - **anova_f_r:** Create a SelectKBest object to select features with the k-best ANOVA F-values for regression. We can define k with the k_features parameter.
 - **pearson:** The main idea for feature selection is to retain the variables that are highly correlated with the target and keep features that are uncorrelated among themselves. The Pearson correlation coefficient between features is defined by cc_features, and that between features and the target is defined by cc_target.

Here are short examples:

```
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', feature_selection = 'pearson', cc_features
= 0.7, cc_target = 0.7)
```

or

```
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', feature_selection = anova_f_c, k_features
= 2)
```

- Wrapper methods: The following options are available for feature_selection: “forward_stepwise,” “backward_elimination,” and “exhaustive.”

- **wrapper_classifier:** In wrapper methods, we need to select a classifier or regressor. We can choose one from scikit-learn, such as KneighborsClassifier(), RandomForestClassifier, LinearRegression, or others, and apply it to forward stepwise (forward_stepwise), backward elimination (backward_elimination), or exhaustive (exhaustive) methods.
- **min_features and max_features:** These are attributes for exhaustive to specify the minimum and maximum number of features desired in the combination.

Here is a short example:

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

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', feature_selection =
'backward_elimination', wrapper_classifier = KNeighborsClassifier())
```

- Embedded methods:
 - **feature_selection:** We can select from several methods.
- **lasso:** If we choose lasso, we need to add the alpha parameter (lasso_alpha).
- **feat_reg_ml:** Select features with regularization embedded into machine learning algorithms. We need to select the machine learning algorithms (in scikit-learn) by setting the parameter ml_penalty:
 - embedded_linear_regression
 - embedded_logistic_regression
 - embedded_decision_tree_regressor
 - embedded_decision_tree_classifier
 - embedded_random_forest_regressor
 - embedded_random_forest_classifier
 - embedded_permutation_regression
 - embedded_permutation_classification
 - embedded_xgboost_regression
 - embedded_xgboost_classification

Here is a short example:

```
from ml_pipeline_function import ml_pipeline_function
from sklearn.svm import LinearSVC

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', feature_selection = 'feat_reg_ml',
ml_penalty = LinearSVC(C=0.05, penalty='l1', dual=False, max_iter = 5000))
```

Further Reading

- Aeberhard, S., Coomans, D., and de Vel, O. (1992). Comparison of classifiers in high dimensional settings. *Technical Report no. 92-01*. Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to *Technometrics*).
- Aeberhard, S., Coomans, D., and de Vel, O. (1992). The classification performance of RDA. *Technical Report no. 92-01*. Dept. of Computer Science and Dept. of Mathematics and Statistics, James Cook University of North Queensland. (Also submitted to *Journal of Chemometrics*).
- Alkharusi, H. (2012). Categorical variables in regression analysis: a comparison of dummy and effect coding. *International Journal of Education* 4: 202–210. <https://doi.org/10.5296/ije.v4i2.1962>.
- Azur, M.J., Stuart, E.A., Frangakis, C., and Leaf, P.J. (2011). Multiple imputation by chained equations: what is it and how does it work? *International Journal of Methods in Psychiatric Research* 20 (1): 40–49. <https://doi.org/10.1002/mpr.329>.
- Belkin, M. and Niyogi, P. (2001). Laplacian eigenmaps and spectral techniques for embedding and clustering. *Advances in Neural Information Processing Systems* 14: 585–591.
- Bengio, Y. and Monperrus, M. (2004). Non-local manifold tangent learning. In: *Proceedings of the 17th International Conference on Neural Information Processing Systems (NIPS'04)*, vol. 17, pp. 129–136. MIT Press.
- Bingham, E. and Mannila, H. (2001). Random projection in dimensionality reduction: applications to image and text data. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '01)*, New York, NY, USA, pp. 245–250. ACM.
- Birjandtalab, J., Pouyan, M.B., and Nourani, M. (2016). Nonlinear dimension reduction for EEG-based epileptic seizure detection. In: *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, Las Vegas, NV, USA, pp. 595–598. <https://doi.org/10.1109/BHI.2016.7455968>.
- Box, G.E.P. and Cox, D.R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society B* 26: 211–252.
- Brand, M. (2002). Charting a manifold. In: *Proceedings of the 15th International Conference on Neural Information Processing Systems (NIPS'02)*, vol. 15, pp. 961–968. MIT Press.
- Breiman, L. (2001). Random forests. *Machine Learning* 45 (1): 5–32. <https://doi.org/10.1023/A:1010933404324>.
- Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth.
- van Buuren, S. (2007). Multiple imputation of discrete and continuous data by fully conditional specification. *Statistical Methods in Medical Research* 16: 219–242. <https://doi.org/10.1177/0962280206074463>.
- van Buuren, S. and Groothuis-Oudshoorn, K. (2011). mice: Multivariate imputation by chained equations in R. *Journal of Statistical Software* 45: 1–67.
- Carey, G. (2003). Coding categorical variables. http://ibgwww.colorado.edu/~carey/p5741ndir/Coding_Categorical_Variables.pdf.
- Cestnik, B. and Bratko, I. (1991). On estimating probabilities in tree pruning. In: *Machine Learning — EWSL-91. EWSL 1991, Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence)*, vol. 482 (ed. Y. Kodratoff), pp. 138–150. Springer. <https://doi.org/10.1007/BFb0017010>.
- Cohen, D. (1972). Magnetoencephalography: detection of the brain's electrical activity with a superconducting magnetometer. *Science* 175: 664–666.
- Cowell, R.G., Dawid, A.P., Lauritzen, S.L., and Spiegelhalter, D.J. (1999). *Probabilistic Networks and Expert Systems*. Springer.
- Daniel, W.W. and Cross, C.L. (2018). *Biostatistics: A Foundation for Analysis in the Health Sciences*. Wiley.
- Dasarathy, B.V. (1980). Nosing around the neighborhood: a new system structure and classification rule for recognition in partially exposed environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-2, 1: 67–71.
- Dasgupta, S. (2000). Experiments with random projection. In: *Proceedings of the Sixteenth conference on Uncertainty in Artificial Intelligence (UAI'00)* (ed. C. Boutilier and M. Goldszmidt), 143–151. Morgan Kaufmann Publishers Inc.
- Deng, X., Li, Y., Weng, J., and Zhang, J. (2019). Feature selection for text classification: a review. *Multimedia Tools and Applications* 78, 3 (Feb 2019): 3797–3816. <https://doi.org/10.1007/s11042-018-6083-5>.
- Donoho, D. and Grimes, C. (2003). Hessian eigenmaps: locally linear embedding techniques for high dimensional data. *Proceedings of National Academy of Sciences of the United States of America* 100 (10): 5591–5596.
- Duda, R.O. and Hart, P.E. (1973). *Pattern Classification and Scene Analysis* (Q327.D83). Wiley.
- Durrett, R. (1996). *Probability: Theory and Examples*, 2e, 62. Duxbury Press.
- Dy, J.G. and Brodley, C.E. (2004). Feature selection for unsupervised learning. *Journal of Machine Learning Research* 5: 845–889.
- Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*, 3e, vol. 1.
- Ferri, F.J., Pudil, P., Hatef, M., and Kittler, J. (1994). Comparative study of techniques for large-scale feature selection. In: *Machine Intelligence and Pattern Recognition*, vol. 16 (ed. E.S. Gelsema and L.S. Kanal), pp. 403–413. North-Holland.

- Fisher, R.A. (1936). The use of multiple measurements in taxonomic problems. *Annual Eugenics* 7 (Part II): 179–188; also in *Contributions to Mathematical Statistics* (Wiley, NY, 1950).
- Florescu, I. (2014). *Probability and Stochastic Processes*. Wiley.
- Friedman, J. (2001). Greedy function approximation: a gradient boosting machine. *The Annals of Statistics* 29 (5): 1189–1232. <https://doi.org/10.1214/aos/1013203451>.
- Friedman, J. (2002). Stochastic gradient boosting. *Computational Statistics and Data Analysis* 38 (4): 367–378. [https://doi.org/10.1016/S0167-9473\(01\)00065-2](https://doi.org/10.1016/S0167-9473(01)00065-2).
- Gallager, R.G. (2013). *Stochastic Processes Theory for Applications*. Cambridge University Press.
- Gashler, M., Ventura, D., and Martinez, T. (2008). Iterative non-linear dimensionality reduction with manifold sculpting. In: *Advances in Neural Information Processing Systems*, vol. 20 (ed. J.C. Platt, D. Koller, Y. Singer, and S. Roweis), pp. 513–520. MIT Press.
- Gates, G.W. (1972). The reduced nearest neighbor rule. *IEEE Transactions on Information Theory* 1972: 431–433.
- Gelman, A. and Hill, J. (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models (Analytical Methods for Social Research)*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511790942>.
- George, G. (2004). Testing for the independence of three events. *Mathematical Gazette* 88: 568.
- Grus, J. (2015). *Data Science from Scratch*. O'Reilly.
- Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research* 3: 1157–1182.
- Hamel, P. and Eck, D. (2010). Learning features from music audio with deep belief networks. In: *Proceedings of the 11th International Society for Music Information Retrieval Conference, ISMIR*, Utrecht, Netherlands, pp. 339–344.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *Elements of Statistical Learning*, 2e. Springer.
- Hong, S. and Lynn, H.S. (2020). Accuracy of random-forest-based imputation of missing data in the presence of non-normality, non-linearity, and interaction. *BMC Medical Research Methodology* 20: 199. <https://doi.org/10.1186/s12874-020-01080-1>.
- Hwei, P. (1997). *Theory and Problems of Probability, Random Variables, and Random Processes*. McGraw-Hill. ISBN: 0-07-030644-3.
- Hyvärinen, A. and Oja, E. (2000). Independent component analysis: algorithms and applications. *Neural Network* 13 (4–5): 411–430. [https://doi.org/10.1016/s0893-6080\(00\)00026-5](https://doi.org/10.1016/s0893-6080(00)00026-5).
- Hyvärinen, A., Karhunen, J., and Oja, E. (2001). *Independent Component Analysis*. Wiley.
- Ioffe, S.; Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 448–456.
- Ipsen, N., Mattei, P., and Frellsen, J. (2022). How to deal with missing data in supervised deep learning? In: *Artemiss - ICML Workshop on the Art of Learning with Missing Values*, 1–30. Vienne, Austria. hal-03044144. <https://openreview.net/pdf?id=J7b4BCtDm4>.
- Jamieson, A.R., Giger, M.L., Drukker, K. et al. (2010). Exploring nonlinear feature space dimension reduction and data representation in breast CADx with Laplacian Eigenmaps and t-SNE. *Medical Physics* 37 (1): 339–351. <https://doi.org/10.1111/1.3267037>.
- Junn, J. and Masuoka, N. (2020) Replication data for: the gender gap is a race gap: women voters in U.S. Presidential Elections. Harvard Dataverse, V1. <https://doi.org/10.7910/DVN/XQYJKN>.
- Juszczak, P., Tax, D.M.J., and Dui, R.P.W. (2002). Feature scaling in support vector data description. In: *Proceedings of the ASCI 2002 8th Annual Conference of the Advanced School for Computing and Imaging. Citeseer*, pp. 95–102. https://www.researchgate.net/publication/2535451_Feature_Scaling_in_Support_Vector_Data_Description.
- Kohavi, R. and John, G.H. (1997). Wrappers for feature subset selection. *Artificial Intelligence* 97 (1–2): 273–324.
- Lapidoth, A. (2017). *A Foundation in Digital Communication*. Cambridge University Press.
- Levina, E. and Bickel, P.J. (2004). Maximum likelihood estimation of intrinsic dimension. In: *Proceedings of the 17th International Conference on Neural Information Processing Systems (NIPS'04)*. pp. 777–784. MIT Press.
- Li, T., Zhu, S., and Ogihara, M. (2006). Using discriminant analysis for multi-class classification: an experimental investigation. *Knowledge and Information Systems* 10 (4): 453–472.
- Little, R.J.A. and Rubin, D.B. (1986). *Statistical Analysis with Missing Data*. Wiley.
- Liu, H. and Motoda, H. (1998). *Feature Selection for Knowledge Discovery & Data Mining*. Kluwer Academic Publishers.
- Liu, H. and Motoda, H. (ed.) (2007). *Computational Methods of Feature Selection*. Chapman and Hall/CRC Press.
- Liu, H. and Yu, L. (2005). Toward integrating feature selection algorithms for classification and clustering. *IEEE Transactions on Knowledge and Data Engineering* 17 (3): 1–12.

- Ma, S. and Huang, J. (2008). Penalized feature selection and classification in bioinformatics. *Briefings in Bioinformatics* 9 (5): 392–403.
- van der Maaten, L.J.P. (2009). Learning a parametric embedding by preserving local structure. In: *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, Clearwater Beach, Florida, USA, PMLR* 5: 384–391.
- van der Maaten, L.J.P. (2014). Accelerating t-SNE using tree-based algorithms. *Journal of Machine Learning Research* 15 (October): 3221–3245.
- van der Maaten, L.J.P. and Hinton, G.E. (2008). Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research* 9 (November): 2579–2605.
- van der Maaten, L.J.P. and Hinton, G.E. (2012). Visualizing non-metric similarities in multiple maps. *Machine Learning* 87 (1): 33–55.
- Martinez, M. and Kak, A.C. (2001). PCA versus LDA. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23 (2): 228–233. <https://doi.org/10.1109/34.908974>.
- Micci-Barreca, D. (2001). A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *SIGKDD Explorations Newsletter* 3: 1. <http://dx.doi.org/10.1145/507533.507538>.
- Papoulis, A. (1991). *Probability, Random Variables and Stochastic Processes*. McGraw Hill.
- Park, K. (2018). *Fundamentals of Probability and Stochastic Processes with Applications to Communications*. Springer.
- Parson, L., Haque, E., and Liu, H. (2004). Subspace clustering for high dimensional data – a review. *ACM SIGKDD Explorations Newsletter Archive* special issue on learning from imbalanced datasets 6 (1): 90–105. 1931–0145.
- Pedregosa, F., Grisel, O., Blondel, M., et al. (2011). Manifold learning on handwritten digits: locally linear embedding, Isomap. License: BSD 3 clause (C) INRIA 2011, Online scikit-learn documentation, Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825–2830. https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html.
- van der Plas. (2016). *Python Data Science Handbook*. O'Reilly Media, Inc. 9781491912058
- Pudil, P., Novovičová, J., and Kittler, J. (1994). Floating search methods in feature selection. *Pattern Recognition Letters* 15 (11): 1119–1125.
- Radhakrishna Rao, C. (1948). The utilization of multiple measurements in problems of biological classification. *Journal of the Royal Statistical Society, Series B (Methodological)* 10 (2): 159–203.
- Raghunathan, T.W., Lepkowksi, J.M., Van Hoewyk, J., and Solenberger, P. (2001). A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey Methodology* 27: 85–95.
- ResearchGate. Iterative non-linear dimensionality reduction with manifold sculpting. https://www.researchgate.net/publication/220270207_Iterative_Non-linear_Dimensionality_Reduction_with_Manifold_Sculpting.
- Robnik-Sikonja, M. and Kononenko, I. (2003). Theoretical and empirical analysis of relief and relief. *Machine Learning* 53: 23–69.
- Roweis, S.T. (1997). Em algorithms for PCA and SPCA. In: *Advances in Neural Information Processing Systems*, vol. 10 (ed. M.I. Jordan, M.J. Kearns, and S.A. Solla), pp. 626–632. <https://api.semanticscholar.org/CorpusID:1939401>.
- Roweis, S.T. and Saul, L.K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science* 290.
- Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Saporta, G. (2006). *Probabilités, analyse des données et statistique*, Technip Éditions, p. 622 (ISBN 2-7108-0565-0). <https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be>.
- Saul, L.K. and Roweis, S.T. (2003). Think globally, fit locally: unsupervised learning of low dimensional manifolds. *Journal of Machine Learning Research* 4: 119–155.
- Schafer, J.L. (1999). Multiple imputation: a primer. *Statistical Methods in Medical Research* 8 (1): 3–15.
- Schölkopf, B., Smola, A.J., and Müller, K.-R. (1999). Kernel principal component analysis. In: *Advances in Kernel Methods: Support Vector Learning*, 327–352. MIT Press.
- Shah, A.D., Bartlett, J.W., Carpenter, J. et al. (2014). Comparison of random forest and parametric imputation models for imputing missing data using MICE: a CALIBER study. *American Journal of Epidemiology* 179 (6): 764–774. <https://doi.org/10.1093/aje/kwt312>.
- Shanker, M., Hu, M.Y., and Hung, M.S. (1996). Effect of data standardization on neural network, training. *Omega* 24: 385–397. <https://www.sciencedirect.com/science/article/pii/0305048396000102>.
- de Silva, V. and Tenenbaum, J. B. (2002). Global versus local methods in nonlinear dimensionality reduction. In: *Proceedings of the 15th International Conference on Neural Information Processing Systems (NIPS'02)*. pp. 721–728. MIT Press.
- Singhi, S., and Liu, H. (2006). Feature subset selection bias for classification learning. In: *Proceedings of the 23rd international conference on Machine learning (ICML '06)*. Association for Computing Machinery, New York, NY, USA, pp. 849–856. <https://doi.org/10.1145/1143844.1143951>.

- Su, Y.S., Gelman, A., Hill, J., and Yajima, M. (2009). Multiple imputation with diagnostics (mi) in R: opening windows into the black box. *Journal of Statistical Software* 45: 1–31.
- Sumithra, V. and Surendran, S. (2015). A review of various linear and non linear dimensionality reduction techniques. *International Journal of Computer Science and Information Technologies* 6 (3): 2354–2360.
- Tenenbaum, J.B., de Silva, V., and Langford, J.C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science* 290: 2319–2323.
- Vincent, P. and Bengio, Y. (2002). Manifold parzen windows. In: *Proceedings of the 15th International Conference on Neural Information Processing Systems (NIPS'02)*, vol. 15, pp. 825–832. MIT Press.
- Wallach, I. and Liljean, R. (2009). The protein-small-molecule database, a non-redundant structural resource for the analysis of protein-ligand binding. *Bioinformatics* 25 (5): 615620. PMID 19153135. <https://doi.org/10.1093/bioinformatics/btp035>.
- Wang, J. (2012). *Geometric Structure of High-Dimensional Data and Dimensionality Reduction*. Springer. <https://doi.org/10.1007/978-3-642-27497-8>.
- Weinberger, K., Dasgupta, A., Langford, J. et al. (2009). Feature hashing for large scale multitask learning. In: *Proceedings of the 26th Annual International Conference on Machine Learning (ICML June, 14th, 2009)*. Association for Computing Machinery, New York, NY, USA, 1113–1120. <https://doi.org/10.1145/1553374.1553516>.
- Weisberg S. (2001). Yeo-Johnson power transformations. www.stat.umn.edu/arc/ (accessed 26 October 2001).
- Wu, Y.N. (2014). Statistical independence. In: *Computer Vision* (ed. K. Ikeuchi). Springer. https://doi.org/10.1007/978-0-387-31439-6_744.
- Yeo, I.K. and Johnson, R.A. (2000). A new family of power transformations to improve normality or symmetry. *Biometrika* 87 (4): 954–959.
- Yu, L. and Liu, H. (2004). Efficient feature selection via analysis of relevance and redundancy. *Journal of Machine Learning Research* 5 (October): 1205–1224.
- Yu, K., Guo, X., and Liu, L., et al. (2020). Causality-based feature selection: methods and evaluations. *ACM Computing Surveys* 53, 5, Article 111 (September 2021). <https://doi.org/10.1145/3409382>.
- Zhang, Z. and Zha, H. (2006). A domain decomposition method for fast manifold learning. In: *Advances in Neural Information Processing Systems*, vol. 18 (ed. Y. Weiss, B. Schölkopf, and J. Platt). MIT Press.
- Zhao, Z. and Liu, H. (2007a). Searching for interacting features. *Conference: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, Hyderabad, India (6–12 January 2007).
- Zhao, Z. and Liu, H. (2007b). Semi-supervised feature selection via spectral analysis. *SDM*.
- Feature extraction (audio, video, text) <https://www.mathworks.com/discovery/feature-extraction.html>
<https://www.kaggle.com/c/caterpillar-tube-pricing/discussion/15748#143154>
http://contrib.scikit-learn.org/category_encoders/jamesstein.html
<http://genet.univ-tours.fr/gen002200/bibliographie/Bouquins%20INRA/Biblio/Independent%20component%20analysis%20A%20tutorial.pdf>
<http://psych.colorado.edu/~carey/Courses/PSYC5741/handouts/Coding%20Categorical%20Variables%202006-03-03.pdf>
<http://surfer.nmr.mgh.harvard.edu/fswiki>
http://usir.salford.ac.uk/id/eprint/52074/1/AI_Com_LDA_Tarek.pdf
<https://analyticsindiamag.com/5-ways-handle-missing-values-machine-learning-datasets/>
https://bib.irb.hr/datoteka/763354.MIPRO_2015_JovicBrkicBogunovic.pdf
https://contrib.scikit-learn.org/category_encoders/index.html
<https://cran.r-project.org/web/packages/miceRanger/vignettes/miceAlgorithm.html>
<https://cs.nyu.edu/~roweis/lle/papers/lleintro.pdf>
<https://datascienceplus.com/understanding-the-covariance-matrix/>
<https://docs.rapids.ai/api>
https://en.wikipedia.org/wiki/Decision_tree_learning
<https://inside-machinelearning.com/regularization-deep-learning/>
<https://machinelearningmastery.com/basic-feature-engineering-time-series-data-python/>
<https://machinelearningmastery.com/power-transforms-with-scikit-learn/>
<https://medium.com/analytics-vidhya/linear-discriminant-analysis-explained-in-under-4-minutes-e558e962c877>
<https://medium.com/rapids-ai/accelerating-random-forests-up-to-45x-using-cuml-dfb782a31bea>
https://miro.medium.com/max/2100/0*NBVi7M3sGyiUSyd5.png
<https://nycdatascience.com/blog/meetup/featured-talk-1-kaggle-data-scientist-owen-zhang/>

<https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html>
https://scikit-learn.org/dev/modules/lda_qda.html
<https://scikit-learn.org/stable/>
https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_vs_lda.html#sphx-glr-auto-examples-decomposition-plot-pca-vs-lda-py
https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.GenericUnivariateSelect.html#sklearn.feature_selection.GenericUnivariateSelect
<https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>
<https://scikit-learn.org/stable/modules/impute.html#impute>
https://sebastianraschka.com/Articles/2014_python_lda.html
<https://stats.oarc.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/>
<https://towardsdatascience.com/7-ways-to-handle-missing-values-in-machine-learning-1a6326adf79e>
<https://towardsdatascience.com/all-about-categorical-variable-encoding-305f3361fd02>
<https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35>
<https://towardsdatascience.com/box-cox-transformation-explained-51d745e34203>
<https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be>
<https://towardsdatascience.com/preprocessing-with-sklearn-a-complete-and-comprehensive-guide-670cb98fcfb9>
<https://towardsdatascience.com/top-4-time-series-feature-engineering-lessons-from-kaggle-ca2d4c9cbbe7>
<https://towardsdatascience.com/types-of-transformations-for-better-normal-distribution-61c22668d3b9>
<https://towardsdatascience.com/understand-data-normalization-in-machine-learning-8ff3062101f0>
<https://www.analyticsvidhya.com/blog/2019/12/6-powerful-feature-engineering-techniques-time-series/>
<https://www.analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/>
<https://www.datacamp.com/community/tutorials/categorical-data>
<https://www.kaggle.com/code/louise2001/rapids-feature-importance-is-all-you-need/notebook>
<https://www.kaggle.com/davidbn92/weight-of-evidence-encoding>
<https://www.kaggle.com/pmarcelino/data-analysis-and-feature-extraction-with-python>
<https://www.kaggle.com/prashant111/comprehensive-guide-on-feature-selection>
<https://www.kaggle.com/subinium/11-categorical-encoders-and-benchmark>
<https://www.kaggle.com/sumanthvrao/daily-climate-time-series-data>
<https://www.mygreatlearning.com/blog/label-encoding-in-python/#labelencoding>
<https://www.statsmodels.org/dev/contrasts.html>

3

Machine Learning Algorithms



Photo by Annamária Borsos

In the literature, we can find many machine learning algorithms that can be used for different tasks, including simple linear regression for prediction problems, decision trees, naïve Bayes classifiers, random forests, neural networks, and support vector machines (SVMs). In this chapter, we will study some of the most commonly used machine learning algorithms along with their fundamental math, use cases, and coding using Python and the various libraries presented in the first chapter of this book. We have already encountered the concepts of supervised, unsupervised, and reinforcement learning. If we probe a bit more, we can categorize the algorithms based on their underlying mathematical model: regression, clustering, Bayesian, neural network, ensemble, regularization, rule system, dimensionality reduction, or decision tree. We have already seen some algorithms and their respective categories.

As its name indicates, Bayesian machine learning models are based on Bayes' theorem. It means that machine learning models are based on the calculation of probability, for instance, the probability that Cristiano Ronaldo will score three goals knowing that he scored two in his last match. Regression involves finding a relationship between variables in our data. This is based on geometry as we try to find the line with best slope that can be fit into our data and minimize error. We can then use this line to output our prediction values. The objective of clustering is to divide our data points into several groups. The groups bring together the data points that are more like other data points in the same group than those in other groups. Artificial neural networks are inspired by the human brain and mimic the way biological neurons communicate. Neural networks are composed of node layers (artificial neurons) with an input layer, hidden layers, and an output layer. Each node has a weight and a threshold and is connected to another. If the output of an individual node is above the threshold value (specified by the user), the node is activated and sends data to the next layer of the network. Previously, we have seen regularization used in conjunction with classification or regression algorithms that penalize features that do not contribute to

the model given a coefficient threshold. Ensemble methods use multiple learning algorithms to obtain better performance. Rule-based machine learning algorithms run with predefined sets of rules, created by the user.

Let us explore some popular algorithms.

3.1 Linear Regression

3.1.1 The Math

Linear regression is one of the most simple and popular algorithms in which the output is in a continuous range, for example, salary, age, weight, or height, and is not classified into categories. The goal is to show a linear relationship between a single dependent variable y and one or multiple independent variables x (simple/univariate linear regression or multiple linear regression, respectively) by fitting a line between them called the regression line. A linear regression model uses a straight line to fit the model. With linear regression, we can predict continuous values with a constant slope.

Let us say we want to predict the weight of something according to the height of something or maybe the price of a house (y) according to the size of the house (x). We can imagine that y is linearly dependent on x and that we can create a straight line corresponding to the equation $y = ax + b$:

$$h_{\theta}(x) = y = \theta_0 + \theta_1 x$$

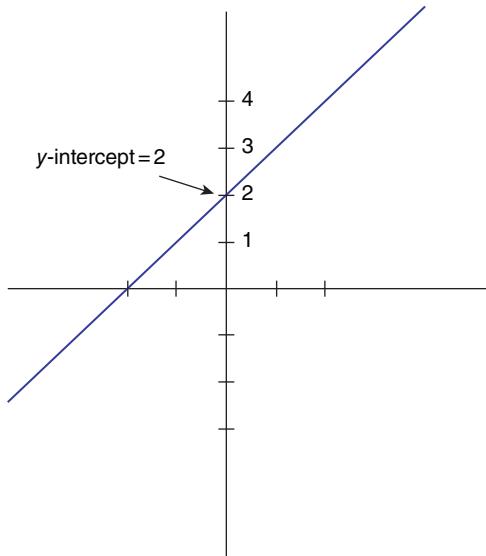
where h_{θ} is the hypothesis function, x is the independent variable, and θ_0 and θ_1 are regression coefficients (the parameters that need to be learned).

For multiple linear regression with several independent features (x) and a single dependent feature (y), the equation would be the following:

$$h_{\theta}(x) = y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

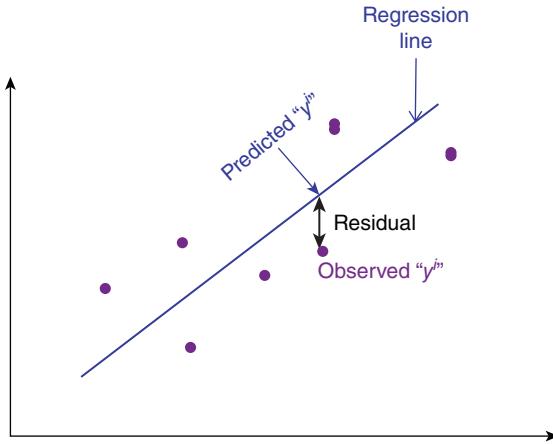
where we consider an $n \times m$ matrix of x and n is the total number of dependent features. It is important to ensure that our features are on the same scale before making any hypothesis.

In the case of a univariate linear regression, the y -intercept is θ_0 and the slope of the line is θ_1 . The slope indicates how much our target variable will change as the independent variable increases or decreases. The y -intercept is wherever the regression line crosses the y -axis.



The question is how we can find the regression coefficient to draw the best-fitting line for linear regression. We can use batch gradient descent, stochastic gradient descent, or normal equation.

First, we need to define a cost function. We call the error between predicted values and observed values “residuals.” Our cost function is the sum of squares of residuals.



We denote the cost function (J), which is the minimum of the root-mean-squared error of the model, obtained by subtracting the predicted values from actual values. The cost function is the minimum of these error values:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

Let us take a simple linear regression:

$$h_\theta(x) = y = \theta_0 + \theta_1 x$$

For each valid data entry in our data, we want to minimize the following function:

$$(h_\theta(x^i) - y^i)^2$$

This function corresponds to the difference between our hypothetical model's predictions and the real values. For every value, we can generalize as follows:

$$\sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

where m is the number of records in our dataset. The objective of the learning algorithm is to find the ideal parameters θ_0 and θ_1 so that $h_\theta(x)$ is close to y for the training examples (x, y) . As expressed above, this is represented by the following mathematical expression that we need to minimize:

$$\frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

where $h_\theta(x^i) = \theta_0 + \theta_1 x^i$, (x^i, y^i) represent the i th training data, m is the number of training examples, and $\frac{1}{2}$ is a constant introduced to help when performing calculations for gradient descent. It produces the cost function, defined as the $\frac{1}{2}$ of the mean squares of $h_\theta(x^i) - y^i$, with the learning objective to minimize it by using, for example, gradient descent:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

3.1.2 Gradient Descent to Optimize the Cost Function

Gradient descent is an iterative first-order optimization algorithm that is used to find a local minimum or maximum of a given function. Gradient descent is commonly used in machine learning and deep learning to minimize the cost function or the error of a model. In linear regression, we can use it to find the optimal θ_0 and θ_1 to minimize the cost function. In this

case, gradient descent will choose random values of θ_0 and θ_1 and iteratively update values of θ_0 and θ_1 until a convergence at which we reach local minima, meaning that the cost function does not decrease further.

Let us review our cost function:

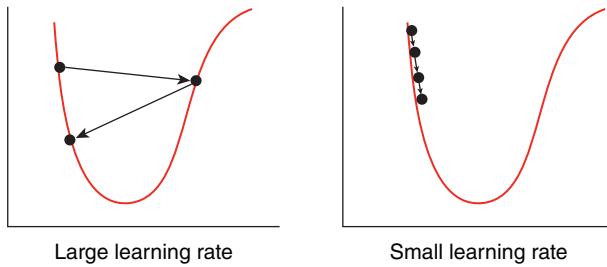
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

where $h_\theta(x^i)$ is a predicted value and y^i is a true value.

In gradient descent, we find the optimal θ values, which we can express as follows:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

where α is the learning rate. The algorithm iteratively calculates the next point using a gradient at the current position then scales it by using the learning rate and subtracts the obtained value from the current position. The smaller the learning rate (maximum iteration), the longer gradient descent converges (reaching the optimum point). On the other hand, if the learning rate is too high, the algorithm has a risk of failing to converge to the optimal point.



The following statements are true:

$$\frac{\partial}{\partial \theta} J_\theta = \frac{\partial}{\partial \theta} \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^m (h_\theta(x^i) - y^i) \cdot \frac{\partial}{\partial \theta_j} (\theta_j x^i - y^i)$$

$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^m [(h_\theta(x^i) - y^i)x^i]$$

Therefore, the following is true:

$$\theta_j = \theta_j - \frac{\alpha}{m} \sum_{i=1}^m [(h_\theta(x^i) - y^i)x^i]$$

where θ_j is the weight of the hypothesis, $h_\theta(x^i)$ is the predicted y value for the i th input, j is the feature index number, and α is the learning rate.

To illustrate the gradient descent process, let us take a simple (univariate) example with a quadratic function:

$$f(x) = x^2 + 2x + 1$$

The gradient function is the following:

$$\frac{df(x)}{dx} = 2x + 2$$

Let us take a learning rate of 0.1 and a starting point at $x = 10$. If we calculate the first two steps, we obtain the following:

$$x_1 = 10 - 0.1 \cdot (2 \cdot 10 + 2) = 7.8$$

$$f(x_1) = 100 + 20 + 1 = 121$$

$$x_2 = 7.8 - 0.1 \cdot (2 \cdot 7.8 + 2) = 6.04$$

$$f(x_2) = 7.8 \cdot 7.8 + 2 \cdot 7.8 + 1 = 77.44$$

Let us now implement a gradient descent in Python to find the local minimum of our function.

Input:

```
from scipy import misc

import matplotlib.pyplot as plt
import numpy as np

# Define a simple (univariate) and concrete example with a quadratic function:
def fonction(x):
    return x**2+2*x+1

# Scaling for plotting
x = np.arange(-10.0, 10.0, 0.01)
y = fonction(x)

plt.plot(x, y, 'r-')

alpha = 0.1 # learning rate
nb_max_iter = 100 # Nb max d'iteration
eps = 0.0001 # stop condition

x0 = 10 # start point
y0 = fonction(x0)
plt.plot(x0, fonction(x0))

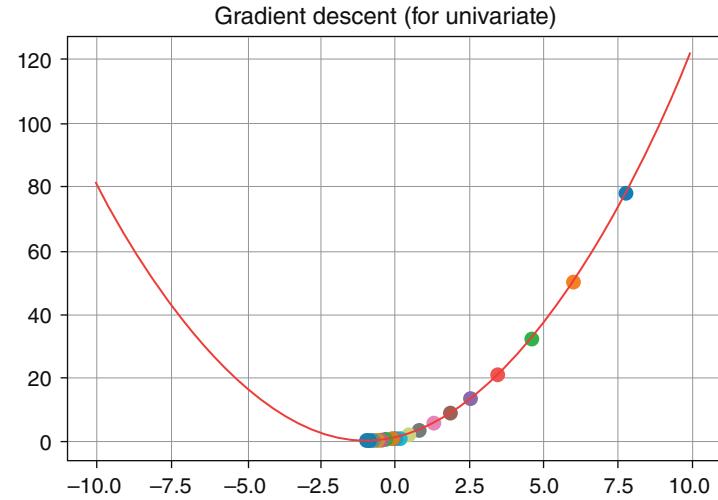
cond = eps + 10.0 # start with cond greater than eps (assumption)
nb_iter = 0
tmp_y = y0

while cond > eps and nb_iter < nb_max_iter:
    # Gradient Descent
    x0 = x0 - alpha * misc.derivative(fonction, x0)
    y0 = fonction(x0)
    nb_iter = nb_iter + 1
    cond = abs( tmp_y - y0 )
    tmp_y = y0
    print(x0,y0,cond)
    plt.scatter(x0, y0)

plt.title("Gradient Descent (For univariate)")
plt.grid()
```

Output:

```
7.8 77.44 43.56
6.0399999999998 49.5615999999998 27.87840000000002
4.63199999999998 31.71942399999975 17.842176000000002
3.50559999999998 20.30043135999987 11.41899263999998
2.68447999999999 12.992276070399992 7.308155289599995
1.883583999999999 8.315856685055994 4.677219385343998
1.306867199999995 5.321636278435836 2.9934284066201584
0.845493759999994 3.4058472181989354 1.9157890602369085
0.476395007999995 2.1797422196473186 1.2261049985516168
0.1811160063999995 1.3958350285742839 0.784707199730348
-0.05510719488000032 0.8928224131675417 0.5822126074067421
-0.24408575590400025 0.5714063444272267 0.32141606874031583
-0.39526860472320025 0.36570006043342507 0.20570628399380164
-0.5162148837785603 0.234048038677392 0.13165202175603308
-0.6129719070228482 0.14979074475353094 0.08425729392386105
-0.6903775256182786 0.09586607664225966 0.05392466811127128
-0.752300204946229 0.06135428905104623 0.03451178759121343
-0.8018416163956983 0.03926674499266958 0.02208754405837665
-0.8414732931165586 0.02513071679530854 0.014136028197361039
-0.8731786344932468 0.01608365874899753 0.009047058046311007
-0.8985429875945974 0.010293541599358358 0.0057901171496391735
-0.9188343260756779 0.006587866623589345 0.0037056749757690133
-0.93506746086050424 0.0042162346390971495 0.0023371631984492195
-0.9426053968688434 0.00269839016902218 0.0015178444700749694
-0.9584431749507472 0.0017269697081742086 0.000971420468847915
-0.9667545399605977 0.0011052606132314624 0.0006217090949427462
-0.9734036319684781 0.0007073667924681892 0.0003978938207632732
-0.9787229055747825 0.0004527147471796722 0.00025465204528851704
-0.9829783244598259 0.0002897374381949369 0.000162977308898473528
-0.9863826595678688 0.00018543196044473742 0.0001843054777501995
-0.9891061276542886 0.00011867645468466384 6.675550576007439e-05
```



To implement gradient descent, we need a starting point, which we have defined manually as 10 but is often a random initialization in the real world. We also need the gradient function, a learning rate, a maximum number of iterations, and a tolerance to stop the algorithm conditionally.

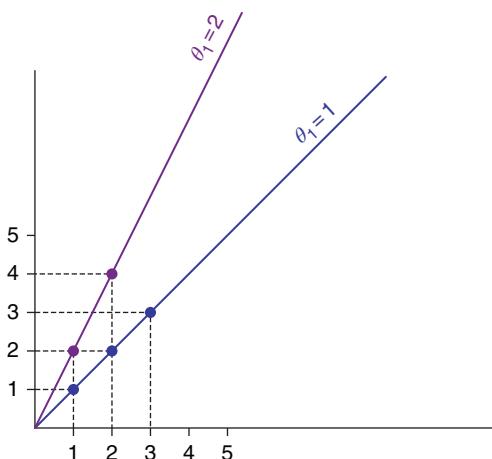
Let us take another example with the cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

We will consider the case of setting $\theta_0 = 0$, meaning that we have the following:

$$h_\theta(x) = \theta_1 x$$

Different lines will pass through the origin for any value of θ_1 because the y -intercept is equal to zero.



We can calculate the cost function manually for different values of θ_1 . Let us set the number of records in our dataset to $m = 3$.

At $\theta_1 = 2$, the following result is obtained:

$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^m (\theta_1(x^i) - y^i)^2$$

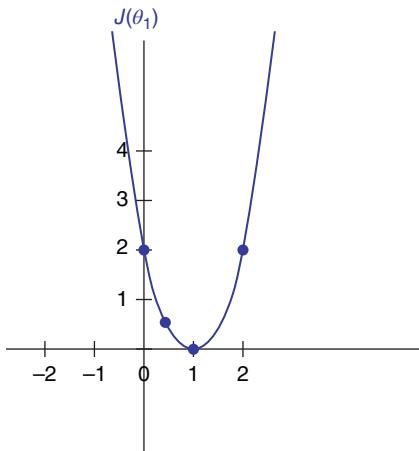
$$J(2) = \frac{1}{2 \times 3} (1^2 + 2^2 + 3^2) = 2.33$$

At $\theta_1 = 1$, the following result is obtained:

$$J(1) = \frac{1}{2 \times 3} (0^2 + 0^2 + 0^2) = 0$$

At $\theta_1 = 0.5$, the following result is obtained:

$$J(0.5) = 0.58$$



Gradient descent does not work for all functions, as they must be differentiable (having derivatives for each point in their domains) and convex (a line segment connecting two points of the function should lie on or above its curve rather than cross it).

In the case of multiple linear regression, after performing feature rescaling (using mean normalization, for example), our cost function will take the following form:

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

The gradient descent is also updated as follows:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \dots, \theta_n)$$

where $j = 0, 1, \dots, n$.

We can use gradient descent or normal equation to find the optimal parameters. Batch gradient descent and stochastic gradient descent are further methodologies we can use; batch gradient descent involves calculations over the full training set at each step, and stochastic gradient descent takes a random instance of training data at each step and then computes the gradient. The consequence is that batch gradient descent is slower for large training datasets, efficient for convex or relatively smooth error manifolds, and scalable with the number of features. Stochastic gradient descent is faster but does not provide an optimal result, as it approaches the minimum but does not settle.

3.1.3 Implementation of Linear Regression

If we go to hephAIstos/Notebooks with a terminal and open a Jupyter Notebook, we will see in the browser a file named ML_Algorithms_Linear_Regression.ipynb. There, we can find all the code examples shown in this section. We can also download the notebook here: https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/ML_Algorithms_Linear_Regression.ipynb.

3.1.3.1 Univariate Linear Regression

To illustrate what has been described above, let us implement a simple linear regression with scikit-learn (*LinearRegression()*) by generating a random dataset.

Input:

```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate random data-set
np.random.seed(0)
# There will be 200 data points ranging from 0 to 200
x = np.random.rand(200, 1)
y = 4 + 2 * x + np.random.rand(200, 1)

# Scikit-learn implementation

# Model initialization
linear_regression_model = LinearRegression()
# Fit the data (train the model)
linear_regression_model.fit(x, y)
# Model prediction
y_predicted = linear_regression_model.predict(x)

# Model evaluation (rmse, r2)
rmse = mean_squared_error(y, y_predicted)
r2 = r2_score(y, y_predicted)

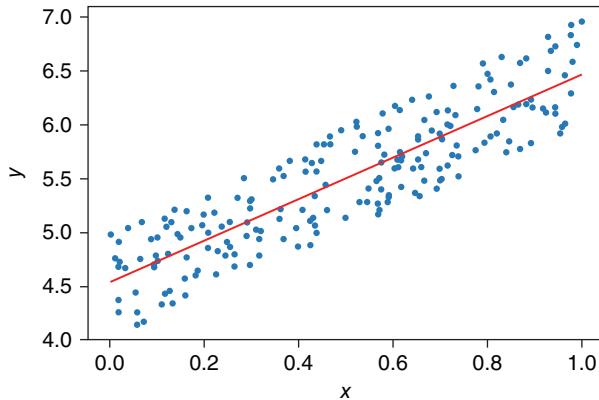
# Printing Slope, Intercept, RMSE, and R2
print('Slope:', linear_regression_model.coef_)
print('Intercept:', linear_regression_model.intercept_)
print('Root mean squared error: ', rmse)
print('R2 score: ', r2)

# Plot data points
plt.scatter(x, y, s=10)
plt.xlabel('x')
plt.ylabel('y')

# Regression line
plt.plot(x, y_predicted, color='r')
plt.show()
```

Output:

```
Slope: [[1.94369415]]
Intercept: [4.5193723]
Root mean squared error: 0.08610037959679763
R2 score: 0.7795855368773891
```



Using the code, we have computed the slope and the intercept and have assessed the model by introducing the R-squared score (coefficient of determination), and the root-mean-squared error (RMSE, the square root of the average of the sum of the squares of residuals):

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(x^i) - y^i)^2}$$

$$R^2 = 1 - \frac{\sum_{i=1}^m (y^i - \bar{y})^2}{\sum_{i=1}^m (h(x^i) - y^i)^2}$$

As we can see in the results, we have reduced the prediction error by 77.95% by using regression.

Now, let us implement a simple linear regression with TensorFlow by generating a random dataset. We will use gradient descent, a learning rate of 0.01, and 1000 epochs.

Input:

```
# Import Libraries
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import numpy
import matplotlib.pyplot as plt
rng = numpy.random

# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50

# Training Data
# Generating random linear data
np.random.seed(0)
# There will be 200 data points ranging from 0 to 200
train_X = np.random.rand(200, 1)
train_Y = 4 + 2 * x + np.random.rand(200, 1)
```

```

n_samples = train_X.shape[0]

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)

# Gradient descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initialize the variables
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(c), \
                  "W=", sess.run(W), "b=", sess.run(b))

    print("Optimization Finished!")
    training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
    print("Training cost=", training_cost, "W=", sess.run(W), "b=", sess.run(b), '\n')

    # Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()

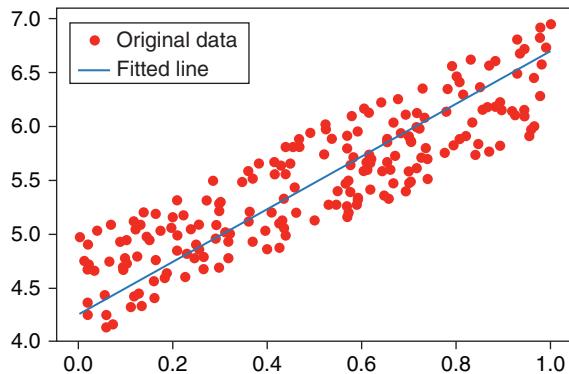
```

Output:

```

Epoch: 0050 cost= 4.046965122 W= 1.706719 b= 1.8089526
Epoch: 0100 cost= 1.194122195 W= 2.2308412 b= 2.860583
Epoch: 0150 cost= 0.389478683 W= 2.4960198 b= 3.4254918
Epoch: 0200 cost= 0.161574557 W= 2.6242018 b= 3.7319198
Epoch: 0250 cost= 0.096121706 W= 2.6800892 b= 3.900989
Epoch: 0300 cost= 0.076482169 W= 2.698006 b= 3.9969735
Epoch: 0350 cost= 0.069815665 W= 2.696131 b= 4.053974
Epoch: 0400 cost= 0.066869356 W= 2.684106 b= 4.090116
Epoch: 0450 cost= 0.065035008 W= 2.66703 b= 4.115022
Epoch: 0500 cost= 0.063579641 W= 2.6476405 b= 4.1337013
Epoch: 0550 cost= 0.062286984 W= 2.6273458 b= 4.1489716
Epoch: 0600 cost= 0.061094370 W= 2.606832 b= 4.16219
Epoch: 0650 cost= 0.059985485 W= 2.5866368 b= 4.1741753
Epoch: 0700 cost= 0.058943450 W= 2.5668473 b= 4.1854954
Epoch: 0750 cost= 0.057964709 W= 2.5475132 b= 4.196224
Epoch: 0800 cost= 0.057046860 W= 2.5287216 b= 4.2064776
Epoch: 0850 cost= 0.056184921 W= 2.5104952 b= 4.216401
Epoch: 0900 cost= 0.055373792 W= 2.492721 b= 4.225916
Epoch: 0950 cost= 0.054615609 W= 2.475573 b= 4.235103
Epoch: 1000 cost= 0.053904906 W= 2.4589992 b= 4.244027
Optimization Finished!
Training cost= 0.053904906 W= 2.4589992 b= 4.244027

```



As we can see, the result shows the weight $W = 2.4589992$ and the bias $b = 4.244027$.

If we want to print the number of GPUs available in our system and how the operations and tensors are assigned (to GPUs, CPUs, etc.), we can add lines at the top of the code.

Input:

```

# Print number of GPUs available
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
# For future use, which devices the operations and tensors are assigned to (GPU, CPU)
tf.debugging.set_log_device_placement(True)

```

3.1.3.2 Multiple Linear Regression: Predicting Water Temperature

In the real world, we rarely have one single independent variable and clean data. To illustrate multiple linear regression, we will use the California Cooperative Oceanic Fisheries Investigations (CalCOFI) dataset, which represents the longest (1949–present) and most complete (more than 50,000 sampling stations) time series of larval fish and oceanographic data

in the world (<https://www.kaggle.com/datasets/sohier/calcofi?select=bottle.csv>). This dataset has become valuable for documenting climatic cycles locally. For this exercise, we will extract the following variables:

- **Depthm:** Depth in meters.
- **T_degC:** Water temperature in degrees Celsius.
- **Salnty:** Salinity in g of salt per kg of water (g/kg).
- **O2ml_L:** O₂ mixing ratio in ml/l.

With these data, we can already ask some questions such as whether there is a relationship between water salinity and water temperature, or if we can predict the water temperature based on salinity and depth in meters. Let us use the water temperature (T_DegC) as our dependent variable (target).

To start, we will capture the dataset using pandas DataFrame, drop rows having at least one missing value, and split the data into the variable we want to predict (T_degC) and the selected features (Depthm, Salnty, O2ml_L).

Input:

```
import pandas as pd

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['Depthm', 'T_degC', 'Salnty', 'O2ml_L']]

# Drop rows having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and x the features (Depthm,
# Salnty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC']
X = df.loc[:, df.columns != 'T_degC']

print(y)
print(X)
```

Output:

T_degC		Depthm	Salnty	O2ml_L
2160	10.300	2160	0	33.0300
2161	18.460	2161	6	32.9200
2162	10.290	2162	10	32.9510
2163	10.290	2163	15	32.9900
2164	10.330	2164	20	33.0050
...
864858	18.744	864858	0	33.4083
864859	18.744	864859	2	33.4083
864860	18.692	864860	5	33.4150
864861	18.161	864861	10	33.4062
864862	17.533	864862	15	33.3880

661489 rows × 1 columns

661489 rows × 3 columns

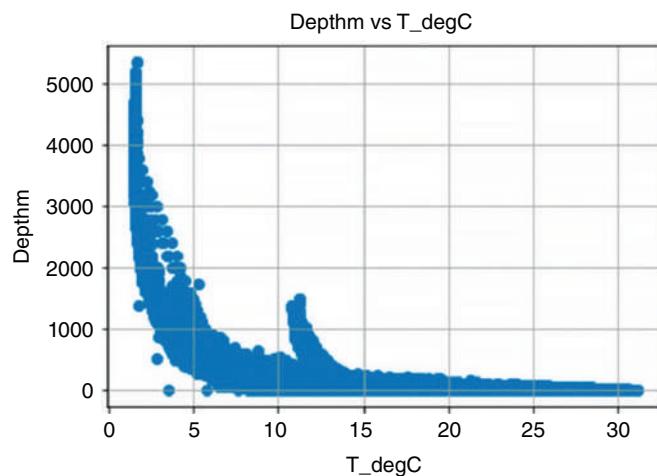
Before creating the linear regression model, we can check visually whether a linear relationship exists between the variables T_degC versus Depthm, T_degC versus Salnty, and T_degC versus O2ml_L. To perform this check, we can compute some scatter diagrams with the matplotlib library by adding the lines below.

Input:

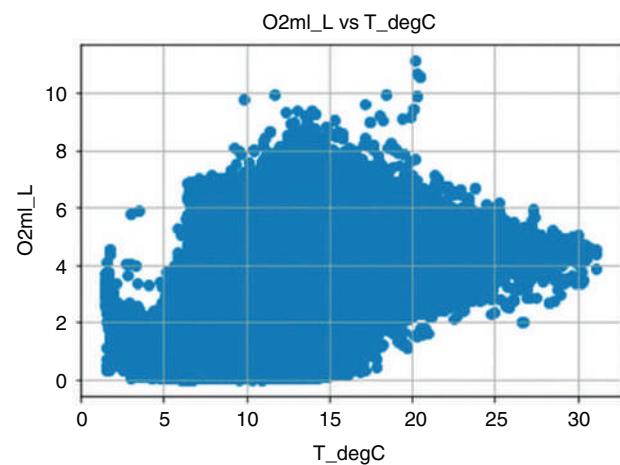
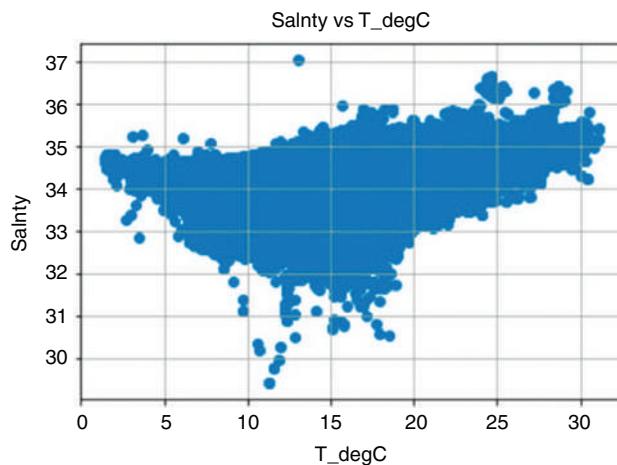
```
# Plot diagrams with the matplotlib library.
plt.scatter(y, X['Depthm'])
plt.title('Depthm Vs T_degC')
plt.xlabel('T_degC')
plt.ylabel('Depthm')
plt.grid(True)

plt.show()
```

Output:



We can do the same for the other variables:



We can see that there is no linearity between T_degC and Depthm. We will drop the Depthm variable from our linear regression model and start dividing the data between the variables to predict T_DegC and both features Depthm and Salnty.

Input:

```

import pandas as pd

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]

# Drop rows having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
# Salnty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC']
X = df.loc[:, df.columns != 'T_degC']

```

Multiple Linear Regression with scikit-learn To apply linear regression to the data and find the intercept and coefficients, we can use sklearn.

Input:

```

# Import libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
# Salnty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC']
X = df.loc[:, df.columns != 'T_degC']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Normalize the data
Normalize = preprocessing.Normalizer()
# Transform data

```

```

X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Compute a linear regression model with sklearn
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)

print('Intercept: \n', regr.intercept_)
print('Coefficients: \n', regr.coef_)

# Compute and print predicted output with X_test as inputs
print('\n')
print ('Predicted T_degC: \n', regr.predict(X_test))

# To save the model for future use
from joblib import dump, load
dump(regr, 'linear_regression.joblib')

# To load the model when needed
regr = load('linear_regression.joblib')

```

Output:

```

Intercept:
[143.98894321]
Coefficients:
[[ -138.18191503   41.17953079]]

Predicted T_degC:
[[15.24926345]
 [ 8.14078918]
 [ 9.9864041 ]
 ...
 [15.81746572]
 [ 7.8816554]
 [ 7.53595955]]

```

At this stage, we are ready to apply our linear regression model to new data, as the output shows the intercept and coefficients:

```
T_degC = Intercept + (Salnty Coefficient) × X1 + (O2ml_L Coefficient) × X2
```

Once we supply numerical values, we obtain the following:

```
T_degC = 143.98894321 + (-138.18191503) × X1 + (41.179.53079) × X2
```

We can use new data (X_test) and apply our model (regr) to predict the water temperature.

Input:

```

# Compute and print predicted output with X_test as inputs
print('\n')
print ('Predicted T_degC: \n', regr.predict(X_test))

```

Output:

```
Predicted T_degC :
[[15.24926345]
 [ 8.14078918]
 [ 9.9864041 ]
...
[15.81746572]
[ 7.8816554]
[ 7.53595955]]
```

To use the final model in the future, it is important to save the model and load it when needed.

Input:

```
# To save the model for future use
from joblib import dump, load
dump(regr, 'linear_regression.joblib')

# To load the model when needed
regr = load('linear_regression.joblib')
```

An important step in multiple linear regression is data scaling. In our code, we will normalize both X_train and X_test after splitting the data.

Input:

```
# Normalize the data
Normalize = preprocessing.Normalizer()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)
```

We could normalize the data before splitting it and then create and train our model on the normalized data. We can do it to understand the mathematical structure of our models. But in real life, we do not have the new data. Therefore, we need to normalize it as it comes. This is all dependent on the size of the datasets and whether both training and test sets are equally representative of the domain we are attempting to learn with our model. If we have many data points and the test set is representative of our training set, then we can normalize the test dataset as shown above or use the normalization parameters of the training set (mean, standard deviation). Both methods would be satisfactory. For a small but representative test dataset, it would be better to use the training parameters only, as sampling errors may negatively bias the predictions. If the test dataset is not representative of the training set, then we need to reconsider our sampling procedure.

There is no learning rate here, as this is not learned with gradient descent. In addition, the regressors X are normalized before regression by subtracting the mean and dividing by the l2 norm. We can use a different scaling method; for example, we can use StandardScaler.

It is possible to fit a linear model fitted by minimizing a regularized empirical loss stochastic gradient descent by using SGDRegressor in scikit-learn. The default value of the learning rate is 0.01.

Input:

```
# Importing. libraries
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import SGDRegressor
```

```

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
# Salnty, O2ml_L)
y = pd.DataFrame(df.loc[:, df.columns == 'T_degC'], columns = ["T_degC"])
X = pd.DataFrame(df.loc[:, df.columns != 'T_degC'])

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# with sklearn
from sklearn import linear_model
regr = linear_model.SGDRegressor(learning_rate = 'constant', max_iter=1000, tol=1e-3)
regr.fit(X_train, y_train)

print('Intercept: \n', regr.intercept_)
print('Coefficients: \n', regr.coef_)

# Prediction with sklearn
print ('Predicted T_degC: \n', regr.predict(X_test))

# To save the model for future use
from joblib import dump, load
dump(regr, 'linear_regression.joblib')

# To load the model when needed
regr = load('linear_regression.joblib')

```

Output:

```

Intercept :
[10.99528094]
Coefficients :
[1.92547404 4.84608926]
Predicted T_degC :
[16.19572688 8.31397028 10.44245724 ... 16.86515669 7.3999601
 7.90504924]

```

Multiple Linear Regression with Statsmodels It is also possible to display a comprehensive table with statistical information generated by *statsmodels* to provide insights regarding our model. Let us add a few more lines.

Input:

```
# Import libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and x the features (Depthm,
# Salnty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC']
X = df.loc[:, df.columns != 'T_degC']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Normalize the data
Normalize = preprocessing.Normalizer()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# with statsmodels
import statsmodels.api as sm
X_train = sm.add_constant(X_train) # adding a constant
model = sm.OLS(y_train, X_train).fit()

# Print model summary
print_model = model.summary()
print(print_model)

print('Statsmodels parameters:')
print(np.round(reg.params, 3))
print('\n')
```

Output:

```
OLS Regression Results
=====
Dep. Variable: T_degC R-squared: 0.628
Model: OLS Adj. R-squared: 0.628
Method: Least Squares F-statistic: 4.463e+05
Date: Sat, 07 May 2022 Prob (F-statistic): 0.00
Time: 08:31:27 Log-Likelihood: -1.2519e+06
No. Observations: 529191 AIC: 2.504e+06
Df Residuals: 529188 BIC: 2.504e+06
Df Model: 2
Covariance Type: nonrobust
=====
            coef    std err      t      P>|t|      [0.025      0.975]
const    143.9889   2.611    55.142     0.000    138.871    149.107
x1       -138.1819   2.604   -53.068     0.000   -143.285   -133.078
x2        41.1795   0.260    158.293     0.000     40.670     41.689
=====
Omnibus: 146709.668 Durbin-Watson: 2.001
Prob(Omnibus): 0.000 Jarque-Bera (JB): 626334.600
Skew: 1.310 Prob(JB): 0.00
Kurtosis: 7.641 Cond. No. 1.47e+03
=====
```

Notes:

- [1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
- [2] The condition number is large, 1.47e+03. This might indicate that there are strong multicollinearity or other numerical problems.

Statsmodels parameters:

```
const    143.989
x1       -138.182
x2        41.180
dtype: float64
```

As we can see in the results, the model does not perform very well, as we have reduced the prediction error by only 62.8% by using regression.

Multiple Linear Regression with TensorFlow As we will see, computing a linear regression on the same data with TensorFlow will provide the same results; however, it is also a way to learn some vector and matrix operations (multiply, transpose, inverse, etc.) in TensorFlow. We will see that it is quite similar to that of np.array with some differences.

Input:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import tensorflow as tf

# Print number of GPUs available
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
# For future use, which devices the operations and tensors are assigned to (GPU, CPU)
tf.debugging.set_log_device_placement(True)

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')
```

```

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
# Salnty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC'].values.ravel()
X = df.loc[:, df.columns != 'T_degC']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Normalize the data
Normalize = preprocessing.Normalizer()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Shape of the data (number of rows and columns)
nrow, ncol = X_train.shape; print (nrow, ncol)
nparam = ncol+1 # number of parameters

# Keep columns name for table of results
row_name_results = [['Intercept', 'Salnty', 'O2ml_L']]

##### Using sklearn
from sklearn import linear_model
reg_mod = linear_model.LinearRegression()
reg_mod.fit(X_train, y_train)

df_results_sk = pd.DataFrame(
    np.hstack([reg_mod.intercept_, reg_mod.coef_]))
df_results_sk.columns = ["estimate"]
df_results_sk.index = row_name_results
print("\n##### using sklearn #####")
print(df_results_sk)

##### Using statsmodels
import statsmodels.api as sm
Xw1 = sm.add_constant(X_train)
ols = sm.OLS(y_train, Xw1)
fit = ols.fit()

df_results_sm = pd.DataFrame(np.vstack([fit.params,
                                         fit.bse, fit.params/fit.bse]).T)
df_results_sm.columns = ["estimate", "std.err", "t-stats"]
df_results_sm.index = row_name_results

```

```

print("\n##### using statsmodels #####")
print(df_results_sm)

##### Using matrix formula (np.array)

mX    = np.column_stack([np.ones(nrow), X_train])
beta  = np.linalg.inv(mX.T.dot(mX)).dot(mX.T).dot(y_train)
err   = y_train - mX.dot(beta)

s2    = err.T.dot(err)/(nrow - ncol - 1)
cov_beta = s2*np.linalg.inv(mX.T.dot(mX))
std_err = np.sqrt(np.diag(cov_beta))

df_results_np = pd.DataFrame(
    np.row_stack((beta, std_err, beta/std_err)).T)
df_results_np.columns = ["estimate", "std.err", "t-stats"]
df_results_np.index = row_name_results

print("\n##### using np.array #####")
print(df_results_np)

##### Using matrix formula (Tensorflow)

import tensorflow as tf

# from np.array
y_train = tf.constant(y_train, shape=[nrow, 1])
X_train = tf.constant(X_train, shape=[nrow, ncol])

# need double tensor
one = tf.cast(tf.ones([nrow, 1]), tf.float64)
oneX = tf.concat([one, X_train], 1); # 1, X

XtX = tf.matmul(oneX, oneX, transpose_a=True)
Xty = tf.matmul(oneX, y_train, transpose_a=True)
beta = tf.matmul(tf.linalg.inv(XtX), Xty)
err = y_train - tf.matmul(oneX, beta)
s2 = tf.matmul(err, err, transpose_a=True)/(nrow - nparam)
cov_beta = s2*tf.linalg.inv(XtX)
std_err = tf.sqrt(tf.linalg.diag_part(cov_beta))
beta = tf.reshape(beta, [nparam])

est_out = tf.stack([beta, std_err, beta/std_err], 1)
df_results_tf = pd.DataFrame(np.asarray(est_out))
df_results_tf.columns = ["estimate", "std.err", "t-stats"]
df_results_tf.index = row_name_results

print("\n##### using Tensorflow #####")
print(df_results_tf)

```

Output:

```
Num GPUs Available: 1

##### using sklearn #####
    estimate
Intercept 143.988943
Salnty -138.181915
O2ml_L 41.179531

##### using statsmodels #####
    estimate std.error t-stats
Intercept 143.988943 2.611227 55.142251
Salnty -138.181915 2.603864 -53.068028
O2ml_L 41.179531 0.260148 158.292511

##### using np.array #####
    estimate std.error t-stats
Intercept 143.988944 2.611227 55.142251
Salnty -138.181915 2.603864 -53.068029
O2ml_L 41.179531 0.260148 158.292511

##### using Tensorflow #####
    estimate std.error t-stats
Intercept 143.988943 2.611227 55.142251
Salnty -138.181915 2.603864 -53.068028
O2ml_L 41.179531 0.260148 158.292511
```

Multiple Linear Regression with Keras on TensorFlow Let us now implement a multiple linear regression with Keras on TensorFlow. The approach is very similar to scikit-learn. We capture the dataset in Python using pandas DataFrame, we select the variables to use, we drop rows having at least one missing value, we divide the data with a target variable and the features, we split the data to produce training and test datasets, we scale the data, and we create a sequential model. The sequential layer allows stacking of one layer on top of the other, enabling the data to flow through them. We will use a mini-batch gradient descent optimizer and mean square loss. Finally, we will check the performance by examining the loss over time; over time, the loss should decrease.

Let us first code a univariate linear regression with Keras on TensorFlow.

Input:

```
#importing the libraries
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]
```

```

# Drop rows having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
SaInty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC'].values.ravel()
X = df.loc[:, df.columns == 'Salnty']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Creating a Sequential Model with TF2
# Sequential Layer allows stacking of one layer on top of the other , enabling the
data to flow through them
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
# As we have only one feature, input_dim = 1
model.add(Dense(1, input_dim = 1, activation = 'linear'))

# We use mini-batch gradient descent optimizer and mean square loss
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.losses import mse
model.compile(optimizer=SGD(learning_rate=0.0001), loss=mse)
train = model.fit(X_train,y_train,epochs=50)

# Performance Analysis: loss over time
# We should see that the loss is reduced over time
plt.plot(train.history['loss'],label='loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()

# Testing the model and print the predictions of the inputs X_test
y_pred = model.predict(X_train)

# Our linear equation with weights and bias
w0 = model.layers[0].get_weights()[0][0]
b = model.layers[0].get_weights()[1]
print("Linear Regression Equation: %f x X_train + %f"%(w0,b))

# Plot the model (regression line in green)
plt.scatter(X_train, y_train, c='blue')
plt.plot(X_train, y_pred, color='green')

# If we plot the linear equation in red, we will have the same line as the previous one
#(green)
#The red line will cover the green one.
plt.plot(X_train, w0*X_train + b, color='r')
plt.xlabel('X_train')

```

```

plt.ylabel('y_train')
plt.show()

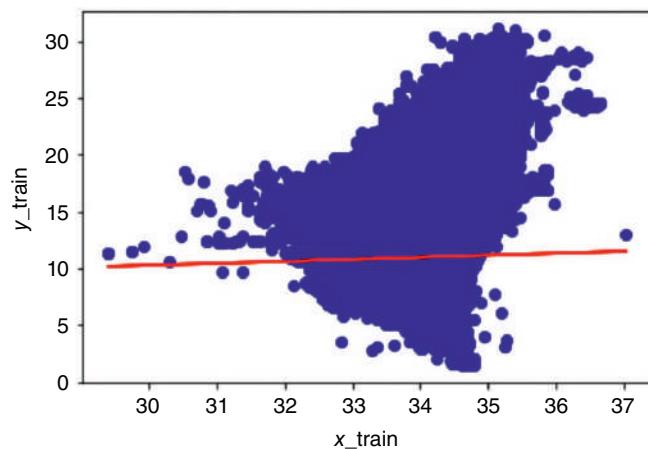
# Let's predict new values
new_predictions = model.predict(X_test)
print(new_predictions)

# Save our model for future use
model.save('../Outputs/keras_linear_model')

```

Output:

Regression Equation: $0.173148 \times X_{\text{train}} + 5.086714$



```

[[10.906912]
 [10.992794]
 [10.961627]
 ...
 [10.905527]
 [10.973402]
 [11.017553]]

```

We have chosen a learning rate of 0.0001 and 50 epochs. To plot the regression line, we can use two different methods that produce the same result. The first method is to compute the weights and bias of our model and plot the equation:

```

# Our linear equation with weights and bias
w0 = model.layers[0].get_weights()[0][0]
b = model.layers[0].get_weights()[1]
plt.plot(X_train, w0*X_train + b, color='r')

```

The second is simply the following:

```
plt.plot(X_train, y_pred, color='green')
```

Our model is ready to predict new values (X_{test}):

```

# Let's predict new values
new_predictions = model.predict(X_test)
print(new_predictions)

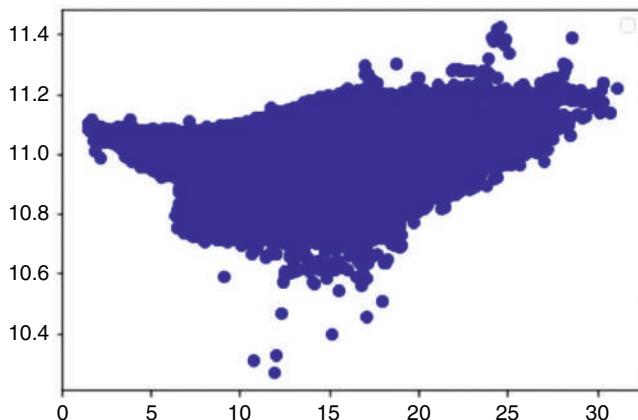
```

We can also print the true values (y_{test}) versus predicted values (new_predictions).

Input:

```
# We can also print the true values (y_test) versus predicted values (new_predictions)
plt.scatter(y_test,new_predictions,color='b')
```

Output:



For future use, we also need to save the model using the following line:

```
# Save our model for future use
model.save('..../Outputs/keras_linear_model')
```

In the future, we can load it by inserting the following line:

```
# Load our model for future use
model = keras.models.load_model('..../Outputs/keras_linear_model')
```

Let us now code a multiple (two-feature) linear regression with Keras on TensorFlow.

Input:

```
# Importing the libraries
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
from sklearn import preprocessing

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '..../data/datasets/bottle.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Select the variables we want to keep: 'Depthm', 'T_degC', 'Salnty', 'O2ml_L'
df = df[['T_degC', 'Salnty', 'O2ml_L']]
```

```

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (T_degC) and X the features (Depthm,
SaInty, O2ml_L)
y = df.loc[:, df.columns == 'T_degC']
X = df.loc[:, df.columns != 'T_degC']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Normalize the data
Normalize = preprocessing.Normalizer()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Creating a Sequential Model with TF2
# Sequential Layer allows stacking of one layer on top of the other , enabling the
data to flow through them
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# We have two inputs for our model: 'Salnty' and 'O2ml_L' features
model = Sequential()
model.add(Dense(1, input_dim = 2, activation = 'linear'))
model.summary()

# Optimizer and Gradient Descent
# We use mini-batch gradient descent optimizer and mean square loss
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.losses import mse
model.compile(optimizer=SGD(learning_rate=0.01),loss=mse)
train = model.fit(X_train,y_train,epochs=10)

# Performance Analysis: loss over time
# We should see that the loss is reduced over time
plt.plot(train.history['loss'],label='loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()

# Model prediction
y_pred = model.predict(X_train)

# Extracting the weights and biases is achieved quite easily
model.layers[0].get_weights()

```

```

# We can save the weights and biases in separate variables
weights = model.layers[0].get_weights()[0]
bias = model.layers[0].get_weights()[1]
print('Intercept and Weights from keras model')
print("Intercept:")
print(bias)
print("Weights:")
print(weights)
print('\n')

# Model evaluation using scikit-learn tooling
from sklearn.metrics import mean_squared_error, r2_score
print('Keras model evaluation:')
rmse = mean_squared_error(y_train, y_pred)
r2 = r2_score(y_train, y_pred)

# Printing values
print('Root mean squared error: ', rmse)
print('R2 score: ', r2)

print('\n')

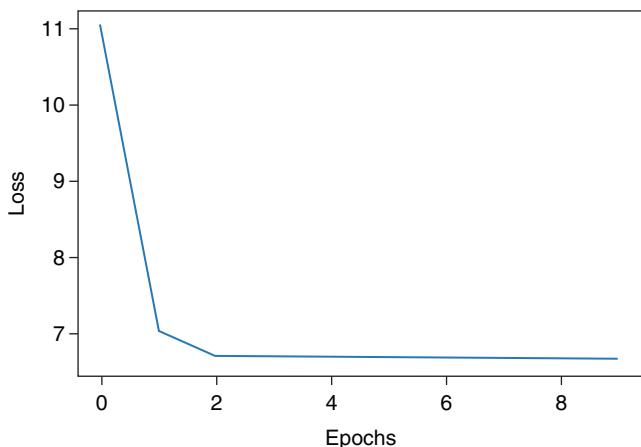
# Make predictions with X_test
y_pred_new = model.predict(X_test)
print('predictions:')
print(y_pred_new)

print('\n')

# Let's predict new values
new_predictions = model.predict(X_test)
print('\n')
print(new_predictions)

```

Output:



```

Intercept and Weights from keras model
Intercept:
[5.45528]

```

```
Weights:
[[4.5139156e-02]
 [5.4653141e+01]]
```

```
Keras model evaluation:
Root mean squared error: 6.685948426017527
R2 score: 0.6254232016793214
```

If we rerun the same code, the output will be different because Keras will retrain the current model. In our example, we can see that we have reduced the prediction error by only 62.8% by using regression. It is not easy to provide rules, as they would depend on the context. R-squared has been proposed in many fields, but there is no standard guideline. In academic research, R-squared values of 0.75, 0.5, and 0.25 can be described respectively as strong, moderate, and weak (Henseler et al. 2009).

3.2 Logistic Regression

Logistic regression is a supervised machine learning classifier used to predict categorical variables or discrete values. There are many applications of logistic regression, such as determining the probability of having a heart attack according to weight or exercise, filtering emails, or calculating the probability of getting accepted in a contest. We have seen that linear regression solves regression problems and predicts continuous values. Logistic regression solves classification problems by providing outputs such as 0 and 1, positive and negative, or multiple classes; multinomial logistic regression is widely used in text classification or part-of-speech labeling. In logistic regression, we are not looking for the best-fitting line but rather building an s-shaped curve, called a logistic function, that lies between 0 and 1.

Logistic regression extracts continuous features from the input, multiplies each by a weight, sums them, and passes the sum through a sigmoid function to generate a probability. For decisions, we include a threshold function (sigmoid or logistic function) such that any value above the threshold will tend toward 1 and any value below the threshold will tend toward 0. The weights (vector w and bias b) are learned from a training dataset that is labeled and use a loss function, such as the cross-entropy loss, that needs to be minimized.

In this section, we will learn the concepts of sigmoid function or logistic function, as well as the logit function, odds ratio, and cross-entropy.

3.2.1 Binary Logistic Regression

The objective of binary logistic regression is to make a binary decision by training a classifier on real-value features. Similar to linear regression, logistic regression learns a vector of weights (w_i) and a bias term or intercept (b) from a training set.

After extracting continuous features from the input, we multiply features by the weights, sum them, and add the bias:

$$z = \left(\sum_{i=1}^n w_i x_i \right) + b$$

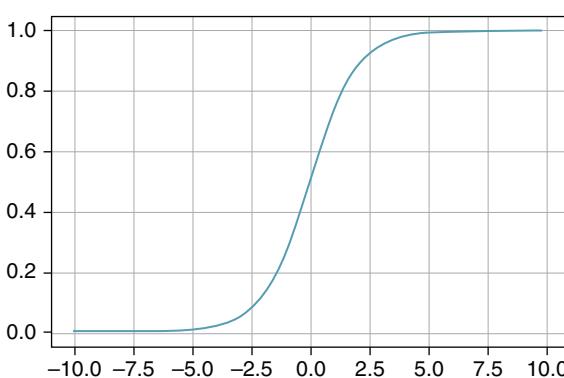
We can represent the equation above as follows (with a dot product):

$$z = w \cdot x + b$$

So far, nothing in the equation above forces us to be a probability lying between 0 and 1, as z ranges from $-\infty$ to $+\infty$. We can pass the sum through a sigmoid function (Figure 3.1) to generate a probability, also called a logistic function, allowing us to map real values into the range $[0, 1]$:

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

Figure 3.1 The sigmoid function is nearly linear around 0 and then flattens.



Now that we have obtained a number in the range $[0, 1]$ by applying the sigmoid function to the sum of the weighted features, we state a probability, such as $P(y = 1) + P(y = 0) = 1$:

$$\begin{aligned} P(y = 1) &= \sigma(z) = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} \\ P(y = 0) &= 1 - \sigma(z) = 1 - \sigma(w \cdot x + b) = 1 - \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{e^{-(w \cdot x + b)}}{1 + e^{-(w \cdot x + b)}} \\ P(y = 1) + P(y = 0) &= 1 \end{aligned}$$

We expect our model to produce outputs based on probability scores between 0 and 1. Let us say we would like our model to identify whether patients have a disease (target = 1) or not (target = 0) according to a certain number of features. By defining a threshold value, for example, 0.5, we can decide in which category the patient belongs. If the prediction function returns a value of 0.6, we would classify the patient in the “no disease” category.

3.2.1.1 Cost Function

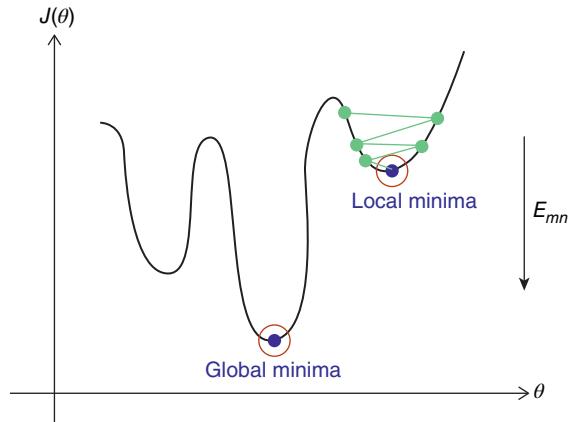
We have seen in Section 3.1 that we need a cost function $J(\theta)$, which is the minimum of the root-mean-squared error of the model, obtained by subtracting the predicted values from actual values. It is an optimization objective to design an accurate model that minimizes error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^i) - y^i)^2$$

In the case of a univariate linear regression, the following has been stated:

$$h_\theta(x) = y = \theta_0 + \theta_1 x$$

The problem with using the same cost function for logistic regression is that it will end up as a non-convex function with several local minima. It will be challenging to minimize the cost value and find the global minimum.



We need a loss function that expresses how close the model output $\sigma(w \cdot x + b)$ is to the correct output ($y = 0$ or 1) given an observation x . The idea is to have a loss function that tends to select the correct labels for the observed (training) data to be more probable. This goal can be achieved by maximizing a likelihood function, and this is what is called conditional maximum likelihood estimation. We will choose the weights and bias (w, b) that maximize the log probability of the true y labels in the training data given the observed data x . For logistic regression, the loss function is the negative log-likelihood loss, also called cross-entropy loss:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^i \log((h_\theta(x^i))) + (1-y^i) \log(1-h_\theta(x^i))]$$

where $h_\theta(x^i) = \theta_0 + \theta_1 x^i$, (x^i, y^i) is the i th training data, and m is the number of training examples. The cost function is composed of two parts:

$$J(h_\theta(x), y) = f(x) = \begin{cases} -\log(h_\theta(x)), & y = 1 \\ -\log(1 - h_\theta(x)), & y = 0 \end{cases}$$

The cost function can be simplified by the notation we have used above:

$$J(\sigma(w \cdot x + b), y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log(1 - \sigma(w \cdot x + b))]$$

3.2.1.2 Gradient Descent

As described above for linear regression (in Section 3.1.2), we can use gradient descent to find the optimal θ values:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

where α is the learning rate.

3.2.2 Multinomial Logistic Regression

Multinomial logistic regression is an extension of binary logistic regression. Instead of allowing only two categories of the target, it allows more than two. In multinomial logistic regression, we need the softmax function instead of the sigmoid function, which turns all the inputs (vector) and maps them to a probability distribution, with each value ranging between 0 and 1 with the sum of all values being 1.

The softmax function is defined as follows:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

where $z = [z_1, z_2, z_3, \dots, z_K]$ is a vector of dimension K and $1 \leq i \leq K$. The softmax of z is also a vector:

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{j=1}^K e^{z_j}}, \frac{e^{z_2}}{\sum_{j=1}^K e^{z_j}}, \dots, \frac{e^{z_K}}{\sum_{j=1}^K e^{z_j}} \right]$$

If we have K classes, we will have K different weight vectors. We will have a matrix packing all the weight vectors together and a vector output \hat{y} . In binary logistic regression, we use a single weight vector w and a scalar output.

In multinomial logistic regression, we also use maximum likelihood estimation. The loss function generalizes for binary logistic regression from 2 to K classes. Both y and $\sigma(w \cdot x + b)$ can be represented as vectors of K elements (\hat{y}, y):

$$J(\hat{y}, y) = \sum_{k=1}^K y_k \log(\hat{y}_k)$$

where y is called a one-hot vector, meaning that all positions in the vector are 0 except the entry representing the class that the observations fall into is 1.

Finally, we can apply gradient descent to minimize the cost.

We have seen binary logistic regression for data having two types of possible output and multinomial logistic regression for more than two categories of output. We can also explore ordinal logistic regression for data having more than two categories but in ordering.

3.2.3 Multinomial Logistic Regression Applied to Fashion MNIST

If we go to hephAistos/Notebooks with a terminal and open a Jupyter Notebook, we will see in the browser a file named `ML_Algorithms_Logistic_Regression.ipynb` that contains all code examples shown in this section. The notebook can also be downloaded here: https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/ML_Algorithms_Logistic_Regression.ipynb.

To practice and apply logistic regression, we will use the famous Fashion MNIST dataset, which is a dataset of 70,000 28×28 labeled Zalando's article images (784 pixels in total per image). There is a training set of 60,000 examples that we will use here. A test set of 10,000 examples is also available. The dataset is available at Kaggle: <https://www.kaggle.com/datasets/zalando-research/fashionmnist>. Each pixel has a single value between 0 and 255, indicating the lightness of that pixel; higher values are darker pixels. When the data are extracted, we obtain 785 columns. One column is the labels (0: T-shirt/top, 1: trouser, 2: pullover, etc.); the rest of the columns are the 784 features, which are the pixel numbers and their respective values.

3.2.3.1 Logistic Regression with scikit-learn

Let us start with a few lines of code to prepare our model and assess our data.

Input:

```
# Import Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/fashion-mnist_train.csv'
df = pd.read_csv(csv_data, delimiter=',')
df.head()

# Initiate the label values
Labels = {0:'T-shirt/top',
          1:'Trouser',
          2:'Pullover',
          3:'Dress',
          4:'Coat',
          5:'Sandal',
          6:'Shirt',
          7:'Sneaker',
          8:'Bag',
          9:'Ankle boot'}

# Divide the data, y the variable to predict (label) and X the features
X = df[df.columns[1:]]
y = df['label']

# Let's just show a couple of examples
features = X.loc[1].values
print("Actual Label: ", Labels[y.loc[1]])
plt.imshow(features.reshape(28,28))
```

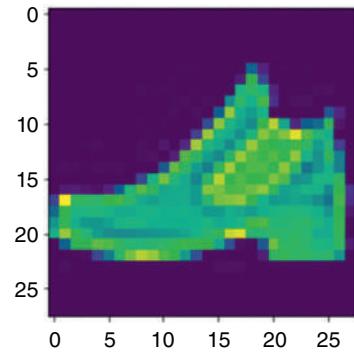
Output:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pi
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0	
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0	
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

5 rows × 785 columns

Actual Label: Ankle boot

<matplotlib.image.AxesImage at 0x12a4f5c70>



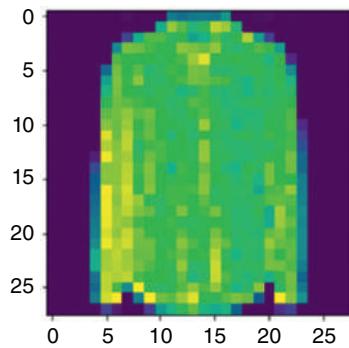
Input:

```
features = X.loc[2].values
print("Actual Label: ", Labels[y.loc[2]])
plt.imshow(features.reshape(28,28))
```

Output:

Actual Label: Shirt

<matplotlib.image.AxesImage at 0x12a3ccc10>



The pixel values range from 0 to 255. Therefore, dividing all values by 255 will convert them to a range from 0 to 1.

Input:

```
X = X/255
X.head()
```

Output:

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781
0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0
1	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0
2	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.019608	0.0	0.0	...	0.000000	0.0	0.0	0.117647	0.168627	0.000000	0.0
3	0.0	0.0	0.0	0.003922	0.007843	0.0	0.0	0.000000	0.0	0.0	...	0.011765	0.0	0.0	0.000000	0.000000	0.003922	0.0
4	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0

5 rows × 784 columns

We will now split the data into training and testing sets, initialize the model, and train it on the training dataset. Once the model has been trained, we can perform predictions with the test dataset and print metrics for the model (classification accuracy, precision, recall, f1-score). We can also perform a cross-validation.

Input:

```
# Splitting the data : training (80%) and test (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Sckit-learn implementation

# Model initialization
logistic_model = LogisticRegression(solver='sag', multi_class='auto')
# Fit the data (train the model)
logistic_model.fit(X_train, y_train)
# Model prediction on new data (X_test)
y_pred = logistic_model.predict(X_test)
print(y_pred)

# Model Metrics and Evaluation
print("classification accuracy:", metrics.accuracy_score(y_test, y_pred))
print("precision:", metrics.precision_score(y_test, y_pred, average='micro'))
print("recall:", metrics.recall_score(y_test, y_pred, average='micro'))
print("f1 score:", metrics.f1_score(y_test, y_pred, average='micro'))
print("cross validation:", cross_val_score(clf, X, y, cv=cv))
```

Output:

[7 8 8 ... 9 5 5]

```
classification accuracy: 0.8510833333333333
precision: 0.8510833333333333
recall: 0.8510833333333333
f1 score: 0.8510833333333333
cross validation: [0.85583333 0.85575 0.8555 0.84933333 0.85125]
```

We can also use the *classification_report* from sklearn.

Input:

```
print(classification_report(y_test,y_pred))
```

Output:

	precision	recall	f1-score	support
0	0.79	0.82	0.80	1232
1	0.97	0.96	0.96	1174
2	0.76	0.76	0.76	1200
3	0.85	0.87	0.86	1242
4	0.74	0.77	0.75	1185
5	0.93	0.94	0.93	1141
6	0.65	0.58	0.61	1243
7	0.93	0.94	0.93	1224
8	0.95	0.95	0.95	1149
9	0.96	0.95	0.96	1210
accuracy			0.85	12000
macro avg	0.85	0.85	0.85	12000
weighted avg	0.85	0.85	0.85	12000

3.2.3.2 Logistic Regression with Keras on TensorFlow

Using the same dataset, we will now perform a multinomial logistic regression using Keras on TensorFlow. For this process, we will prepare the data as we did previously for sklearn, define the number of classes and features, choose softmax, and use stochastic gradient descent as the optimizer.

Input:

```
import numpy as np
import pandas as pd
import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import SGD

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/fashion-mnist_train.csv'
df = pd.read_csv(csv_data, delimiter=',')

# Divide the data, y the variable to predict (label) and X the features
X = df[df.columns[1:]]
y = df['label']

# As the pixel values range from 0 to 256, apart from 0 the range is 255.
# So dividing all the values by 255 will convert it to range from 0 to 1.
X = X/255

# Splitting the data : training (80%) and test (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define number of classes and number of features to include in our model
number_of_classes = 10
number_of_features = X_train.shape[1]
```

```

keras_model = Sequential()
keras_model.add(Flatten(input_dim=number_of_features))
keras_model.add(Dense(number_of_classes, activation='softmax'))

keras_model.compile(optimizer = SGD(learning_rate = 1e-2),
                     loss = 'sparse_categorical_crossentropy',
                     metrics = ['sparse_categorical_accuracy'])

history = keras_model.fit(X_train, y_train, epochs=5)

```

Output:

```

Epoch 1/5
1500/1500 [=====] - 3s 2ms/step - loss: 0.8728 - sparse_categorical_accuracy: 0.7248
Epoch 2/5
1500/1500 [=====] - 3s 2ms/step - loss: 0.6190 - sparse_categorical_accuracy: 0.8002
Epoch 3/5
1500/1500 [=====] - 3s 2ms/step - loss: 0.5644 - sparse_categorical_accuracy: 0.8152
Epoch 4/5
1500/1500 [=====] - 3s 2ms/step - loss: 0.5349 - sparse_categorical_accuracy: 0.8226
Epoch 5/5
1500/1500 [=====] - 3s 2ms/step - loss: 0.5158 - sparse_categorical_accuracy: 0.8282

```

We can now predict and assess the Keras model.

Input:

```
keras_model.evaluate(X_test, y_test) # loss, sparse_categorical_accuracy
```

Output:

```
[0.5115869045257568, 0.8264166712760925]
```

Input:

```

y_keras_pred = keras_model.predict(X_test)
print(y_keras_pred)
print('\n')

print(classification_report(y_test,
                           (tf.argmax(keras_model.predict(X_test), axis=1)).numpy()))

```

Output:

```

[[3.17936647e-05 7.14810594e-05 1.38558680e-04 ... 9.62919056e-01
 1.18203030e-03 4.68243798e-03]
 [6.33775755e-07 1.89247302e-08 3.79887570e-05 ... 1.49886512e-06
 9.99849796e-01 2.32878961e-06]
 [1.99618880e-05 5.27914881e-06 7.60947005e-04 ... 1.03267164e-04
 9.92728114e-01 1.31478976e-03]

...
[8.41648944e-06 5.66815834e-06 1.70480325e-05 ... 2.39801109e-01
 5.72470832e-04 7.31220961e-01]
[2.08604592e-03 1.95018970e-03 3.38473427e-03 ... 1.33155808e-01
 9.27490555e-03 4.09566723e-02]
[8.82503169e-04 6.67409913e-04 1.14875415e-03 ... 5.91883995e-03
 1.06459493e-02 3.06677762e-02]]

```

	precision	recall	f1-score	support
0	0.81	0.78	0.80	1232
1	0.97	0.94	0.95	1174
2	0.75	0.72	0.73	1200
3	0.83	0.85	0.84	1242
4	0.68	0.76	0.72	1185
5	0.91	0.87	0.89	1141
6	0.59	0.57	0.58	1243
7	0.89	0.90	0.89	1224
8	0.94	0.93	0.94	1149
9	0.90	0.94	0.92	1210
accuracy			0.82	12000
macro avg	0.83	0.83	0.83	12000
weighted avg	0.83	0.82	0.82	12000

3.2.4 Binary Logistic Regression with Keras on TensorFlow

If we want to use binary logistic regression with Keras, we need to change some options, including replacing softmax with the sigmoid function. Let us use the dataset provided by the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to predict whether a patient has diabetes based on several measurements (<https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database>).

Input:

```

import numpy as np
import pandas as pd
import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import SGD

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/diabetes.csv'
df = pd.read_csv(csv_data, delimiter=',')
df = df.dropna()

# Divide the data, y the variable to predict (label) and X the features
X = df[df.columns[1:]]
y = df['Outcome'].values.ravel()

# Splitting the data : training (80%) and test (20%)
X_train_, X_test_, y_train_, y_test_ = train_test_split(X, y, test_size=0.2,
random_state=42)
X_train_.shape

# Standardize the data
from sklearn.preprocessing import Normalizer
X_train_ = Normalizer().fit_transform(X_train_)

```

```

X_test_ = Normalizer().fit_transform(X_test_)

# Define number of classes and number of features to include in our model
number_of_classes = 2
number_of_features = X_train_.shape[1]

model_diab = Sequential()
model_diab.add(Flatten(input_dim=number_of_features))
keras_model.add(Dense(number_of_classes, activation='sigmoid'))
model_diab.compile(optimizer='adam', loss='binary_crossentropy')

model_diab.fit(X_train_, y_train_, epochs=5)
model_diab.evaluate(X_test_, y_test_) # loss, sparse_categorical_accuracy

predictions = model_diab.predict(X_test_)
print(predictions)
print('\n')

print(classification_report(y_test_,
                             tf.argmax(model_diab.predict(X_test_), axis=1).numpy()))

```

Output:

```

Epoch 1/5
20/20 [=====] - 0s 792us/step - loss: 1.6246
Epoch 2/5
20/20 [=====] - 0s 777us/step - loss: 1.6246
Epoch 3/5
20/20 [=====] - 0s 791us/step - loss: 1.6246
Epoch 4/5
20/20 [=====] - 0s 942us/step - loss: 1.6246
Epoch 5/5
20/20 [=====] - 0s 951us/step - loss: 1.6246
5/5 [=====] - 0s 1ms/step - loss: 1.7011

[[0.42503497 0.2515513 0.14312401 ... 0.00186495 0.18649493 0.      ]
 [0.7745625 0.5186802 0.22130357 ... 0.00102353 0.14523047 0.      ]
 [0.8246999 0.48871106 0.        ... 0.00120651 0.16035831 0.      ]
 ...
 [0.6926289 0.52493984 0.        ... 0.00353605 0.41557735 0.      ]
 [0.3662772 0.17561236 0.09533242 ... 0.00084545 0.07275369 0.00250875]
 [0.5651558 0.5346069 0.30548963 ... 0.00538425 0.2978524 0.      ]]

      precision    recall   f1-score   support
      0       0.69      0.78      0.73      99
      1       0.20      0.02      0.03      55
      3       0.00      0.00      0.00       0

accuracy                           0.51      154
macro avg       0.30      0.27      0.25      154
weighted avg    0.51      0.51      0.48      154

```

3.3 Support Vector Machine

A support vector machine (SVM) is a supervised learning algorithm that can be used for prediction of both binary variables (classification) and quantitative variables (regression problems), although it is primarily used for classification problems. The goal of SVM is to create a hyperplane that linearly divides n -dimensional data points into two components by searching

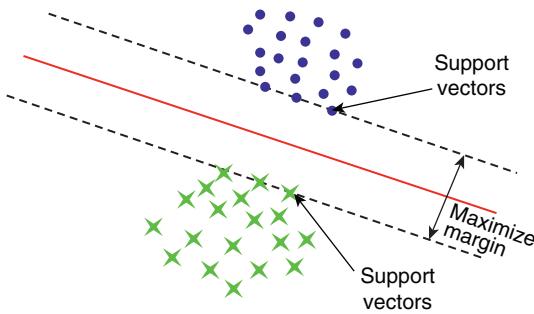


Figure 3.2 In SVM, we need to maximize the margin, which is defined by a subset of training samples, the support vectors. It is a quadratic programming problem that we can solve by standard methods.

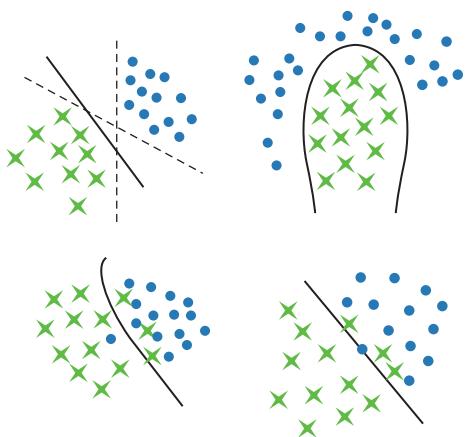


Figure 3.3 In SVM, we can face different binary discrimination issues.

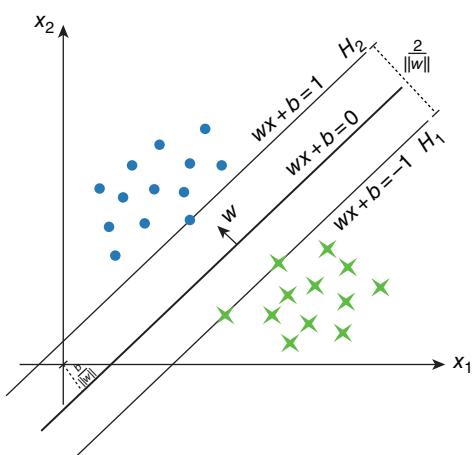


Figure 3.4 For $d = 2$, the data are linearly separable if we can draw a line separating the two classes in a graph of two dimensions (x_1 versus x_2).

for an optimal margin that correctly segregates the data into different classes and at the same time is separated as much as possible from all the observations. In addition to linear classification, it is also possible to compute a nonlinear classification using what we call the kernel trick (a kernel function) that maps inputs into high-dimensional feature spaces. The kernel function, when adapted to specific problems, allows flexibility to adapt to different situations. SVM allows creation of a classifier, or a discrimination function, that we can generalize and apply for predictions such as in image classification, diagnostics, genomic sequences, or drug discovery. SVM was developed at AT&T Bell Laboratories by Vladimir Vapnik and colleagues. To select the optimal hyperplane from among many hyperplanes that might classify our data, we select the one that has the largest margin (Figure 3.2) or, in other words, that represents the largest separation between the different classes. It is an optimization problem under constraints in which the distance between the nearest data point and the optimal hyperplane (on each side) is maximized. The optimal hyperplane is then called the maximum-margin hyperplane, allowing us to create a maximum-margin classifier. The closest data points are known as support vectors, and the margin is an area that generally does not contain any data points. If the optimal hyperplane is too close to the data points and the margin too small, it will be difficult to predict new data and the model will fail to generalize well. In nonlinear cases, we need to introduce a kernel function to search for nonlinear separating surfaces. The method induces a nonlinear transformation of our dataset toward an intermediate space that we call a feature space of higher dimension.

In this chapter, we will explore linearly and not fully linearly separable binary discrimination as well as nonlinear SVMs (Figure 3.3) and SVMs for regression.

3.3.1 Linearly Separable Data

Let us imagine that we have two sets of points. The first set is composed of blue points and the second one red points, visualized in a Euclidean plane. If the two sets are linearly separable, it means that at least one line in the plane separates the blue and the red points. If we have higher dimensional Euclidean spaces, the line is replaced by hyperplane. We will be considering a linear classifier for a binary classification problem with labels y and features x .

Let us say we have n training data points, where each input x_i is of dimensionality d and is included in one of two classes $y_i = -1$ or $+1$:

$$\{x_i, y_i\} \quad \text{where } i = [1, n], \quad y_i \in \{-1, 1\}, \quad x \in \mathbb{R}^d$$

If $d = 2$, the data are linearly separable if we can draw a line separating the two classes (Figure 3.4) in a graph of two dimensions (x_1 versus x_2). If $d > 2$, we refer to a hyperplane on graphs (x_1, x_2, \dots, x_n) that can be described by $w \cdot x + b = 0$ where w is a vector normal to the hyperplane, b is an offset, and $\frac{b}{\|w\|}$ is the perpendicular distance from the hyperplane to the origin.

The objective is to find the variables \mathbf{w} and b that describe our training data as follows:

$$\begin{cases} \mathbf{x}_i \cdot \mathbf{w} + b \geq +1, & \text{for } y_i = +1 \\ \mathbf{x}_i \cdot \mathbf{w} + b \leq -1, & \text{for } y_i = -1 \end{cases}$$

We can combine the two expressions as follows:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \quad \forall i$$

Let us consider two planes, H_1 and H_2 , and d_1 and d_2 , which are the distances between H_1 and the hyperplane and between H_2 and the hyperplane, respectively. The margin is the hyperplane's equidistance from H_1 and H_2 ($d_1 = d_2$).

If we consider only the points (vector support) that are closest to the separating hyperplane, then we can write the following:

$$\begin{cases} \mathbf{x}_i \cdot \mathbf{w} + b = +1, & \text{for } H_1 \\ \mathbf{x}_i \cdot \mathbf{w} + b = -1, & \text{for } H_2 \end{cases}$$

To find the optimal hyperplane, we need to orient the hyperplane to be as far from the support vectors as possible. Let us take a point x_0 and calculate the distance to the hyperplane: $\frac{|\mathbf{x}_0 \cdot \mathbf{w} + b|}{\|\mathbf{w}\|}$.

If x_0 is a point on the margin, the distance will be $\frac{|\pm 1|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$. The idea now is to maximize this distance, which is equivalent to finding the following:

$$\min \|\mathbf{w}\|, \quad \text{such that } y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \quad \forall i$$

We need to find a term to perform quadratic programming, which is the process of solving optimization problems using quadratic functions. Minimizing the term $\frac{1}{2} \|\mathbf{w}\|^2$ is equivalent to finding $\min \|\mathbf{w}\|$:

$$\min \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{such that } y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \quad \forall i$$

The above equation is called a primal optimization problem. Here, we face a constrained optimization problem that we need to convert into an unconstrained optimization problem. To solve it, we will use a classic method, the Lagrange multipliers. Because it is quadratic, the surface is a paraboloid with a single global minimum. Allocating Lagrange multipliers λ , we can write the following for SVM:

$$\begin{aligned} \mathcal{L}_P &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n \lambda_i [y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1] \\ &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n \lambda_i y_i (\mathbf{x}_i \cdot \mathbf{w} + b) + \sum_{i=1}^n \lambda_i \end{aligned}$$

such that $\forall i, \lambda_i \geq 0$, where n is the number of training points

\mathcal{L}_P is the primal form of the optimization problem. As we will see, we will use the dual form of this original problem (\mathcal{L}_D). From the property that the derivative at a minimum is equal to 0, we obtain the following:

$$\frac{\partial \mathcal{L}_P}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i$$

$$\frac{\partial \mathcal{L}_P}{\partial b} = \sum_{i=1}^n \lambda_i y_i = 0$$

Instead of minimizing over \mathbf{w} and b subject to constraints involving Lagrange multipliers λ , we can maximize over λ subject to relationships obtained previously for \mathbf{w} and b . We can eliminate the dependance on \mathbf{w} and b by substituting for \mathbf{w} and b back in the original equation $\min \mathcal{L}_P$:

$$\mathcal{L}_D(\lambda_i) = \sum_{i=1}^n \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad \text{such that } \sum_{i=1}^n \lambda_i y_i = 0 \text{ and } \lambda_i \geq 0$$

As we can see, the dual form requires only the dot product of each input vector x_i to be computed.

Let us express $\max \mathcal{L}_D(\lambda_i)$ as follows to simplify:

$$\begin{aligned}\mathcal{L}_D(\lambda_i) &= \sum_{i=1}^n \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i H_{ij} \lambda_j \quad \text{where } H_{ij} = y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \\ \mathcal{L}_D(\lambda_i) &= \sum_{i=1}^n \lambda_i - \frac{1}{2} \boldsymbol{\lambda}^T \mathbf{H} \boldsymbol{\lambda} \quad \text{such that } \sum_{i=1}^n \lambda_i y_i = 0 \text{ and } \lambda_i \geq 0\end{aligned}$$

We now need to maximize \mathcal{L}_D , which is a convex quadratic optimization challenge:

$$\max_{\boldsymbol{\lambda}} \left[\sum_{i=1}^n \lambda_i - \frac{1}{2} \boldsymbol{\lambda}^T \mathbf{H} \boldsymbol{\lambda} \right] \quad \text{such that } \sum_{i=1}^n \lambda_i y_i = 0 \text{ and } \lambda_i \geq 0$$

We can solve it using a quadratic programming solver that will output $\boldsymbol{\lambda}$ and allow us to calculate \mathbf{w} based on the following:

$$\frac{\partial \mathcal{L}_P}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i$$

The data points satisfying $\frac{\partial \mathcal{L}_P}{\partial b} = \sum_{i=1}^n \lambda_i y_i = 0$, which are the support vectors x_{SV} , will have the following form:

$$y_{SV}(\mathbf{x}_{SV} \cdot \mathbf{w} + b) = 1$$

We can substitute the expression above in $\mathbf{w} = \sum_{i=1}^n \lambda_i y_i \mathbf{x}_i$, yielding the following:

$$y_{SV} \left(\sum_{m \in S} \lambda_m y_m \mathbf{x}_m \cdot \mathbf{x}_{SV} + b \right) = 1 \quad \text{where } S \text{ is the set of indices of the support vectors}$$

The above equation can also be written as follows:

$$y_{SV}^2 \left(\sum_{m \in S} \lambda_m y_m \mathbf{x}_m \cdot \mathbf{x}_{SV} + b \right) = y_{SV} \implies b = y_{SV} - \sum_{m \in S} \lambda_m y_m \mathbf{x}_m \cdot \mathbf{x}_{SV}$$

This result is obtained because $y_{SV}^2 = 1$, as stated at the start:

$$\begin{cases} \mathbf{x}_i \cdot \mathbf{w} + b \geq +1, & \text{for } y_i = +1 \\ \mathbf{x}_i \cdot \mathbf{w} + b \leq -1, & \text{for } y_i = -1 \end{cases}$$

We can take an average over all the support vectors in S instead of using an arbitrary \mathbf{x}_{SV} :

$$b = \frac{1}{N} \sum_{SV \in S} \left(y_{SV} - \sum_{m \in S} \lambda_m y_m \mathbf{x}_m \cdot \mathbf{x}_{SV} \right)$$

We can then define the optimal hyperplane, as we were able to calculate \mathbf{w} and b .

In the end, we can predict the new points \mathbf{x}' by the evaluation of $y' = \text{sign}(\mathbf{w} \cdot \mathbf{x}' + b)$.

3.3.2 Not Fully Linearly Separable Data

In Section 3.3.1, we saw how to find an optimal hyperplane that separates two classes and at the same time is separated as much as possible from the closest data points (support vector). In other words, we have a hard margin in which no data points are allowed. We can encounter situations in which maintaining a hyperplane that is as far as possible from the closest data points can induce a narrow margin, producing a model that will be very sensitive to noisy data and not able to generalize. The possibility of overfitting can definitely increase. In general, real-world data are not fully separable. To address this challenge, we need to allow margin violation (Figure 3.5), meaning that we can keep a large margin and allow data points to be on the wrong side (misclassified points) or within the margin area (soft margin).

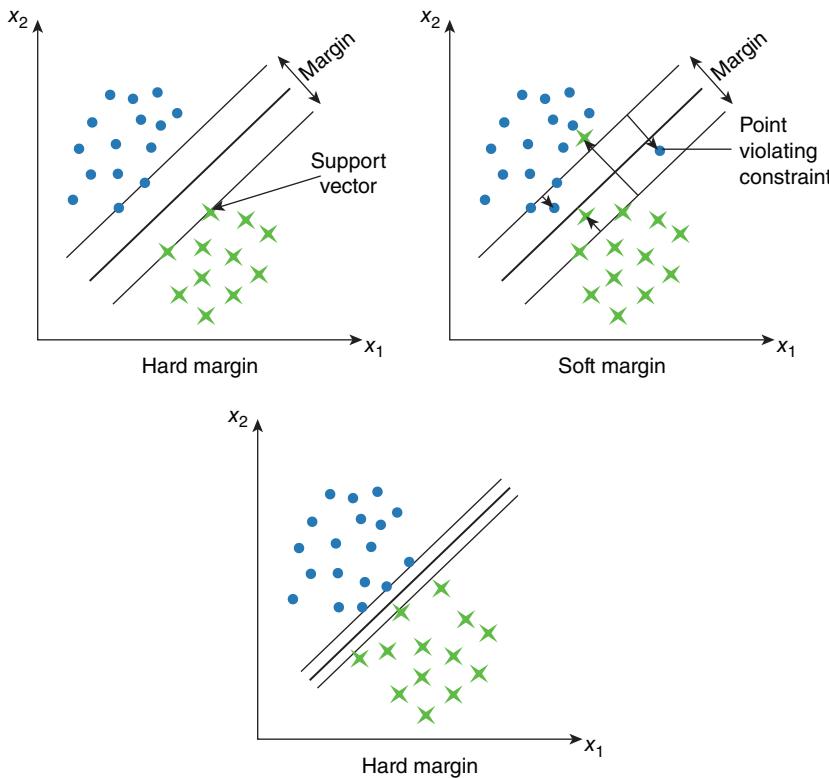


Figure 3.5 We need to make a trade-off between the width of the margin and the number of training errors committed by the linear decision boundary. A soft margin leads to underfitting whereas a hard margin leads to overfitting.

To allow data points to be on the wrong side or within the margin area, we can introduce a slack variable (Figure 3.6):

$$\begin{cases} \mathbf{x}_i \cdot \mathbf{w} + b \geq +1 - \xi_i, & \text{for } y_i = +1 \\ \mathbf{x}_i \cdot \mathbf{w} + b \leq -1 + \xi_i, & \text{for } y_i = -1 \end{cases}$$

where $\xi_i \geq 0 \forall i, i = 1, \dots, n$.

We can combine the two expressions as follows:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 + \xi_i \geq 0, \quad \text{where } \xi_i \geq 0 \forall i$$

In Section 3.3.1 discussing linearly separable data points, we explained that we need to minimize the following expression:

$$\min \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{such that } y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0, \quad \forall i$$

In a case that is not fully linearly separable, we can adapt the expression above by introducing the slack variable and a parameter C that controls the trade-off:

$$\min \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \quad \text{such that } y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 + \xi_i \geq 0, \quad \forall i$$

Allocating Lagrange multipliers ($\alpha_i \geq 0$ and $\beta_i \geq 0$), we can write the following:

$$\mathcal{L}_P = \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^n \xi_i - \sum_{i=1}^n \lambda_i [y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 + \xi_i] - \sum_{i=1}^n \beta_i \xi_i$$

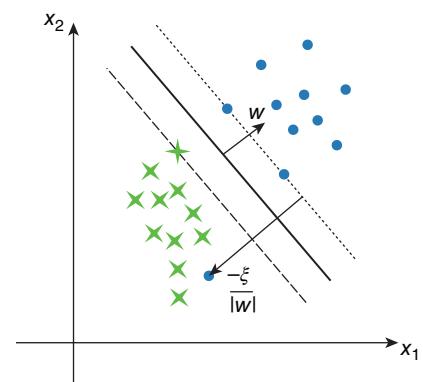


Figure 3.6 Introduction of a slack variable to separate two nonlinearly separable classes.

From the property that the derivative at a minimum is equal to 0, we obtain the following:

$$\frac{\partial \mathcal{L}_P}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n \lambda_i y_i x_i = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^n \lambda_i y_i x_i$$

$$\frac{\partial \mathcal{L}_P}{\partial b} = \sum_{i=1}^n \lambda_i y_i = 0$$

$$\frac{\partial \mathcal{L}_P}{\partial \xi_i} = 0 \Rightarrow C = \lambda_i + \beta_i$$

We now need to maximize \mathcal{L}_D and find $\forall i$:

$$\max_{\lambda} \left[\sum_{i=1}^n \lambda_i - \frac{1}{2} \lambda^T \mathbf{H} \lambda \right] \quad \text{such that} \quad \sum_{i=1}^n \lambda_i y_i = 0 \text{ and } 0 \leq \lambda_i \leq C$$

B is calculated as in the Section 3.3.1.

The new data points x' are classified by the evaluation of $y' = \text{sign}(\mathbf{w} \cdot \mathbf{x}' + b)$.

3.3.3 Nonlinear SVMs

The objective of nonlinear SVMs is to gain separation by mapping the data to a higher dimensional space because many classification or regression problems are not linearly separable or regressable in the space of the inputs x (Figure 3.7). For this, we use “kernel trick” to move to a higher dimensional feature space given a suitable mapping $x \rightarrow \phi(x)$.

In Section 3.1.1, we eliminated the dependance on \mathbf{w} and b by substituting for \mathbf{w} and b in the original equation for $\min \mathcal{L}_P$:

$$\mathcal{L}_D(\lambda_i) = \sum_{i=1}^n \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad \text{such that} \quad \sum_{i=1}^n \lambda_i y_i = 0 \text{ and } \lambda_i \geq 0$$

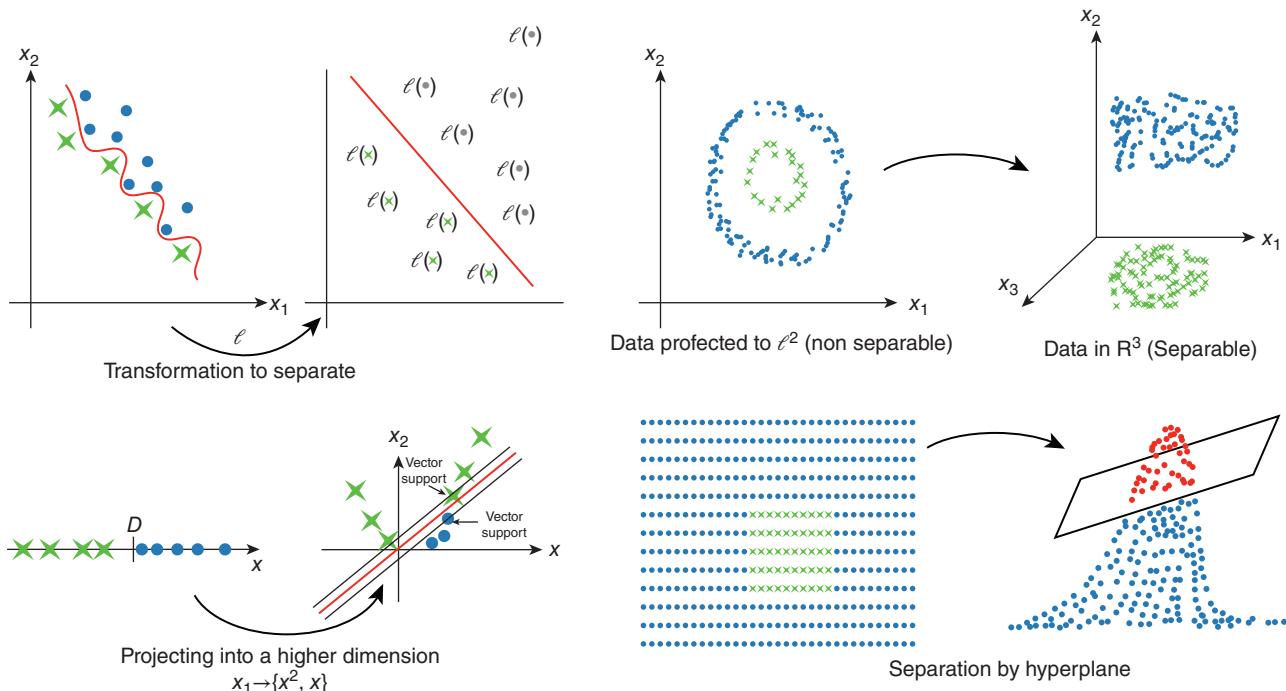


Figure 3.7 The objective of nonlinear SVMs is to gain separation by mapping the data to higher dimensional space because many classification or regression problems are not linearly separable or regressable in the space of the inputs x . For this, we use “kernel trick” to move to a higher dimensional feature space.

where $(\mathbf{x}_i \cdot \mathbf{x}_j)$ is the dot product of the two feature vectors. If we now transform to ϕ instead of calculating the dot product $(\mathbf{x}_i \cdot \mathbf{x}_j)$, we need to compute $(\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j))$, which can be very expensive and time consuming. If we introduce a kernel function K such that $K(x_i, x_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$, we do not need to calculate ϕ . The kernel functions allow us to have only inner products of the mapped inputs in the feature space determined without calculating ϕ .

There are many popular kernel functions that we can use such as the following:

- **Linear:** $K(x_i, x_j) = \mathbf{x}_i^T \mathbf{x}_j$
- **Gaussian radial basis function:** $K(x_i, x_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$
- **Polynomial:** $K(x_i, x_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + a)^b$
- **Sigmoidal:** $K(x_i, x_j) = \tanh(a\mathbf{x}_i \cdot \mathbf{x}_j - b)$
- **Laplacian:** $K(x_i, x_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|}{\sigma}}$
- **Rational quadratic:** $K(x_i, x_j) = 1 - \frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\|\mathbf{x}_i - \mathbf{x}_j\|^2 + c}$
- **Power:** $K(x_i, x_j) = \|\mathbf{x}_i - \mathbf{x}_j\|^d$
- **Log:** $K(x_i, x_j) = -\log \|\mathbf{x}_i - \mathbf{x}_j\|^d + 1$
- **Multiquadratic:** $K(x_i, x_j) = \sqrt{\|\mathbf{x}_i - \mathbf{x}_j\|^2 + c}$
- **Wave:** $K(x_i, x_j) = \frac{\theta}{\|\mathbf{x}_i - \mathbf{x}_j\|} \sin \frac{\|\mathbf{x}_i - \mathbf{x}_j\|}{\theta}$

Defining the proper kernel (Figure 3.8) will allow us to make the nonlinearly separable dataset in the data space \mathbf{x} separable in the nonlinear feature space defined implicitly by the chosen nonlinear kernel function.

The new points \mathbf{x}' are classified as follows: $\mathbf{y}' = \text{sign}(\mathbf{w} \cdot \phi(\mathbf{x}') + b)$.

3.3.4 SVMs for Regression

At the beginning of this section, we began with the training data points, where each input x_i is of dimensionality d and is included in one of two classes $y_i = -1$ or $+1$:

$$\{x_i, y_i\} \quad \text{where } i = [1, n], y_i \in \{-1, 1\}, x \in \mathbb{R}^d$$

We now wish to predict a real-valued output for y_i :

$$\{x_i, y_i\} \quad \text{where } i = [1, n], y_i \in \mathbb{R}, x \in \mathbb{R}^d$$

and $y_i = \mathbf{w} \cdot \mathbf{x}_i + b$.

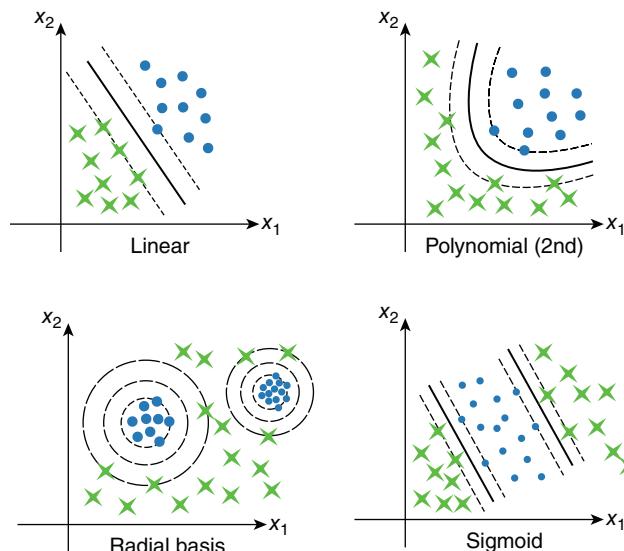
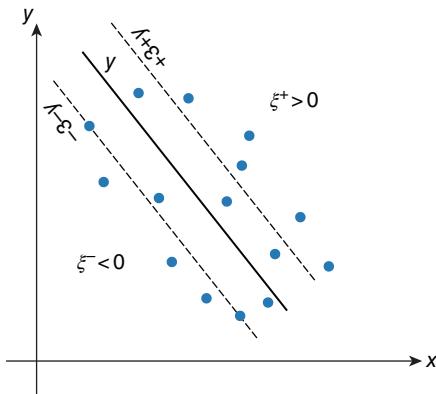


Figure 3.8 Defining the proper kernel will allow us to make the nonlinearly separable dataset in the data space \mathbf{x} separable in the nonlinear feature space defined implicitly by the chosen nonlinear kernel function.

Figure 3.9 Regression with an ϵ -insensitive tube.

For regression, it is necessary to use a penalty function implying that the prediction error is ignored if the difference between the predicted value y'_i and the actual value y_i is smaller than a distance ϵ (the ϵ -insensitive loss function or ϵ -insensitive tube):

$$y_i \leq y'_i + \epsilon + \xi^+$$

$$y_i \geq y'_i - \epsilon - \xi^-$$

The output variables outside the tube are given one of two slack variable penalties where $\forall i$, $\xi^+ > 0$ and $\xi^- > 0$. They are assigned depending on whether they lie above ξ^+ or below ξ^- in the tube (Figure 3.9).

The main goal remains to minimize error and to individualize the hyperplane to maximize the margin.

The error function that we need to minimize can be written as follows:

$$C \sum_{i=1}^n (\xi_i^+ + \xi_i^-) + \frac{1}{2} \|\mathbf{w}\|^2, \quad \text{with } \xi^+ \geq 0 \text{ and } \xi^- \geq 0, \forall i$$

As described above, we introduce Lagrange multipliers to minimize the function subject to the constraints:

$$\begin{aligned} \mathcal{L}_P = & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n (\xi_i^+ + \xi_i^-) - \sum_{i=1}^n (\alpha_i^+ \xi_i^+ + \alpha_i^- \xi_i^-) - \sum_{i=1}^n \lambda_i^+ (y'_i - y_i + \epsilon + \xi^+) \\ & - \sum_{i=1}^n \lambda_i^- (-y'_i + y_i + \epsilon + \xi^-) \end{aligned}$$

where $\forall i \alpha_i^+, \alpha_i^-, \lambda_i^+, \lambda_i^- \geq 0$.

From the property that the derivative at a minimum is equal to 0, we obtain the following:

$$\frac{\partial \mathcal{L}_P}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) \mathbf{x}_i = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) \mathbf{x}_i$$

$$\frac{\partial \mathcal{L}_P}{\partial b} = 0 \Rightarrow \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) = 0$$

$$\frac{\partial \mathcal{L}_P}{\partial \xi_i^+} = 0 \Rightarrow C = \lambda_i^+ + \alpha_i^+$$

$$\frac{\partial \mathcal{L}_P}{\partial \xi_i^-} = 0 \Rightarrow C = \lambda_i^- + \alpha_i^-$$

We now need to maximize \mathcal{L}_D :

$$\mathcal{L}_D = \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) y_i - \epsilon \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) - \frac{1}{2} \sum_{i,j} (\lambda_i^+ - \lambda_i^-) (\lambda_j^+ - \lambda_j^-) \mathbf{x}_i \cdot \mathbf{x}_j$$

where $\forall i, \lambda_i^+, \lambda_i^- \geq 0$.

$$\max_{\lambda^+, \lambda^-} \left[\sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) y_i - \epsilon \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) - \frac{1}{2} \sum_{i,j} (\lambda_i^+ - \lambda_i^-) (\lambda_j^+ - \lambda_j^-) \mathbf{x}_i \cdot \mathbf{x}_j \right]$$

where $\forall i, 0 \leq \lambda_i^+ \leq C, 0 \leq \lambda_i^- \leq C$ and $\sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) = 0$.

As previously stated, the following is true:

$$b = \frac{1}{N} \sum_{SV \in S} \left[y_{SV} - \epsilon - \sum_{m \in S} (\lambda_m^+ - \lambda_m^-) \mathbf{x}_m \cdot \mathbf{x}_S \right]$$

The new points \mathbf{x}' are classified as follows:

$$y' = \sum_{i=1}^n (\lambda_i^+ - \lambda_i^-) \mathbf{x}_i \cdot \mathbf{x}' + b$$

3.3.5 Application of SVMs

If we go to hephAIstos/Notebooks with a terminal and open a Jupyter Notebook, we will see in a browser a file named SVM.ipynb that contains all code examples shown in this section. The notebook can also be downloaded here: <https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/SVM.ipynb>.

In this section, we will apply SVMs to the quantitative characterization of neuronal morphologies from histological neuronal reconstructions, which represent a primary resource to investigate anatomical comparisons and morphometric analyses. It is still unclear how many classes of cortical neurons exist despite a century of research on cortical circuits. Morphology-based classification of neuron types in the entire brain, such as that of the rat, remains a challenge as it is not clear how to designate a neuronal cell given the significant number of neuron types, limited samples (reconstructed neuron), uncertainty regarding the best features by which to define them, and diverse data formats such as 2D and 3D images (structured with high dimensions and fewer samples than the complexity of morphologies) or SWC-format files (low dimensional and unstructured).

There are several reasons that neuroscientists are interested in this topic. Some brain diseases affect specific cell types; for example, amyotrophic lateral sclerosis and congenital nystagmus affect upper and lower motor neurons, respectively. The idea is to improve our knowledge by correlating disorders with vulnerable neuronal types. Neuron morphology studies can lead to the identification of genes so that we can target specific morphology and functions linked to genes. The discovery of gene functions associated with specific cell types can be used as entry points. Before acquiring its form and function, a neuron goes through different stages of development that need to be understood to identify new markers, marker combinations, or mediators of developmental choices. The understanding of neuron morphologies often represents the basis of modeling efforts or data-driven modeling approaches to study the impact of a cell's morphology on, for instance, its electrical behavior and on the network in which it is embedded.

The brain is composed of an enormous number of neurons and an even greater number of synapses with a huge diversity. Therefore, the classification of types of neurons is a way to reduce the dimensionality for modeling objectives. We have access to digitally reconstructed neurons by species, brain regions, and cell types in NeuroMorpho.org, which is one of the largest public neuronal morphology databases. The database contains more than 170,000 cells distributed in more than 300 brain regions. Tools such as L-Measure (<http://cng.gmu.edu:8080/Lm/help/index.htm>) and NeuroM (<https://neurom.readthedocs.io/en/stable/index.html>) already exist for use by researchers to extract quantitative morphological measurements from neuronal reconstructions. The reconstructions are usually obtained from brightfield or fluorescence microscopy preparations or can be synthesized via computation simulations.

For this section, we will extract the features from 29,321 labeled rat neurons. The data contain two main classes (principal cell and interneuron cell) and 14 subclasses, including six types of principal cells (ganglion, granule, medium spiny, parachromaffin, Purkinje, and pyramidal cell), six types of interneurons (basket, chandelier, martinotti, double bouquet, bitufted, and nitrergic), and two types of glial cells (microglia and astrocyte).

Principal cells:

- Ganglion: 317
- Granule: 1130
- Medium spiny: 867
- Parachromaffin: 525
- Purkinje: 497
- Pyramidal: 14002

Interneurons:

- Basket: 503
- Chandelier: 26
- Martinotti: 137

- Double bouquet: 50
- Bitufted: 67
- Nitrergic: 2044

Glial cells:

- Microglia: 7549
- Astrocyte: 1607

Let us apply SVM techniques to these data. We have been referring so far to binary classification. Natively, even though SVM does not support multi-class classification in its simplest form, we can use it for multi-class classification by applying the same principles after breaking down the multi-classification problem into multiple binary classification problems. A first approach in the process, called one-to-one, is to map data points to high-dimensional space to gain mutual linear separation between every pair of classes. A second approach is called one-to-rest, in which breakdown is set to a binary classifier for each class. Fortunately, most of the data science frameworks perform it automatically.

3.3.5.1 SVM Using scikit-learn for Classification

In the code below, we build an SVM with a linear kernel on the neuron morphology data. We split the data into training and testing datasets and scale them separately. After fitting the model, we calculate accuracy metrics and perform fivefold cross-validation (we return the mean and standard deviation).

Input:

```
# Importing libraries
import pandas as pd
from sklearn import preprocessing
from sklearn import metrics
from sklearn import svm
from sklearn.model_selection import cross_val_score

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/neurons.csv'
df = pd.read_csv(csv_data, delimiter=';')

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)
```

```

model.fit(X_train,y_train)
y_pred = model.predict(X_test)

# Metrics
results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score(y_test,
y_pred, average='micro'), metrics.recall_score(y_test, y_pred, average='micro'),
metrics.f1_score(y_test, y_pred, average='micro'), cross_val_score(model, X_train,
y_train, cv=5).mean(), cross_val_score(model, X_train, y_train, cv=5).std()]
metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
{'SVM_linear'})

metrics_dataframe

```

Output:

SVM_linear	
Accuracy	0.801471
Precision	0.801471
Recall	0.801471
F1 Score	0.801471
Cross-validation mean	0.934126
Cross-validation std	0.002855

To change the kernel of the SVM, we simply modify the following line:

```

# Model
model=svm.SVC(kernel='linear')

```

We specify the kernel we want to apply (linear, poly, rbf, sigmoid, precomputed):

```

# Model
model=svm.SVC(kernel=sigmoid)

```

Applying different kernels produces the results below, showing the impact of each of them on the performance of the model:

	SVM_linear	SVM_rbf	SVM_sigmoid	SVM_poly
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Missing Method	knn	knn	knn	knn
Extraction Method	no	no	no	no
Accuracy	0.820921	0.877466	0.702092	0.706515
Precision	0.820921	0.877466	0.702092	0.706515
Recall	0.820921	0.877466	0.702092	0.706515
F1 Score	0.820921	0.877466	0.702092	0.706515
Cross-validation mean	0.931697	0.897622	0.695635	0.65044
Cross-validation std	0.00287	0.004224	0.023332	0.012177

Another process that influences the results is the method we use to scale the data, as shown in the results below from the same data:

Table 1

	SVM_linear	SVM_rbf	SVM_sigmoid	SVM_poly
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Cross-validation mean	0.9316968373424395	0.8976222184070262	0.695635293154068	0.6504401169970137
Cross-validation std	0.002870155539182848	0.004223770340391114	0.02333189380870366	0.012176651244570208
Rescaling Method	MinMaxScaler	MinMaxScaler	MinMaxScaler	MinMaxScaler
Cross-validation mean	0.5321787333618948	0.8022650451733219	0.5155256780198503	0.8156899327125414
Cross-validation std	0.0018448744583600223	0.006499403438239211	0.003081145772619621	0.0038418247028325424
Rescaling Method	MaxAbsScaler	MaxAbsScaler	MaxAbsScaler	MaxAbsScaler
Cross-validation mean	0.5341257828152367	0.8773311198195588	0.4509116043564639	0.6785709447965624
Cross-validation std	0.0020671231990968455	0.005860662335322936	0.008919049705886691	0.005208272883512474
Rescaling Method	RobustScaler	RobustScaler	RobustScaler	RobustScaler
Cross-validation mean	0.9459927971547737	0.7855602352960943	0.6694520529605221	0.5114264904466089
Cross-validation std	0.003002315262914603	0.01159626250222911	0.03029509002456383	0.0031256795486157667
Rescaling Method	Normalizer	Normalizer	Normalizer	Normalizer
Cross-validation mean	0.8349047916062885	0.8443328980859615	0.7720841313743526	0.8484832757903756
Cross-validation std	0.001227064700311557	0.001645181264870277	0.0042375356979946635	0.0016026801491159843
Rescaling Method	Yeo-Johnson	Yeo-Johnson	Yeo-Johnson	Yeo-Johnson
Cross-validation mean	0.9527052671757328	0.946966446575354	0.6748306000428422	0.9257017510700049
Cross-validation std	0.0030022938794538384	0.0038296910578962456	0.009438727986960285	0.004699974937488666
Rescaling Method	Quantile-Gaussian	Quantile-Gaussian	Quantile-Gaussian	Quantile-Gaussian
Cross-validation mean	0.9509120112523786	0.9464540464879896	0.6661710936942159	0.9048983311492211
Cross-validation std	0.0008849004355420398	0.003632661544663354	0.007657803589551836	0.005689724212400542
Rescaling Method	Quantile-Uniform	Quantile-Uniform	Quantile-Uniform	Quantile-Uniform
Cross-validation mean	0.937538261341633	0.943020908149679	0.3306515479895667	0.9453267216684937
Cross-validation std	0.003121844825257883	0.0034275832442342553	0.005610577459301739	0.003054572741896653

3.3.5.2 SVM Using scikit-learn for Regression

To use SVMs for regression, we can proceed with the use of SVMs for classification but instead input the following:

```
model=svm.SVR(kernel='linear')
```

The above code corresponds to linear support vector regression. We can replace the kernel with the other options (poly, rbf, sigmoid, etc.).

3.4 Artificial Neural Networks

Artificial neural networks (ANNs) have become a central concept in the field of modern machine learning, addressing a full range of complex problems in classification, regression, image processing, forecasting, speech recognition, NLP, and other applications. They are inspired by the functionality of the human brain and were first introduced by McCulloch and Pitts to model a biological neuron. The idea behind neural networks is that a network of neurons can be constructed by connecting multiple neurons together such that the output of one neuron forms an input to another. Different types of architectures exist for neural networks. The oldest and most simple model is the MLP introduced by Rosenblatt. Convolutional neural networks (CNN), which are particularly suited for image processing, have been developed recently. We can also cite another famous neural network architecture, recurrent neural networks, which can be used for sequential data that occur in time series or text. As we will see, in an ANN we have an input x and an output $y = f(x, \theta)$ where the parameters θ are estimated from a learning sample. As is usual in statistical learning, we need to minimize a function that is not convex, implying local minimizers. Cybenko (1989) and Hornik (1991) have proposed a universal approach, whereas Le Cun (1989) has introduced backpropagation to compute the gradient of a neural network.

Let us consider the following artificial neuron:

$$y_j = f_j(x) = \phi\left(\sum_{i=1}^d w_{j,i}x_i + b_j\right) = \phi(\langle w_j, x \rangle + b_j)$$

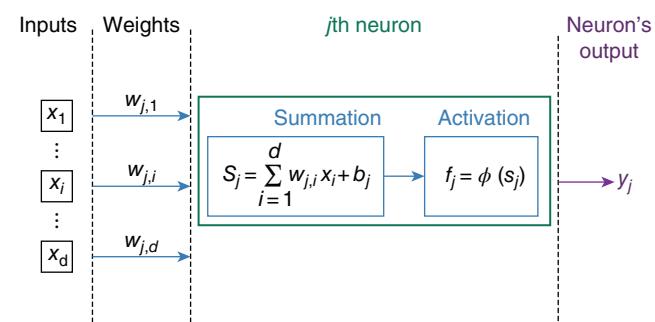
The artificial neuron is represented by the function f_j . This function has an input $x = (x_1, \dots, x_d)$, weighted by a vector of connection weights $w_j = (w_{j,1}, \dots, w_{j,d})$. We also have neuron bias b_j . $\sum_{i=1}^d w_{j,i}x_i + b_j$ called the summation. In addition, we have an activation function ϕ , specifically $\phi\left(\sum_{i=1}^d w_{j,i}x_i + b_j\right)$. We need to consider activation functions such as the identity function $\phi(x) = x$ or the sigmoid $\phi(x) = \frac{1}{1 + e^{-x}}$.

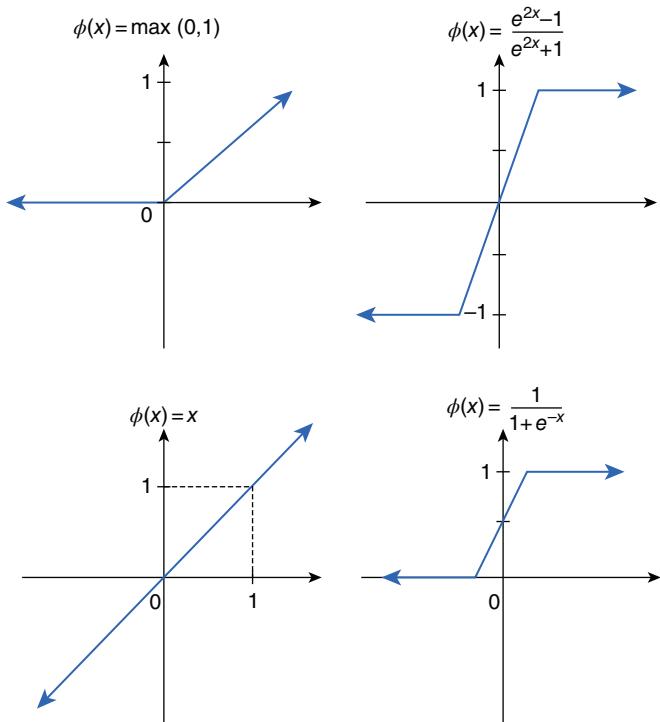
Figure 3.10 presents a schematic representation of an artificial neuron.

As stated above, several activation functions can be considered (Figure 3.11):

- **Identity function:** $\phi(x) = x$
- **Sigmoid function:** $\phi(x) = \frac{1}{1 + e^{-\beta x}}$
- **Rectified linear unit (ReLU):** $\phi(x) = \max(0, x)$
- **Hard threshold:** $\phi_\beta(x) = \mathbf{1}_{x \geq \beta} = \begin{cases} 0 & \text{if } x < \beta \\ 1 & \text{if } x \geq \beta \end{cases}$
- **Hyperbolic tangent (tanh):** $\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$
- **Piecewise linear:** $\phi(x) = \begin{cases} 0 & \text{if } x \leq x_{\min} \\ mx + b & \text{if } x_{\max} > x > x_{\min} \\ 1 & \text{if } x \geq x_{\max} \end{cases}$
- **Gaussian:** $\phi(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$

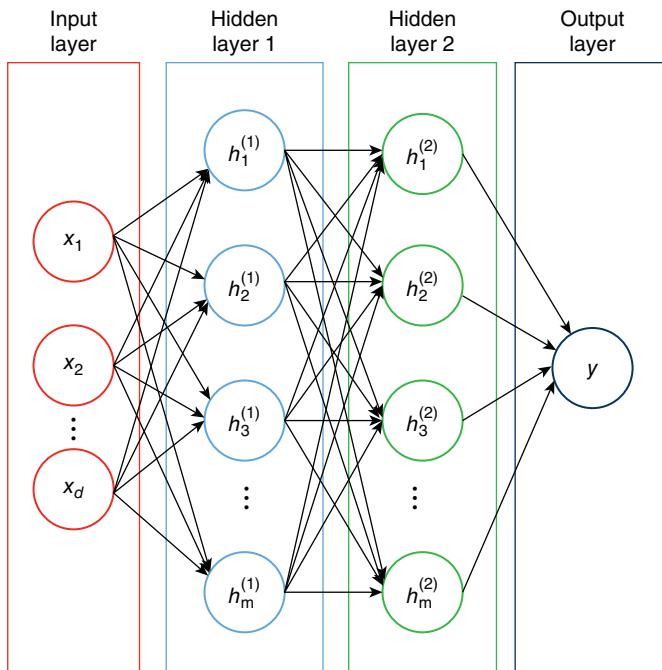
Figure 3.10 Schematic representation of an artificial neuron.



**Figure 3.11** Activation function examples.

3.4.1 Multilayer Perceptron

A multilayer perceptron (MLP), or neural network, is formed by connecting one neuron to every other neuron. In order to do this, neural networks are connected to each other through layers. A neural network is composed of several hidden layers of neurons, in which every neuron in a layer is connected to neurons in the next layer as represented in Figure 3.12. Specifically

**Figure 3.12** Schematic representation of neural network layers.

in an MLP, there are no links between neurons inside the same layer, but each neuron of a layer is linked to all neurons of the next layer. The output of a neuron in a hidden layer becomes the input of another neuron in the next layer. The last layer is called the output layer. Depending on the problem we are addressing, either classification or regression, we can apply a different activation function in the last hidden layer. For regression, no activation function is applied. In fact, we apply the identity function, but it does nothing. To run a neural network, we will need to choose a certain number of parameters such as the number of hidden layers, the number of neurons in each layer, the activation function, and the activation function of the last hidden layer. In binary classification problems, each output unit implements a threshold function for which the output value can be, for example, 0 or 1 depending on a prediction $P(Y = 1/X)$ that has generated a value between 0 and 1 and to which we apply a threshold. For binary classification, we can use the sigmoid activation function, for instance, because its output is a value between 0 and 1. For multi-class, we place one neuron per class (i) in the output layer, making the sum of each prediction $P(Y = i/X)$ equal to 1. In this case, we can use the softmax function.

Let us write out the MLP mathematically. As we will see, we will need to choose a notation and apply the same statistical learning concepts that we have seen previously. Let us say that x_i are the input units ($i = 1, \dots, d$), y is the output unit, and h_j^k represents the units in the k th hidden layer.

We set $h^{(0)} = x$ and establish the following:

$$\begin{aligned} h_j^k(x) &= \phi\left(\sum_i w_{ji}^{(k)} h_j^{k-1}(x) + b_j^{(k)}\right) && \text{with } k = 1, \dots, L \text{ (hidden layers)} \\ y_j &= \psi\left(\sum_i w_{ji}^{(k)} h_j^{k-1}(x) + b_j^{(k)}\right) && \text{with } k = L + 1 \text{ (output layer)} \end{aligned}$$

where ϕ is the activation function and ψ is the activation function of the output layer.

We can write the vectorized form as follows:

$$\begin{aligned} \mathbf{h}^k &= \phi(\mathbf{W}^{(k)} \mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}) && \text{with } k = 1, \dots, L \text{ (hidden layers)} \\ \mathbf{y} &= \psi(\mathbf{W}^{(k)} \mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}) && \text{with } k = L + 1 \text{ (output layer)} \end{aligned}$$

where $\mathbf{W}^{(k)}$ is the weight matrix with the number of rows being the number of neurons in the layer k and the number of columns being the number of neurons in the layer $(k - 1)$, $\mathbf{h}^{(k)}$ is an activation vector, and $\mathbf{b}^{(k)}$ is a bias vector present in each layer.

The universal approximation theorem states that feed-forward neural networks with as few as one hidden layer are universal approximators. In other words, a neural network with one hidden layer can approximate any continuous function for inputs that do not have large gaps, meaning that they are within a specific range. One of the first versions was demonstrated by George Cybenko (1989) for a sigmoid activation function.

3.4.2 Estimation of the Parameters

3.4.2.1 Loss Functions

As in other machine learning algorithms, neural network attempts to learn the probability distribution underlying the given data observations. The weights w_j and the biases b_j must be estimated from a learning sample, which is usually obtained by minimizing a chosen loss function with a gradient descent algorithm. In general, we estimate maximum likelihood (or equivalently the logarithm of the likelihood), which means that we attempt to find a set of parameters and a prior probability distribution to build a model that represents the distribution over the data. Cross-entropy-based loss functions, the difference between two probability distributions, also referred to as the negative log likelihood, are commonly used in classification. The resulting difference between the two probability distributions is called loss. The idea is to minimize the loss function, which is the opposite of the log likelihood.

For binary classification, binary cross-entropy is appropriate. The binary cross-entropy for a set of true and predicted labels can be calculated as follows:

$$L = -\frac{1}{m} \sum_{j=1}^m y_j \log(\hat{y}_j) + (1 - y_j) \log(1 - \hat{y}_j)$$

where y_j represents the expected outcome (actual value of the j th sample), \hat{y}_j represents the outcome produced by our model (predicted value of the j th sample), and m is the number of samples. It can be expressed using another notation:

$$L(\theta) = -\frac{1}{m} \sum_{j=1}^m y_j \log(f(X, \theta)) + (1-y_j) \log(1-f(X, \theta))$$

where θ is the vector of parameters to estimate, $f(X, \theta) = p_\theta(Y = 1/X)$, and $Y \in \{0, 1\}$.

For a multi-class classification problem, we consider a generalization of the previous loss function applied to k classes (maximum likelihood estimate):

$$L = -\sum_{j=1}^k y_j \log(\hat{y}_j)$$

In the case of regression settings, we use the mean squared error. The formula of the loss is the squared difference between the expected value and the predicted value:

$$L = \frac{1}{2} \sum_j (y_j - \hat{y}_j)^2$$

where j is the index for an input-output pair.

As we have seen previously, even if different algorithms exist to minimize the loss function, stochastic gradient descent is a classic choice:

$$\theta_j = \theta_j - \epsilon \frac{\partial}{\partial \theta_j} L(\theta)$$

where ϵ is the learning rate that we need to calibrate for the algorithm to converge. We could also use Adaptive Moment Estimation (Adam) or other methodologies to minimize the loss function.

3.4.2.2 Backpropagation: Binary Classification

Looking at history, Minsky and Papert (1969) showed that restrictions in neural networks can be overcome with a two-layer feed-forward network, but there was the challenge of how to adjust the weights from input to hidden units. This is how the idea of backpropagation arose (Parker 1985; Rumelhart et al. 1986; Werbos 1974). Gradient descent and its alternatives help to minimize the cost function. Backpropagation, which is short for “backward propagation of errors,” is a widely used algorithm for training feed-forward neural networks that computes the partial derivatives of a cost function L with respect to the parameters and uses it for optimizers such as stochastic gradient descent. Given a neural network and an error function, we can calculate the gradient of the error function with respect to the neural network’s weights and biases. The backpropagation algorithm computes the gradient one layer at a time and iterates backward from the last layer. This procedure allows elimination of redundant calculations of intermediate terms in the chain rule. In fact, because hidden layer nodes do not have a target output, we cannot define an error function for this specific node itself. Any error function for that node is dependent on the values of the parameters in the previous layers, input for that specific node, and the following layers as the output of that specific node will impact the calculation of the error function. It leads to very complicated math and can make the calculation of stochastic gradient descent very slow. This is where backpropagation makes a difference by simplifying the math and facilitating calculation of the gradient descent.

To understand backpropagation better, let us consider its use in the case of a binary classification problem.

As we have seen, for binary classification, we can work with the following loss function (binary cross-entropy):

$$L = -\frac{1}{m} \sum_{i=1}^m y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i)$$

where m is the number of samples, y_i is the actual output of the i th sample, and \hat{y}_i is the predicted output of the i th sample.

We need to find the gradients of the loss function with respect to weight and bias:

$$\frac{\partial L}{\partial W_j} = \frac{\partial}{\partial W_j} \left[-\frac{1}{m} \sum_{i=1}^m y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i) \right] = -\frac{1}{m} \sum_{i=1}^m \left[\left[\frac{y_i}{\hat{y}_i} - \frac{(1-y_i)}{(1-\hat{y}_i)} \right] \frac{\partial \hat{y}_i}{\partial W_j} \right]$$

Let us take the sigmoid function as the activation function: $\phi(x) = \frac{1}{1 + e^{-x}}$.

The following equation can be derived:

$$\frac{\partial \hat{y}_i}{\partial W_j} = \frac{\partial}{\partial W_j} \left[\frac{1}{1 + e^{-z_i}} \right]$$

where $z_i = W^T x_i + b$.

$$\frac{\partial \hat{y}_i}{\partial W_j} = -\frac{1}{(1 + e^{-z_i})^2} \times (-e^{-z_i}) \times \frac{\partial z_i}{\partial W_j} = \hat{y}_i(1 - \hat{y}_i) \frac{\partial z_i}{\partial W_j}$$

By substituting the expression in:

$$\begin{aligned} \frac{\partial L}{\partial W_j} &= -\frac{1}{m} \sum_{i=1}^m \left[\left[\frac{y_i}{\hat{y}_i} - \frac{(1-y_i)}{(1-\hat{y}_i)} \right] \left[\hat{y}_i(1 - \hat{y}_i) \frac{\partial z_i}{\partial W_j} \right] \right] = \frac{1}{m} \sum_{i=1}^m \left[\frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \right] \left[\hat{y}_i(1 - \hat{y}_i) \frac{\partial z_i}{\partial W_j} \right] = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i) \frac{\partial z_i}{\partial W_j} \\ &= \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i) x_{ij} \end{aligned}$$

where x_{ij} is the j th feature of the i th input sample.

$$\begin{aligned} \frac{\partial L}{\partial W_j} &= \frac{1}{m} [(\hat{y}_1 - y_1)x_{1j} + \dots + (\hat{y}_m - y_m)x_{mj}] = \frac{1}{m} x_{j1 \times m} [\vec{\hat{y}} - \vec{y}]_{m \times 1}^T = \frac{1}{m} \begin{bmatrix} x_{11 \times m} [\vec{\hat{y}} - \vec{y}]_{m \times 1}^T \\ \dots \\ x_{n1 \times m} [\vec{\hat{y}} - \vec{y}]_{m \times 1}^T \end{bmatrix} \\ &= \frac{1}{m} \vec{x}_{n \times m} [\vec{\hat{y}} - \vec{y}]_{m \times 1}^T \end{aligned}$$

where $\vec{x}_{n \times m}$ is the input matrix of m samples and n features.

With the same approach, we can now find a gradient of the loss function with respect to bias:

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i) \frac{\partial z_i}{\partial b} = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)$$

As $z_i = W^T x_i + b$ and then $\frac{\partial z_i}{\partial b} = 1$.

The stochastic gradient descent is defined as follows:

$$\theta_j = \theta_j - \epsilon \frac{\partial}{\partial \theta_j} L(\theta)$$

After calculation of the gradients, we can update the weights and bias with stochastic gradient descent:

$$W_j = W_j - \epsilon \frac{\partial}{\partial W_j} L$$

$$b = b - \epsilon \frac{\partial}{\partial b} L$$

3.4.2.3 Backpropagation: Multi-class Classification

Let us consider the case of multi-class classification. As we have seen, in the multi-class classification context, we can consider the following loss function applied to K classes:

$$L = - \sum_{j=1}^K y_j \log(\hat{y}_j)$$

We can recall the following equation:

$$h_j^k(x) = \phi \left(\sum_i w_{ji}^{(k)} h_j^{k-1}(x) + b_j^{(k)} \right) \quad \text{with } k = 1, \dots, L \text{ (hidden layers)}$$

And let us introduce $a_j(x)$ as follows:

$$a_j(x) = \sum_{i=1}^d w_{ji}^{(k)} h_j^{(k-1)}(x) + b_j^{(k)}$$

Let us consider the following function:

$$f(x) = \begin{pmatrix} \mathbb{P}(Y = 1/x) \\ \dots \\ \mathbb{P}(Y = K/x) \end{pmatrix}$$

We will also use the multidimensional function *softmax* as the activation function:

$$\text{softmax}(x_1, \dots, x_K) = \frac{1}{\sum_{j=1}^K e^{x_j}} (e^{x_1}, \dots, e^{x_K})$$

If we calculate the gradient, we obtain the following:

$$\frac{\partial \text{softmax}(\mathbf{x})_i}{\partial x_j} = \begin{cases} \text{softmax}(\mathbf{x})_i (1 - \text{softmax}(\mathbf{x})_i), & \text{if } i = j \\ -\text{softmax}(\mathbf{x})_i \text{softmax}(\mathbf{x})_j, & \text{if } i \neq j \end{cases}$$

Let us also introduce $(f(x))_j$, which is the j th element of $f(x)$ such that the following is true:

$$(f(x))_y = \sum_{j=1}^K \mathbf{1}_{y=j} (f(x))_j$$

where $(f(x))_k = \mathbb{P}(Y = \frac{k}{x})$. If we take the logarithm of the expression above, we obtain the following for the loss function L :

$$L(f(x), y) = -\log (f(x))_y = \sum_{j=1}^K \mathbf{1}_{y=j} \log (f(x))_j = -\sum_{j=1}^K y_j \log (\hat{y}_j)$$

The idea now is to compute the gradients according to weights and biases in both the output and hidden layers:

- The hidden weights and biases:

$$\frac{\partial L(f(x), y)}{\partial W_{ij}^{(h)}}, \frac{\partial L(f(x), y)}{\partial b_i^{(h)}}$$

- The output weights and biases:

$$\frac{\partial L(f(x), y)}{\partial W_{ij}^{(L+1)}}, \frac{\partial L(f(x), y)}{\partial b_i^{(L+1)}}$$

where $h \in (1, \dots, L)$.

If we use the chain rule:

$$\text{if } z(x) = \phi(a_1(x), \dots, a_J(x))$$

Then:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^J \frac{\partial z}{\partial a_j} \frac{\partial a_j}{\partial x_i}$$

Hence, we obtain the following:

$$\begin{aligned}
\frac{\partial L(f(x), y)}{\partial (a^{(L+1)}(x))_i} &= \sum_j \frac{\partial L(f(x), y)}{\partial f(x)_j} \frac{\partial f(x)_j}{\partial (a^{(L+1)}(x))_i} = - \sum_j \frac{\mathbf{1}_{y=j}}{(f(x))_y} \frac{\partial \text{softmax}(a^{(L+1)}(x))_j}{\partial (a^{(L+1)}(x))_i} \\
&= - \frac{1}{(f(x))_y} \frac{\partial \text{softmax}(a^{(L+1)}(x))_y}{\partial (a^{(L+1)}(x))_i} \\
&= - \frac{1}{(f(x))_y} \text{softmax}\left(a^{(L+1)}(x)\right)_y (1 - \text{softmax}\left(a^{(L+1)}(x)\right)_y) \mathbf{1}_{y=i} \\
&\quad + \frac{1}{(f(x))_y} \text{softmax}\left(a^{(L+1)}(x)\right)_i \text{softmax}\left(a^{(L+1)}(x)\right)_y \mathbf{1}_{y \neq i} \\
&= \left(-1 + f(x)_y\right) \mathbf{1}_{y=i} + f(x)_i \mathbf{1}_{y \neq i}
\end{aligned}$$

We finally produce the following statement:

$$\nabla a^{(L+1)}(x) L(f(x), y) = f(x) - e(y)$$

where $y \in \{1, \dots, K\}$ and $e(y)$ is the \mathbb{R}^K vector with i^{th} component $\mathbf{1}_{i=y}$.

We can now compute the gradients of the loss function according to weights and biases in the output layers.

For the output bias:

$$\nabla b^{(L+1)} L(f(x), y) = f(x) - e(y) \quad \text{as} \quad \frac{\partial a^{(L+1)}(x)_j}{\partial (b^{(L+1)})_i} = \mathbf{1}_{i=j}$$

If we use the chain rule, we produce the following:

$$\begin{aligned}
\frac{\partial L(f(x), y)}{\partial W_{ij}^{(L+1)}} &= \sum_k \frac{\partial L(f(x), y)}{\partial (a^{(L+1)}(x))_k} \frac{\partial (a^{(L+1)}(x))_k}{\partial W_{ij}^{(L+1)}} = \sum_k \frac{\partial L(f(x), y)}{\partial (a^{(L+1)}(x))_k} \left(a^{(L)}(x)\right)_j \mathbf{1}_{i=k} \\
\nabla W^{(L+1)} L(f(x), y) &= (f(x) - e(y)) \left(a^{(L)}(x)\right)'
\end{aligned}$$

We can also compute the gradients of the loss function according to weights and biases in the hidden layers.

As usual, we use the chain rule as follows:

$$\frac{\partial L(f(x), y)}{\partial (h^{(k)}(x))_j} = \sum_i \frac{\partial L(f(x), y)}{\partial (a^{(k+1)}(x))_i} \frac{\partial a^{(k+1)}(x)_i}{\partial (h^{(k)}(x))_j} = \sum_i \frac{\partial L(f(x), y)}{\partial (a^{(k+1)}(x))_i} W_{ij}^{(k+1)}$$

We can now summarize as follows:

$$\nabla h^{(k)}(x) L(f(x), y) = \left(W^{(k+1)}\right)' \nabla a^{(k+1)}(x) L(f(x), y)$$

where

$$\nabla a^{(k)}(x) L(f(x), y) = \nabla h^{(k)}(x) L(f(x), y) \odot \left(\phi' \left(a^{(k)}(x)_1, \dots, \phi' \left(a^{(k)}(x)_{j,\dots}\right)\right)'\right)$$

The gradient to the loss function with respect to hidden weights produces the following:

$$\frac{\partial L(f(x), y)}{\partial W_{ij}^{(k)}} = \frac{\partial L(f(x), y)}{\partial (a^{(k)}(x))_i} \frac{\partial (a^{(k)}(x))_i}{\partial W_{ij}^{(k)}} = \frac{\partial L(f(x), y)}{\partial (a^{(k)}(x))_i} h_j^{(k-1)}(x)$$

We finally obtain this statement:

$$\nabla W^{(k)} L(f(x), y) = \nabla a^{(k)}(x) L(f(x), y) h^{(k-1)}(x)'$$

We can now calculate the gradient with respect to the hidden biases:

$$\frac{\partial L(f(x), y)}{\partial b_i^{(k)}} = \frac{\partial L(f(x), y)}{\partial a^{(k)}(x)_i}$$

We finally arrive at the following:

$$\nabla b^{(k)} L(f(x), y) = \nabla a^{(k)}(x) L(f(x), y)$$

Thus, backpropagation provides a way to compute gradients that will serve for the stochastic gradient descent, which updates the parameters of a model to minimize a loss function by using gradients of the loss function with respect to the parameters. Backpropagation avoids repeating calculations by computing gradients one layer at a time.

Here, we have seen the case of a multi-class classification problem. We can also calculate the gradient to the loss function with respect to weights and biases in both the output and hidden layers in binary classification and regression problems by following the same procedure and taking the corresponding loss functions.

3.4.3 Convolutional Neural Networks

MLPs are not suitable for some types of data such as images, because the process requires transformation of the images into vectors and therefore eliminates information such as spatial data. Convolutional neural networks (CNNs), introduced by the French scientist Yan LeCun, allow us to avoid the manual feature extraction procedure and work directly on matrices or tensors with three RGB color channels; they are now widely used for image processing (classification segmentation, recognition, etc.).

A digital image can be considered a matrix of numbers in which each number corresponds to the brightness of a pixel. In the RGB model, we have three matrices (red, green, blue) storing values from 0 to 255 (Figure 3.13). In images that are black and white, only a single matrix is needed.

CNNs are composed of several layers: **convolutional layers, pooling layers, and fully connected layers**. Let us begin to understand convolution layers. Mathematically, convolution is an operation in which two functions (f and g) produce a third function $f * g$ that expresses the modification of the shape of one by the other.

The discrete convolution between f and g is defined as follows:

$$(f * g) = \sum_t f(t)g(x + t)$$

For 2D signals for which we apply a kernel K to a 2D signal I , we have the following:

$$(K * I)(i, j) = \sum_{m, n} K(m, n)I(i + n, j + m)$$

A kernel convolution used in computer vision algorithms and also in CNNs is the process where we pass a small matrix of number (the kernel or filter) over our image and transform it based on the values from filter. The input image can be denoted X and our filter f , given the expression $X * f$.

To better understand the process, let us consider an image of size 3×3 and a filter of size 2×2 (Figure 3.15).

The filter is passed over our main image and performs an element-wise multiplication such as the following (Figure 3.16):

$$(10 \times 1 + 5 \times 0 + 8 \times 1 + 2 \times 0) = 18$$

$$(5 \times 1 + 6 \times 0 + 2 \times 1 + 14 \times 0) = 7$$

$$(8 \times 1 + 2 \times 0 + 2 \times 1 + 3 \times 0) = 10$$

$$(2 \times 1 + 14 \times 0 + 3 \times 1 + 4 \times 0) = 5$$

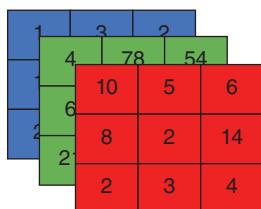


Figure 3.13 Schematic representation of an input image (3×3 pixels) using the RGB model in which we have three matrices (red, green, blue) storing values from 0 to 255.

For an image of dimensions $n \times n$ and a filter of dimensions $f \times f$, the output will be of dimensions $(n - f + 1) \times (n - f + 1)$.

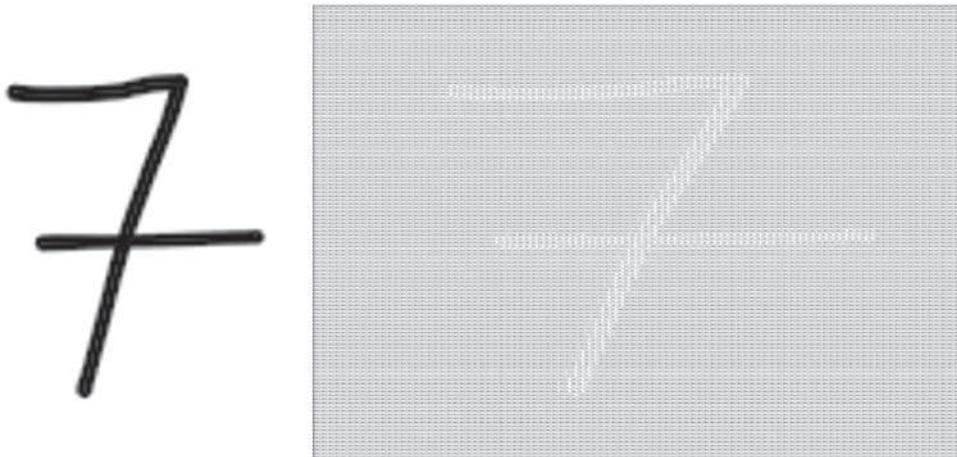


Figure 3.14 A digital image, such as this picture of a handwritten “7,” can be considered as a matrix of numbers in which each number corresponds to the brightness of a pixel.

After seeing the convolution layers, let us approach fully connected layers. As explained above, the convolution layer extracts features from original data and generates a two-dimensional matrix (Figure 3.14). We send these features to a fully connected layer, a traditional neural network, that generates the final output. The fully connected layer can only work with one-dimensional data; therefore, we need to convert our two-dimensional matrix into a one-dimensional format (Figure 3.16). The fully connected layer will perform two operations on the input data that we have described:

- A linear transformation: $Z = W^T + b$
 - A nonlinear transformation with the activation function

10	5	6
8	2	14
2	3	4

Figure 3.15 An image of size 3×3 and a filter of size 2×2

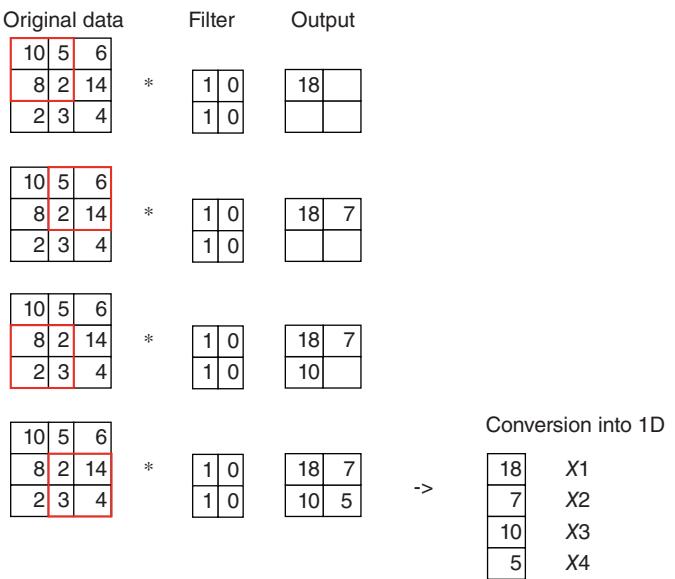


Figure 3.16 The filter is passed over our main image and performs an element-wise multiplication.

We can summarize the steps as follows:

$$\begin{aligned}
 \text{Input} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_d \end{bmatrix} &\rightarrow \text{Convolution } C_1 = X * f \rightarrow \text{Activation function} = A_1(C_1) \\
 &\rightarrow \text{Linear transformation } Z_1 = W^T A_1 + b \rightarrow \text{Output} = A_1(Z_1) \rightarrow \text{Output}
 \end{aligned}$$

where A_1 could be a sigmoid function. The goal is now to determine the values in the filter that will be randomly initialized by the CNN and learning during the training process. In fact, we apply the same process as a traditional neural network but we add filters, meaning that we randomly initialize the weights, biases, and filters and update the parameters by using the gradient descent technique, in which we calculate the change in error with respect to both weights and biases: $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. After calculation of the gradients, we can update the weights and bias with stochastic gradient descent:

$$W_j = W_j - \epsilon \frac{\partial}{\partial W_j} L$$

$$b = b - \epsilon \frac{\partial}{\partial b} L$$

For the convolution layer, we had a filter matrix parameter. We calculate the gradients of $\frac{\partial L}{\partial f}$ and update parameters as follows:

$$f = f - \epsilon \frac{\partial}{\partial f} L$$

3.4.4 Recurrent Neural Network

Recurrent neural networks (RNNs) are well suited for sequential data such as text or time series. An RNN is not so different from a traditional neural network; we can see it as multiple copies of the same network, each passing information to its successor (Figure 3.17).

RNNs are widely used for speech recognition from an input audio clip, sentiment classification to predict the number of stars a service will be given according to the feedback provided, DNA sequence analysis, machine translation, etc.

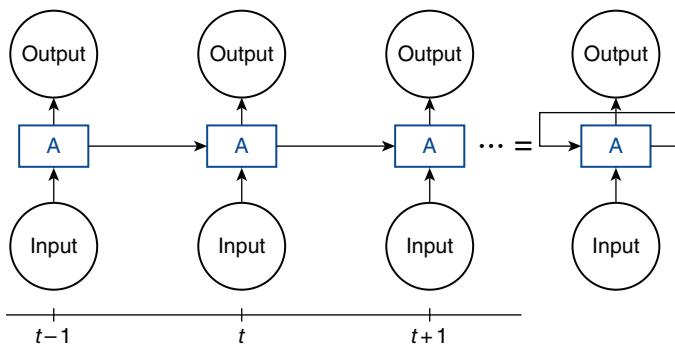


Figure 3.17 An RNN with certain inputs at $(t - 1)$ that will lead to outputs at time $(t - 1)$. At the next timestamp, the information at $(t - 1)$ is provided along with the input at time t , eventually providing an output at time t as well. This process is repeated through all the timestamps in the model.

Let us consider the weight matrix w and the bias b . At time t_0 and input x_0 , we need to find h_0 such that the following is true:

$$h^{(t)} = \phi(w_i x^{(t)} + w_R h^{(t-1)} + b_n)$$

Then, we need to calculate y_0 according to the following formula:

$$y^{(t)} = \psi(w_y h^{(t)} + b_y)$$

This process is repeated through all the timestamps (Figure 3.18).

RNNs use backpropagation to train, but it is applied for every timestamp. This process is commonly called backpropagation through time (BTT).

All these applications that we have seen as examples above are due to the performance of a particular RNN called long short-term memory (LSTM), which can learn long-term dependencies. LSTM cells were introduced by Hochreiter and Schmidhuber (1997). An LSTM cell includes at time t , a state C_t , and an output h_t . This cell receives inputs from x_t , C_{t-1} , and h_{t-1} . Inside the LSTM, the computations are defined by doors that either allow the transmission of information or do not. The computations are performed by equations described by Hochreiter and Schmidhuber.

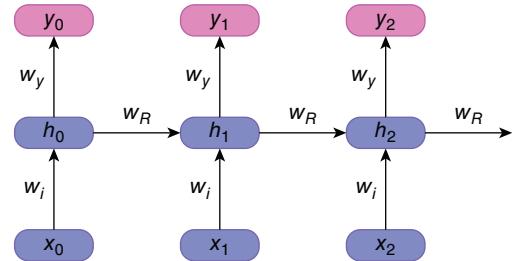


Figure 3.18 An RNN with weights w and bias b .

3.4.5 Application of MLP Neural Networks

If we go to hephAIstos/Notebooks with a terminal and open a Jupyter Notebook, we will see in a browser a file named `Neural_Networks.ipynb` that contains all code examples shown in this section. The notebook can also be downloaded here: https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Neural_Networks.ipynb.

An initial example is to use scikit-learn to implement a MLP neural network using `MLPClassifier` to classify data. `MLPClassifier` trains using backpropagation. For regression, we can replace `MLPClassifier` with `MLPRegressor`.

In this sample, we will import our data (neuron morphologies, as described above), split the data into training and test datasets, and separate features (`X`, `X_train`, `X_test`) from labels (`y`, `y_train`, `y_test`). We will apply cross-validation. The MLP neural network takes the following inputs:

- **max_iter:** The maximum number of iterations (default = 200).
- **hidden_layer_sizes:** The i th element represents the number of neurons in the i th hidden layer.
- **activation:** The activation function for the hidden layer (“identity,” “logistic,” “relu,” “softmax,” “tanh”; default = “relu”).
- **solver:** The solver for weight optimization (“lbfgs,” “sgd,” “adam”; default = “adam”).
- **alpha:** Strength of the l2 regularization term (default = 0.0001).
- **learning_rate:** The learning rate schedule for weight updates (“constant,” “invscaling,” “adaptive”; default = “constant”).
- **learning_rate_init:** The initial learning rate used (for sgd or adam). It controls the step size in updating the weights.

The application below provides us with a dataframe for the accuracy score (the ratio of the number of correct predictions to all predictions made by the classifier), the precision score (the number of correct outputs or how many of the correctly predicted cases turned out to be positive), the recall score (how many of the actual positive cases we were able to predict correctly), the f1-score (the harmonic mean of precision and recall), and the cross-validation score (mean and standard deviation).

Input:

```
# Importing libraries
import pandas as pd
from sklearn import preprocessing
from sklearn import metrics
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/neurons.csv'
df = pd.read_csv(csv_data, delimiter=';')

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Parameters and Hyperparameters
cv=5
max_iter = 400
hidden_layer_sizes = 10
activation = 'relu'
solver = 'sgd'
alpha = 0.0001
learning_rate = 'constant'
learning_rate_init = 0.0001

model = MLPClassifier(max_iter = max_iter, hidden_layer_sizes = hidden_layer_sizes,
activation = activation, solver = solver, alpha = alpha, learning_rate =
learning_rate, learning_rate_init = learning_rate_init)
```

```

model.fit(X_train,y_train)
y_pred = model.predict(X_test)

results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score(y_test,
y_pred, average='micro'), metrics.recall_score(y_test, y_pred, average='micro'),
metrics.f1_score(y_test, y_pred, average='micro'), cross_val_score(model, X_train,
y_train, cv=cv).mean(), cross_val_score(model, X_train, y_train, cv=cv).std()]
metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
{'MLP_neural_network'})

print('Classification Report for MLP Neural Network:\n')
print(classification_report(y_test,y_pred))
metrics_dataframe

```

Output:

Classification Report for MLP Neural Network:

	precision	recall	f1-score	support
astrocytes	0.80	0.52	0.63	322
basket	0.57	0.36	0.44	91
bitufted	0.00	0.00	0.00	13
chandelier	0.00	0.00	0.00	5
double_bouquet	0.00	0.00	0.00	13
ganglion	0.00	0.00	0.00	72
granule	0.67	0.05	0.10	187
martinotti	0.00	0.00	0.00	30
medium_spiny	0.00	0.00	0.00	176
microglia	0.67	0.97	0.79	1247
nitrergic	0.61	0.67	0.64	400
parachromaffin	0.92	0.68	0.78	114
purkinje	0.00	0.00	0.00	97
pyramidal	0.86	0.91	0.88	2809
accuracy			0.77	5576
macro avg	0.36	0.30	0.31	5576
weighted avg	0.72	0.77	0.73	5576

MLP_neural_network	
Accuracy	0.773852
Precision	0.773852
Recall	0.773852
F1 Score	0.773852
Cross-validation mean	0.799821
Cross-validation std	0.007810

To define the best hyperparameters automatically, we can use *GridSearchCV* (from *sklearn.model_selection import GridSearchCV*).

Input:

```

# Importing libraries
import pandas as pd
from sklearn import preprocessing
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/neurons_binary.csv'
neuron = pd.read_csv(csv_data, delimiter=';')

df = neuron.head(22).copy()
df = pd.concat([df, neuron.iloc[17033:17053]])

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Instantiate the estimator
mlp_gs = MLPClassifier()
parameter_space = {
    'hidden_layer_sizes': [(10,30,10),(20,)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['sgd', 'adam', 'lbfgs'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant','adaptive','invscaling'],
}
# Hyperparameters search
from sklearn.model_selection import GridSearchCV
model = GridSearchCV(mlp_gs, parameter_space, n_jobs=-1, cv=5)
# Fit the estimator to the data
model.fit(X_train, y_train) # X is train samples and y is the corresponding labels
# Use the model to predict the last several labels
y_pred = model.predict(X_test)

```

```

results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score(y_test,
y_pred, average='micro'), metrics.recall_score(y_test, y_pred, average='micro'),
metrics.f1_score(y_test, y_pred, average='micro'), cross_val_score(model, X_train,
y_train, cv=cv).mean(), cross_val_score(model, X_train, y_train, cv=cv).std()]
metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
{'mlp_neural_network_auto'})

print('Classification Report for MLP Neural Network Auto:\n')
print(classification_report(y_test,y_pred))

metrics_dataframe

```

Output:

Classification Report for MLP Neural Network Auto:

	precision	recall	f1-score	support
interneurons	1.00	0.80	0.89	5
principal	0.80	1.00	0.89	4
accuracy			0.89	9
macro avg	0.90	0.90	0.89	9
weighted avg	0.91	0.89	0.89	9

mlp_neural_network_auto	
Accuracy	0.888889
Precision	0.888889
Recall	0.888889
F1 Score	0.888889
Cross-validation mean	0.880952
Cross-validation std	0.108588

The Keras Python library for deep learning focuses on the creation of models as a sequence of layers. In the example below, we will code a simple MLP neural network using Keras from TensorFlow by defining a sequential model and specifying all the layers.

As stated previously, we will need the following inputs:

- The first layer of our model that specifies the shape of the input (input_dim).
- The weight initialization, specifically “uniform,” in which weights are initialized to small uniformly random values between 0 and 0.05, “normal,” in which weights are initialized to small Gaussian random values, or “zero,” in which weights are set to zero values.
- The activation functions such as softmax, sigmoid, linear, or tanh.
- The layer types such as dense, dropout, or concatenate. Dense is the type most often used for MLP models with fully connected layers.
- The model optimizers such as stochastic gradient descent, Adam, or RMSprop.
- The loss functions such as the mean squared error (MSE), the binary logarithmic loss (binary_crossentropy), or the multi-class logarithmic loss (categorical_crossentropy).

Let us code an example with neurons classified as principal cells or interneurons (a binary classification problem).

Input:

```

# Importing libraries
import pandas as pd
import numpy as np
import tensorflow as tf

from sklearn import preprocessing
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report

# load data: Capture the dataset in Python using Pandas DataFrame
csv_data = '../data/datasets/neurons_binary.csv'
neuron = pd.read_csv(csv_data, delimiter=';')
# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
neuron[['Target']] = neuron[['Target']].apply(enc.fit_transform)

# Select a subset of the data
df = neuron.head(300).copy()
df = pd.concat([df, neuron.iloc[17033:17333]])

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import SGD
from sklearn.metrics import classification_report
from tensorflow.keras.losses import binary_crossentropy

# Print number of GPUs available
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU'))))

```

```

# For future use, which devices the operations and tensors are assigned to (GPU, CPU)
#tf.debugging.set_log_device_placement(True)

# Parameters and Hyperparameters of the model
gpu_mlp_activation = 'sigmoid'
gpu_mlp_optimizer= SGD(learning_rate = 1e-2)
gpu_mlp_epochs = 10
gpu_mlp_loss = 'binary_crossentropy'

# Define number of classes and number of features to include in our model
number_of_classes = df.groupby('Target').count().shape[0]
number_of_features = X_train.shape[1]

# Model
keras_model = Sequential()
keras_model.add(Dense(number_of_classes, activation=gpu_mlp_activation))
keras_model.add(Flatten(input_dim=number_of_features))
keras_model.compile(optimizer = gpu_mlp_optimizer,
                     loss = 'binary_crossentropy',
                     metrics = [gpu_mlp_loss])
keras_model.fit(X_train, y_train, epochs=gpu_mlp_epochs)
keras_model.evaluate(X_test, y_test) # loss, sparse_categorical_accuracy

# Predicting X_test data with the created model
y_keras_pred = keras_model.predict(X_test)
y_keras_test = np.argmax(y_keras_pred, axis=1) #Make labels back

from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score

print("Classification report for multi-layer perceptron using GPUs (if available)\n")
print(classification_report(y_test, y_keras_test))
print("\n")

# Compute and print predicted output with X_test as new input data
print('Print predicted output with X_test as new input data \n')
print('\n')
print('Predictions: \n', y_keras_test)
print('\n')
print('Real values: \n', y_test)
print('\n')

results = [accuracy_score(y_test, y_keras_test), precision_score(y_test,
y_keras_test,average='micro'), recall_score(y_test, y_keras_test,average='micro'),
f1_score(y_test, y_keras_test,average='micro')]
metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score"], columns={'gpu_mlp'})

metrics_dataframe

```

Output:

```
Classification report for multi-layer perceptron using GPUs (if available)
```

	precision	recall	f1-score	support
0	0.76	0.72	0.74	57
1	0.76	0.79	0.78	63
accuracy			0.76	120
macro avg	0.76	0.76	0.76	120
weighted avg	0.76	0.76	0.76	120

```
Print predicted output with X_test as new input data
```

Predictions:

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

Real values:

```
110      1
287      1
17298     0
77       1
181      1
...
17131     0
17166     0
148       1
17227     0
17171     0
Name: Target, Length: 120, dtype: int64
```

gpu_mlp	
Accuracy	0.758333
Precision	0.758333
Recall	0.758333
F1 Score	0.758333

If we wish to use a MLP neural network for regression data, the code is similar. We define the function to create the baseline model. The difference is that we do not use an activation function for the output layer.

Below is an example of a function we could create for a regression problem.

Input:

```
def gpu_mlp_regression(X, X_train, X_test, y, y_train, y_test, cv, gpu_mlp_epochs_r,
gpu_mlp_activation_r):

    """
    Multi-Layer perceptron using GPU for regression

    Inputs:
        X,y non splitted dataset separated by features (X) and labels (y). This is used
        for cross-validation
        X_train, y_train selected dataset to train the model separated by features
        (X_train) and labels (y_train)
        X_test, y_test: selected dataset to test the model separated by features
        (X_test) and labels (y_test)
        cv: number of k-folds for cross-validation
        gpu_mlp_epochs: The number of epochs (integer)
        gpu_mlp_activation_r: The activation function such as softmax, sigmoid, linear
        or tanh.

    Output:
        A DataFrame with the following metrics:
        - Root mean squared error (MSE)
        - R2 score

    """

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from sklearn.metrics import classification_report

# Print number of GPUs available
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

# Define of features to include in our model
number_of_features = X_train.shape[1]

# Model creation
keras_model = Sequential()
keras_model.add(Dense(number_of_features, input_shape=(number_of_features,), kernel_initializer='normal', activation=gpu_mlp_activation_r))
keras_model.add(Dense(1, kernel_initializer='normal'))
keras_model.compile(loss='mean_squared_error', optimizer='adam')

# Model prediction
y_pred = keras_model.predict(X_train)

# Extracting the weights and biases is achieved quite easily
keras_model.layers[0].get_weights()
```

```

# We can save the weights and biases in separate variables
weights = keras_model.layers[0].get_weights()[0]
bias = keras_model.layers[0].get_weights()[1]

mse = mean_squared_error(y_train, y_pred)
r2 = r2_score(y_train, y_pred)

# Compute and print predicted output with X_test as new input data
print("\n")
print('Print predicted output with X_test as new input data \n')
print('\n')
print('Predictions: \n', keras_model.predict(X_test))
print('\n')
print('Real values: \n', y_test)
print('\n')

# Printing metrics
print("MLP for Regression Metrics on GPU \n")
print('Root mean squared error: ', mse)
print('R2 score: ', r2)
print("Intercept:", bias)
print("Weights:", weights)
print('\n')

results = [mse, r2]
metrics_dataframe = pd.DataFrame(results, index=["MSE", "R-squared"], columns=['gpu_mlp_regression'])

return metrics_dataframe

```

3.4.6 Application of RNNs: LST Memory

To implement an RNN such as LSTM, we can proceed as in any other project by using Pandas and NumPy for data manipulation, scikit-learn for scaling and evaluation, and TensorFlow for modeling. Keras provides LSTM or gated recurrent unit (GRU) modules to build the model. As we have done for the other models, we will provide a few parameters and hyperparameters such as an activation function, the input size of the model, the number of units for LSTM (more will yield better results), an optimizer, and a loss function.

In the example below, we will use the weather forecasting dataset containing data from 2013 to 2017 in Delhi, India, with five features: date, meantemp, humidity, wind_speed, and meanpressure. First, we load the data from a .csv file, drop lines with missing values, and create new columns from the date feature that we have split into year, month, and day. We also use meantemp as our target to predict. We have split the data into training and test datasets and rescaled it (StandardScaler).

We define the following parameters and hyperparameters:

- **Units:** A positive integer representing the dimensionality of the output space.
- The activation function.
- The number of features.
- The optimizer.
- The loss function.
- The number of epochs

Input:

```

import tensorflow as tf

from sklearn import preprocessing
from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, mean_squared_error, r2_score

import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from tensorflow.keras.optimizers import SGD

# Load data
DailyDelhiClimateTrain = '../data/datasets/DailyDelhiClimateTrain.csv'
df = pd.read_csv(DailyDelhiClimateTrain, delimiter=',')

# define time format
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')

# create a DataFrame with new columns (year, month and day)
df['year']=df['date'].dt.year
df['month']=df['date'].dt.month
df['day']=df['date'].dt.day

# Delete column 'date'
df.drop('date', inplace=True, axis=1)

# Rename column meantemp to Target
df = df.rename(columns={"meantemp": "Target"})

# Drop row having at least 1 missing value
df = df.dropna()

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

```

```

# Parameters and Hyperparameters of the model
rnn_units = 500 # Positive integer, dimensionality of the output space.
rnn_activation = 'tanh' # Activation function to use.
rnn_features = X_train.shape[1] # Number of features
rnn_optimizer= 'RMSprop' # Optimizer
rnn_loss = 'mse' # Loss function
rnn_epochs = 50 # Number of epochs

# The LSTM architecture
model_lstm = Sequential()
model_lstm.add(LSTM(units=rnn_units, activation=rnn_activation, input_shape =
(rnn_features, 1)))
model_lstm.add(Dense(units=1))
# Compiling the model
model_lstm.compile(optimizer=rnn_optimizer, loss=rnn_loss)
model_lstm.fit(X_train, y_train, epochs=rnn_epochs)

# Model prediction
y_pred = model_lstm.predict(X_train)

# Extracting the weights and biases is achieved quite easily
model_lstm.layers[0].get_weights()

# We can save the weights and biases in separate variables
weights = model_lstm.layers[0].get_weights()[0]
bias = model_lstm.layers[0].get_weights()[1]

mse = mean_squared_error(y_train, y_pred)
r2 = r2_score(y_train, y_pred)

# Compute and print predicted output with X_test as new input data
print("\n")
print('Print predicted output with X_test as new input data \n')
print('\n')
print('Predictions: \n', model_lstm.predict(X_test))
print('\n')
print('Real values: \n', y_test)
print('\n')

# Printing metrics
print("RNN on GPU \n")
print('Root mean squared error: ', mse)
print('R2 score: ', r2)
print("Intercept:", bias)
print("Weights:", weights)
print('\n')

results = [mse, r2]
metrics_dataframe = pd.DataFrame(results, index=["MSE", "R-squared"], columns=
{'RNN'})

metrics_dataframe

```

Output:

```
Epoch 1/50
37/37 [=====] - 3s 27ms/step - loss: 133.7000
Epoch 2/50
37/37 [=====] - 1s 30ms/step - loss: 46.5097
Epoch 3/50
37/37 [=====] - 1s 30ms/step - loss: 45.7743
Epoch 4/50
37/37 [=====] - 1s 30ms/step - loss: 46.2194
Epoch 5/50
37/37 [=====] - 1s 33ms/step - loss: 44.2782
```

```
Epoch 49/50
37/37 [=====] - 2s 62ms/step - loss: 4.0243
Epoch 50/50
37/37 [=====] - 2s 60ms/step - loss: 4.9160
```

Print predicted output with `X_test` as new input data

Predictions:

```
[[35.005676]
[14.977965]
...
[13.91688]
[31.52404]]
```

Real values:

```
892      35.875000
1106     18.000000
413      15.250000
522      38.500000
1036     24.000000
...
1362     31.240000
802      21.500000
651      24.500000
722      9.875000
254      31.166667
```

Name: Target, Length: 293, dtype: float64

RNN on GPU

```
Root mean squared error: 3.426273762485222
R2 score: 0.9364344926733309
Intercept: [[-0.05915993  0.04514894  0.05786226 ...  0.09068689 -0.03867587
  0.04705746]
[-0.06163495 -0.00744027  0.00724653 ...  0.01702344  0.01215462
 -0.02267423]]
```

```
[ 0.08192775  0.00580381 -0.01737174 ... -0.00572672 -0.00982367
 -0.03814929]
...
[ 0.00308989  0.03659     0.00428902 ... -0.01632125 -0.02669154
 0.01698471]
[ 0.1659518   0.04997669  0.11947034 ...  0.15972894 -0.03002974
 0.09136766]
[-0.10103803 -0.00107457 -0.04552393 ... -0.13199584 -0.01182847
 -0.04028853]]
Weights: [[-0.11194573 -0.09326565 -0.0154289   ... -0.14395078  0.5907008
 0.6180739 ]]
```

RNN

MSE	3.426274
R-squared	0.936434

3.4.7 Building a CNN

An excellent way to classify images is to build a CNN. In the example below, we will use the popular mnist dataset, which is composed of 70,000 images (60,000 for training and 10,000 for testing) of handwritten digits from 0 to 9. Our goal will be to identify each image using a model created with the Keras library.

Let us first download the dataset and display an image from it.

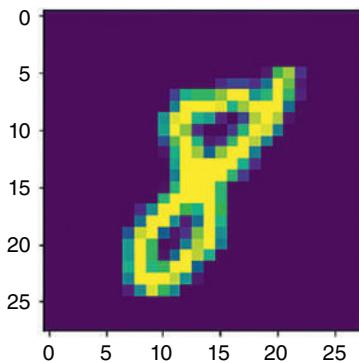
Input:

```
from keras.datasets import mnist
import matplotlib.pyplot as plt

# Download mnist data and split into train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Display one image
plt.imshow(x_train[17])
```

Output:



To create our convolutional neural network model, we will need to perform some data preprocessing, such as reshaping the data, and perform one hot encoding. We will then build our model, which means setting the parameters and hyperparameters such as activation functions, the optimizer, and the loss function, training the model, and finally using the model to predict new data.

In our model, we will reshape the data to fit the model with 60,000 images for training and 10,000 for testing. The images have a size of 28×28 , and we will set the image as greyscale. We will use one hot encoding for the target column (`y_train` and `y_test`), which means that we will create a column for each output category. For example, for an image with the number “3,” we will have `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`. We will then create our model using `Sequential()` from Keras to build the model layer by layer. We will create two convolutional layers (`Conv2D`), the first one with 60 nodes and the second one with 30 nodes. These numbers can be adjusted. Depending on the size of the dataset, the numbers can be higher or lower. We will also use `relu` as the activation function for our first two layers and `softmax` for the last one. The size of the filter matrix for the convolution will be set to 3 (3×3). A flattened layer is added between the convolutional and dense layers to connect both of them. The dense layer is our output layer. The output will be series of arrays with probabilities because we have chosen the softmax activation function for the last layer. Summing the arrays will yield 1. Taking the highest probability will give us the predicted number.

Input:

```

import tensorflow as tf
from keras.datasets import mnist
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten

# Download mnist data and split into train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Display one image
plt.imshow(X_train[8])

# Check image shape
X_train[8].shape

# Here we reshape the data to fit model with 60000 images for training, image size is
# 28x28
# 1 means that the image is greyscale (one channel).
# If we want to use RGB values (a color image), 3 can be used.
X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)

# One-hot encode target column
# A column will be created for each output category.
# For example, for an image with the number 2 we will have [0,1,0,0,0,0,0,0,0,0]
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

print(y_train[8])

# Creation of the model using Sequential() to build a model layer by layer
model = Sequential()

# Adding model layers
# The first two layers are convolutional layers (Conv2D) dealing with 2D matrices
# We set the first layer with 60 nodes and the second layer with 30 nodes. We can
# adjust these numbers.
# We choose relu as activation function for our first two layers, softmax for the last
# one.

```

```

# We set the kernel_size parameter to 3 which means that the size of the filter matrix
for the convolution is 3x3.
# The input_shape is simply the size of our images and 1 means that the image is
greyscale
model.add(Conv2D(60, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(30, kernel_size=3, activation='relu'))
# Here we add a Flatten layer between the Convolutional layers and the Dense layer in
order to connect both of them.
model.add(Flatten())
# The Dense layer is our output layer (standard) with the softmax activation function
in order to make the output sum up to 1.
# It means that we will have "probabilities" to predict our images
model.add(Dense(10, activation='softmax'))

# Compile model
# We use the adam optimizer and the categorical cross-entropy loss function.
# We use accuracy to measure model performance
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=
['accuracy'])

# Train the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5)

# Let's predict the first 5 images in the test set
predictions = model.predict(X_test[:5])
print("\n")
print("Predicted values: ")
print(predictions)
print("\n")

# Actual results for first 5 images in test set
print("Actual values:")
print(y_test[:5])

```

Output:

```

[0. 1. 0. 0. 0. 0. 0. 0. 0.]
Epoch 1/5
1875/1875 [=====] - 75s 40ms/step - loss: 0.2565 - accuracy:
0.9518 - val_loss: 0.1033 - val_accuracy: 0.9706
Epoch 2/5
1875/1875 [=====] - 74s 40ms/step - loss: 0.0680 - accuracy:
0.9797 - val_loss: 0.0817 - val_accuracy: 0.9738
Epoch 3/5
1875/1875 [=====] - 75s 40ms/step - loss: 0.0456 - accuracy:
0.9862 - val_loss: 0.0777 - val_accuracy: 0.9789
Epoch 4/5
1875/1875 [=====] - 78s 41ms/step - loss: 0.0336 - accuracy:
0.9895 - val_loss: 0.0932 - val_accuracy: 0.9767
Epoch 5/5
1875/1875 [=====] - 73s 39ms/step - loss: 0.0275 - accuracy:
0.9915 - val_loss: 0.0945 - val_accuracy: 0.9760

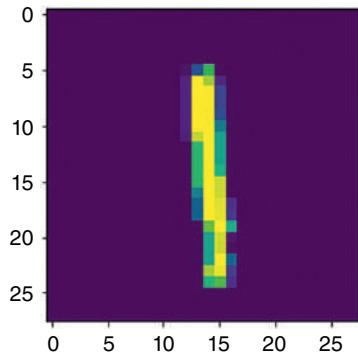
```

Predicted values:

```
[[3.0388667e-11 4.6775282e-16 2.5344246e-10 6.1128755e-09 6.3794181e-15
 4.2486505e-12 9.2009594e-21 1.0000000e+00 1.9324492e-12 6.5055183e-10]
[1.6037160e-09 7.9608098e-09 9.9999225e-01 1.2304556e-10 3.5661774e-14
 4.2955827e-13 7.7231443e-06 1.9912785e-13 3.4886691e-09 1.8349003e-16]
[8.1152383e-09 9.9996758e-01 4.8646775e-08 1.1176332e-12 2.9895480e-06
 3.7775965e-08 9.3667410e-08 2.0743075e-08 2.9326071e-05 1.7420209e-08]
[1.0000000e+00 5.1645927e-12 2.5680147e-10 7.0902344e-13 5.4258855e-15
 2.6562469e-10 3.5401580e-08 1.8963197e-12 3.2704232e-11 1.0075266e-10]
[9.7294602e-14 1.1301348e-12 6.7401423e-15 1.9974581e-15 1.0000000e+00
 2.0294430e-12 2.5000656e-15 1.6443602e-10 5.6247194e-11 3.1983134e-09]]
```

Actual values:

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```



As we can see, when we used the model to predict the first five images from the test dataset, the model predicted the following numbers (taking the highest probability): 7, 2, 1, 0, and 4. The actual values were also 7, 2, 1, 0, and 4.

3.5 Many More Algorithms to Explore

Now that you have explored some machine learning algorithms and how to code them, you are ready to use many more! In the realm of supervised learning algorithms, we can consider the following (we have seen some of them):

- Naïve Bayes with Gaussian naïve Bayes algorithm (GNB) and multinomial naïve Bayes (MNB).
- K-nearest neighbors.
- Radius nearest neighbors.
- Nearest centroid classifier.
- Linear discriminant analysis.
- SVMs.
- Stochastic gradient descent.
- Decision tree.
- Random forests classifier.
- Extremely randomized trees.
- Neural networks such as MLP.

- Classification and regression tree (C&R tree).
- Chi-squared automatic interaction detector.
- Exhaustive CHAID.
- C5.0.

For example, in scikit-learn, most of them are available and well documented:

```
from sklearn.tree import ExtraTreeClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm.classes import OneClassSVM
from sklearn.neural_network.multilayer_perceptron import MLPClassifier
from sklearn.neighbors.classification import RadiusNeighborsClassifier
from sklearn.neighbors.classification import KNeighborsClassifier
from sklearn.multioutput import ClassifierChain
from sklearn.multioutput import MultiOutputClassifier
from sklearn.multiclass import OutputCodeClassifier
from sklearn.multiclass import OneVsOneClassifier
from sklearn.multiclass import OneVsRestClassifier
from sklearn.linear_model.stochastic_gradient import SGDClassifier
from sklearn.linear_model.ridge import RidgeClassifierCV
from sklearn.linear_model.ridge import RidgeClassifier
from sklearn.linear_model.passive_aggressive import PassiveAggressiveClassifier
from sklearn.gaussian_process.gpc import GaussianProcessClassifier
from sklearn.ensemble.voting_classifier import VotingClassifier
from sklearn.ensemble.weight_boosting import AdaBoostClassifier
from sklearn.ensemble.gradient_boosting import GradientBoostingClassifier
from sklearn.ensemble.bagging import BaggingClassifier
from sklearn.ensemble.forest import ExtraTreesClassifier
from sklearn.ensemble.forest import RandomForestClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.calibration import CalibratedClassifierCV
from sklearn.naive_bayes import GaussianNB
from sklearn.semi_supervised import LabelPropagation
from sklearn.semi_supervised import LabelSpreading
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import NearestCentroid
from sklearn.svm import NuSVC
from sklearn.linear_model import Perceptron
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.svm import SVC
from sklearn.mixture import DPGMM
from sklearn.mixture import GMM
from sklearn.mixture import GaussianMixture
from sklearn.mixture import VBGMM
```

Throughout our journey, we have utilized various unsupervised learning algorithms like PCA, ICA, Isomap, LLE, and t-SNE, primarily for dimensionality reduction. Other clustering techniques such as k-means, affinity propagation, mean shift, and DBSCAN can also be examined. We will discuss some of these algorithms in Section 3.6.

Depending on the specific application, readers might be interested in exploring certain algorithms in more depth, either through user training or by utilizing pre-trained models:

- UNet-3D or VNet for image segmentation tasks.
- Wide & Deep, DLRM, HugeCTR, NCF, DIEN for recommendation systems.
- Variations of ResNet, ResNext, and EfficientNet for image classification.
- SSD, Mask R-CNN, and Faster R-CNN for object detection.
- Generative Adversarial Networks (GANs) and their derivatives for generating new data such as images and videos.
- BERT (Bidirectional Encoder Representations from Transformers), Transformer, or Generative Pre-training (GPT) models for natural language processing (NLP) tasks, including language comprehension and unsupervised language representation.

As evident, there is an abundance of topics to explore. Machine learning continues to evolve as a highly active and innovative domain.

3.6 Unsupervised Machine Learning Algorithms

Unsupervised learning is a type of machine learning approach that is utilized when the objective is to discover and extract meaningful patterns or structure from a given dataset without prior knowledge of the target variable or any labeling. This technique is particularly helpful in scenarios where the available data is large, complex, or diverse, and there is a need to gain insights and understanding of the data. Unsupervised learning algorithms can identify similarities, groupings, and outliers within the data, which can be used to form clusters, reduce dimensionality, or create visualizations that aid in data exploration.

One common application of unsupervised learning is in the field of data mining, where the objective is to uncover previously unknown relationships or associations between variables. Another use case is in anomaly detection, where the algorithm can identify unusual patterns or outliers that may indicate errors, fraud, or unusual behavior. Unsupervised learning algorithms are also used in NLP to identify topics and themes in large textual datasets.

By reducing the size of data, unsupervised learning algorithms can significantly reduce computational requirements and enhance the efficiency of downstream analysis. This can be particularly beneficial in applications such as image and speech recognition, where the amount of data can be vast, and the processing power required to analyze the data can be prohibitive. Most of the techniques for data reduction were developed in the feature extraction section (Chapter 2, Section 2.5). We explored techniques such as principal component analysis, independent component analysis, locally linear embedding, t-distributed stochastic neighbor embedding, and manifold learning techniques.

In summary, unsupervised learning is a versatile technique that can aid in data exploration, pattern discovery, clustering, and dimensionality reduction, and can be used in various domains such as data mining, anomaly detection, and NLP. Its ability to reduce the size of data can enhance the efficiency of downstream analysis and enable the processing of large-scale datasets.

3.6.1 Clustering

Clustering aims to group data points into subsets, or clusters, based on similarities in their features or attributes. There are two main types of clustering: partitional clustering and hierarchical clustering. Partitional clustering involves dividing the data into a fixed number of clusters, while hierarchical clustering builds a hierarchy of nested clusters by iteratively grouping similar clusters into larger ones. Both methods aim to maximize the similarity within clusters while minimizing the similarity between clusters. The resulting clusters can reveal insights into the underlying structure of the data and can be used for tasks such as anomaly detection, pattern recognition, and data compression.

Clustering algorithms can be evaluated based on their ability to produce meaningful and useful clusters, as well as their computational efficiency and scalability to large datasets. Popular clustering algorithms include k-means, hierarchical clustering, and density-based clustering. The choice of algorithm and clustering approach will depend on the specific characteristics of the data and the objectives of the analysis.

One example of a clustering algorithm that can be used for unsupervised learning is the k-means algorithm. In k-means clustering, the objective is to partition a given dataset into K non-overlapping clusters, where K is a predetermined value. Each data point is assigned to only one of the K clusters based on its similarity to the centroid, or center point, of the cluster. The centroids are iteratively updated until the cluster assignments stabilize, resulting in a final set of clusters. Unlike hierarchical clustering, k-means clustering does not create a hierarchical structure of nested clusters, and each data point is assigned to only one cluster. This makes k-means more suitable for datasets with a large number of data points and where non-overlapping clusters are desired. However, the performance of k-means clustering can be sensitive to the initial placement of the centroids, and it may not work well for datasets with irregular shapes or non-convex clusters. In summary, the k-means algorithm is a clustering algorithm used for unsupervised learning, which partitions a given dataset into K non-overlapping clusters. Each data point is assigned to only one cluster based on its similarity to the centroid of the cluster, and the centroids are updated iteratively until convergence. K-means is suitable for datasets with a large number of data points and non-overlapping clusters but may not work well for irregular or non-convex datasets.

Hierarchical clustering is a technique that can be performed in two different ways, namely, top-down and bottom-up clustering. Agglomerative algorithms are examples of bottom-up clustering algorithms. These algorithms start by considering each data point as an individual cluster and then combine smaller clusters progressively into larger ones. This results in a hierarchical structure of nested clusters where each cluster consists of subclusters with different levels of granularity. In contrast, divisive algorithms employ a top-down approach where the entire dataset is considered as one cluster initially. These algorithms then recursively partition the dataset into smaller and more homogeneous clusters until each data point is assigned to a separate cluster. Divisive clustering results in a binary tree structure where each node represents a partition of the data and each leaf node represents a single data point. Both agglomerative and divisive clustering have their advantages and disadvantages, and the choice of algorithm depends on the specific characteristics of the data and the objectives of the analysis. Agglomerative clustering is more efficient for large datasets with a high number of data points, whereas divisive clustering is better suited for datasets with a small number of data points or when the number of clusters is known *a priori*. In summary, hierarchical clustering can be performed using two different approaches, top-down and bottom-up clustering. Agglomerative algorithms are examples of bottom-up clustering, whereas divisive algorithms use a top-down approach. The choice of clustering algorithm depends on the specific requirements of the data analysis task. In contrast to k-means clustering, hierarchical clustering does not require a predetermined number of clusters as the number of clusters is not known beforehand and is determined based on the similarity or dissimilarity between data points.

There are several unsupervised machine learning algorithms available for clustering and dimensionality reduction, including k-means, mini-batch k-means, Ward, and mean shift. These algorithms are typically implemented in a standardized way, involving data rescaling, instantiation of the estimator, model fitting, cluster assignment (if required), and algorithm assessment. K-means clustering is a popular algorithm for unsupervised learning that has been extensively studied and used in various applications. Mini-batch k-means is a variant of k-means that is faster and more scalable, making it suitable for large datasets. Ward is a hierarchical clustering algorithm that can be used with or without connectivity constraints. Mean shift is another clustering algorithm that iteratively moves a kernel to the local mode of the distribution, resulting in clusters that are of varying sizes and shapes. Affinity propagation is another unsupervised learning algorithm that creates clusters by sending messages between pairs of samples until convergence. The algorithm can be computed based on either the Spearman distance or the Euclidean distance, with the similarity measure computed as the opposite of the distance or equality. The preference value for all points can be computed as the median, minimum, or mean value of the similarity values. Affinity propagation is implemented by rescaling the data, computing the similarity and preference values, performing affinity propagation clustering, and assessing the algorithm's performance. Another example is a density-based spatial clustering algorithm (DBSCAN) designed for applications with noise. It is an unsupervised clustering method that identifies core samples of high density and expands clusters from them. The algorithm partitions data into clusters by grouping together neighboring data points that satisfy a density criterion, while data points that do not belong to any cluster are considered noise. Unlike other clustering algorithms, DBSCAN can identify clusters of arbitrary shapes that do not need to be convex-shaped. DBSCAN is a deterministic algorithm that produces the same clusters when given the same data in the same order. However, changes to the order of the data may result in different cluster formations.

3.6.1.1 K-means

K-means clustering aims to partition a set of observations ($\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$), where each observation is a d -dimensional real vector, into k ($\leq n$) clusters $S = \{S_1, S_2, \dots, S_k\}$ to minimize the within-cluster sum of squares (WCSS) or variance. The objective is to find the values of S_i that minimize the equation:

$$\arg \min_S \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2 = \arg \min_S \sum_{i=1}^k |S_i| \text{Var } S_i$$

where μ_i ($\mu_i = \frac{1}{|S_i|} \sum_{x \in S_i} x$) is the centroid or mean of the data points in S_i , $|S_i|$ is the size of S_i , and $\|\cdot\|$ is the L2 norm. This is equivalent to minimizing the pairwise squared deviations of points within the same cluster:

$$\arg \min_S \sum_{i=1}^k \frac{1}{|S_i|} \sum_{x, y \in S_i} \|x - y\|^2$$

The total variance is constant, so maximizing the between-cluster sum of squares (BCSS) is equivalent to minimizing the WCSS. The deterministic relationship between WCSS and BCSS is also related to the law of total variance in probability theory.

To perform k-means clustering, you need to follow these steps:

- 1) First, you need to determine the number of clusters (k) you want to create and prepare a training set of examples.
- 2) Next, you need to randomly select k cluster centroids.
- 3) Assign each example in the training set to the closest centroid based on a specific distance metric for each fixed set of centroids.
- 4) Update the centroids based on the mean of the assigned data points.
- 5) Keep repeating steps 3 and 4 until convergence is reached, which is typically measured by a threshold for minimum change in either cluster assignment or centroid location.

To demonstrate k-means clustering in Python, we can use the scikit-learn library. We first need to import the necessary modules. As you can see, we included the `make_blobs()` function from `sklearn.datasets` and the K-means algorithm from `sklearn.cluster`. The `make_blobs` function is used to create a synthetic dataset, and the K-means function can be used to perform clustering on this dataset. The `make_blobs` function is used to generate a dataset of 2000 samples, with 5 centers, and a standard deviation of 1.5 for each cluster. The `random_state` parameter is set to 42 to ensure that the same dataset is generated each time the code is run. The next line sets the title of the scatter plot to "Data points." The final line creates the scatter plot of the dataset using the `scatter` function from Matplotlib. The first argument of the `scatter` function (`data[0][:,0]`) is the x -coordinates of the data points, and the second argument (`data[0][:,1]`) is the y -coordinates. The `edgecolors` parameter is set to "black" to specify the color of the edge of each point, and the `linewidths` parameter is set to 0.5 to specify the thickness of the edge. The `show` function is called to display the plot.

Input:

```
# Import the necessary modules
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt

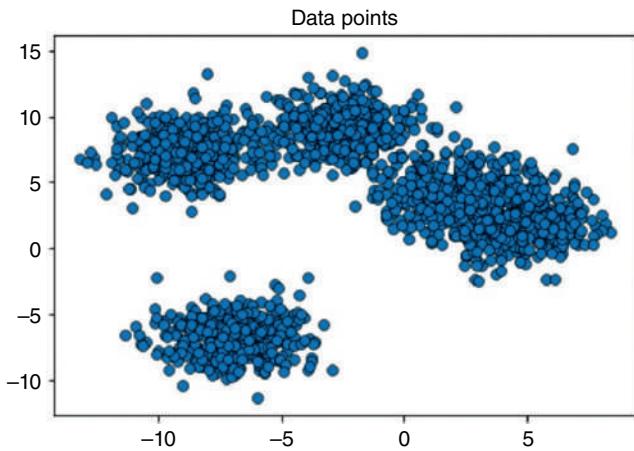
# Create the dataset
data = make_blobs(n_samples=2000, centers=5, cluster_std=1.5, random_state=42)
```

```
# Setting scatter plot title.
plt.title('Data points')

# Show the scatter plot.
plt.scatter(data[0] [:,0], data[0] [:,1], edgecolors='black', linewidths=.5);

plt.show()
```

Output:



The next Python code uses the scikit-learn library to perform k-means clustering on a synthetic dataset generated by the `make_blobs` function and generates a scatter plot of the data points color-coded by cluster. The first line creates a `K_Means` object with five clusters. The `n_clusters` parameter specifies the number of clusters to form. The second line fits the k-means algorithm to the data using the `fit` method of the `K_Means` object. This step assigns each data point to a cluster based on their proximity to the centroid of the cluster. The third line predicts the cluster labels for each data point using the `predict` method of the `K_Means` object. The fourth line sets the title of the scatter plot to “Data points in clusters.” The final line creates the scatter plot of the dataset using the `scatter` function from Matplotlib.

Input:

```
# Creating k-means algorithm and setting the number of clusters (K=5).
K_Means = KMeans(n_clusters=5)

# Training
K_Means.fit(data[0])

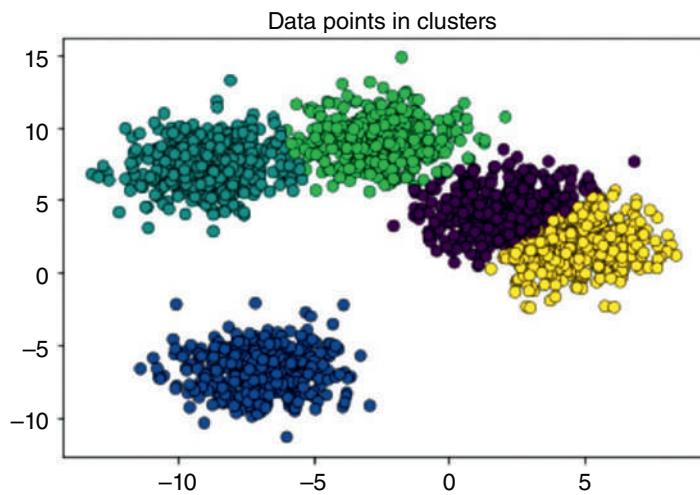
# Make predictions
clusters = K_Means.predict(data[0])

# Setting scatter plot title.
plt.title('Data points in clusters')

# Show scatter plot.
plt.scatter(data[0] [:,0], data[0] [:,1], c=clusters, edgecolors='black',
linewidths=.5)

plt.show()
```

Output:



Overall, this code demonstrates how to perform k-means clustering on a synthetic dataset using scikit-learn and visualize the clusters using a scatter plot.

3.6.1.2 Mini-batch K-means

The mini-batch k-means clustering algorithm is a variant of the popular k-means clustering algorithm, which can be utilized in place of k-means when clustering on large datasets. It is particularly useful when dealing with big datasets because it does not require iterating over the entire dataset during each step. Instead, it creates small, random subsets of the data, or batches, which are stored in memory. During each iteration, the algorithm selects a random batch of data to update the clusters. Compared to the standard k-means algorithm, the mini-batch k-means algorithm has a significant advantage in that it reduces the computational cost of finding clusters. Although k-means may be preferred for smaller datasets, the mini-batch approach should be used for larger ones.

Let us take an example with a new dataset (<https://github.com/xaviervasques/hephaistos/blob/main/data/datasets/housing.csv>).

The code below imports the MiniBatchKMeans class and seaborn, which is a library for data visualization. Next, the code reads in a CSV file called “housing.csv” from a directory one level up from the current directory (.. indicates the parent directory). The data are loaded into a pandas DataFrame called data. Then, the code selects three columns from the DataFrame using the loc method: “median_income,” “latitude,” and “longitude.” These three columns are then stored back into the data DataFrame using assignment. Finally, the head() method is called on data, which returns the first few rows of the DataFrame.

Input:

```
import pandas as pd
from sklearn.cluster import MiniBatchKMeans
import numpy as np
import seaborn as sns

data = pd.read_csv("../data/datasets/housing.csv")
data = data.loc[:, ["median_income", "latitude", "longitude"]]
data.head()
```

Output:

	Median_income	Latitude	Longitude
0	8.3252	37.88	-122.23
1	8.3014	37.86	-122.22
2	7.2574	37.85	-122.24
3	5.6431	37.85	-122.25
4	3.8462	37.85	-122.25

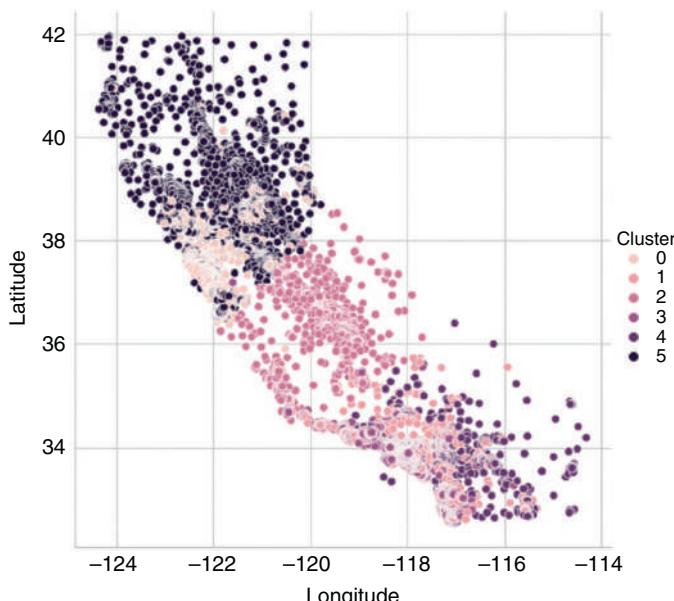
Then, we define a MiniBatchKMeans object called MiniBatch with several parameters. The `n_clusters` parameter specifies the number of clusters to be created ($K = 6$). The `batch_size` parameter sets the size of each random batch of data to be used during each iteration of the algorithm (`batch_size = 6`). Next, the code creates a new column called “Cluster” in the data DataFrame by using the `fit_predict` method of the `K_Means` object. This method fits the model to the data and assigns each data point to a cluster. The resulting cluster labels are stored in the “Cluster” column. Then, the code converts the “Cluster” column to an integer data type using the `astype` method. The next few lines of code set some plotting parameters using the `plt.style.use`, `plt.rc`, and `sns.relplot` methods. The resulting plot provides a visual representation of the clusters and can be used to gain insights into the data.

Input:

```
MiniBatch = MiniBatchKMeans(n_clusters=6, random_state=42, batch_size=6)
data["Cluster"] = MiniBatch.fit_predict(data)
data["Cluster"] = data["Cluster"].astype("int")

plt.style.use('seaborn-colorblind')
plt.rc("figure", autolayout=True)
plt.rc("axes", labelsize='large', titlesize=10, titlepad=10)
sns.relplot(x='longitude', y='latitude', hue='Cluster', data=data, height=6)
plt.show()
```

Output:



3.6.1.3 Mean Shift

Mean-shift clustering is a non-parametric and density-based clustering technique. Mean shift identifies clusters in datasets, particularly when the clusters exhibit nonlinear boundaries and irregular shapes. This method hinges on shifting each data point toward the mode or peak density of point distributions within a specified radius, iteratively continuing the process until the points converge to a local maximum of the density function, thereby representing data clusters. The mean-shift clustering algorithm can be outlined in the following steps:

- 1) Designate data points as initial cluster centroids.
- 2) Iterate the subsequent steps until convergence or a predetermined maximum iteration count is achieved:
 - For every data point, compute the mean of all points encompassed within a specified radius, or “kernel,” centered at the data point.
 - Relocate the data point to the calculated mean.
- 3) Determine cluster centroids as the points that remain stationary after convergence.
- 4) Produce the final cluster centroids and corresponding data point-to-cluster assignments.

A key advantage of mean-shift clustering is the elimination of a priori specification of cluster numbers. For instance, the k-means algorithm entails specifying the number of clusters (k) and subsequently identifying the most suitable cluster for each data instance. However, determining an appropriate initial value for k can be challenging, as k may range from 1 to the total number of data instances. The pursuit of identifying the optimal number of clusters remains an active area of research, with various techniques available, though their success varies with increasing data dimensionality. Additionally, mean shift refrains from making assumptions regarding data distribution and accommodates clusters of varying shapes and sizes. However, the algorithm’s performance is susceptible to kernel choice and kernel radius. In contrast to the widely employed k-means clustering algorithm, the mean-shift technique does not necessitate a predetermined cluster count. Instead, the algorithm inherently ascertains the optimal number of clusters based on the inherent structure of the data. The mean-shift technique is grounded in the principles of kernel density estimation (KDE). Envision the dataset as originating from a probability distribution. KDE serves as a method for approximating the underlying distribution, also known as the probability density function, associated with a dataset. This is achieved by positioning a kernel, or a weight function commonly utilized in convolution, on each data point. Numerous kernel types exist, with the Gaussian kernel being the most prevalent:

$$K(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}} e^{-\frac{1}{2}|\mathbf{x}|^2}$$

where \mathbf{x} is a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, where each observation is a d -dimensional real vector. The summation of individual kernels yields a probability surface, exemplified by a density function. The resulting density function is contingent upon the chosen kernel bandwidth parameter.

For a clearer understanding, let us examine synthetic data.

Input:

```
# Import the necessary modules
from sklearn.datasets import make_blobs
import numpy as np
import matplotlib.pyplot as plt

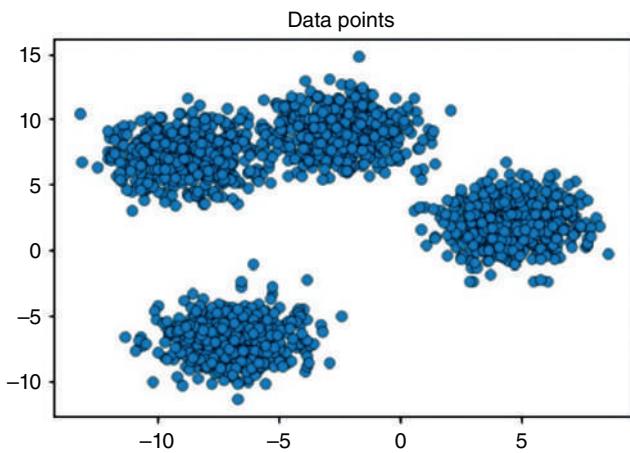
# Create the dataset
X,y = make_blobs(n_samples=2000, centers=4, cluster_std=1.5, random_state=42)

# Setting scatter plot title.
plt.title('Data points')

# Show the scatter plot.
plt.scatter(X[:,0], X[:,1], edgecolors='black', linewidths=.5)

plt.show()
```

Output:



The point density within a cluster reaches its maximum in the vicinity of the cluster's centroid. By generalizing this assertion, one can infer that the probable center of any cluster can be determined by examining the point density at specific locations in the given diagram. Consequently, the number of clusters can be ascertained, along with the approximate centers of the identified clusters. The mean shift clustering algorithm operates by identifying the “mode” of the density and assessing its highest points. It iteratively shifts data points toward the nearest mode, ultimately yielding a set of clusters and facilitating sample-to-cluster assignments upon completion of the fitting process.

Consider a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, where each observation is a d -dimensional real vector. Additionally, assume the selection of a kernel K with a bandwidth parameter h . The bandwidth parameter plays a crucial role as it delineates a region surrounding the samples, within which the mean shift algorithm should investigate to ascertain the most plausible trajectory based on density estimation. However, the determination of an appropriate bandwidth value remains a pertinent question. Employing this set of observations and kernel function, the subsequent kernel density estimator for the entire population's density function can be obtained as follows:

$$f_K(\mathbf{x}) = \frac{1}{nh^d} = \sum_{i=1}^n K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)$$

where the kernel function needs to satisfy the following conditions:

$$\int K(\mathbf{x}) d\mathbf{x} = 1 \text{ and } K(\mathbf{x}) = K(|\mathbf{x}|) \text{ for all values of } \mathbf{x}$$

Input:

```
from sklearn.cluster import MeanShift, estimate_bandwidth

# Estimate the bandwidth for Mean Shift algorithm
bandwidth = estimate_bandwidth(X, quantile=0.12, n_samples=25)

# Instantiate and fit the Mean Shift model using the estimated bandwidth
meanshift = MeanShift(bandwidth=bandwidth)
meanshift.fit(X)
```

```

# Retrieve the labels for each data point and find the unique labels
labels = meanshift.labels_
labels_unique = np.unique(labels)

# Calculate the number of clusters based on unique labels
n_clusters_ = len(labels_unique)

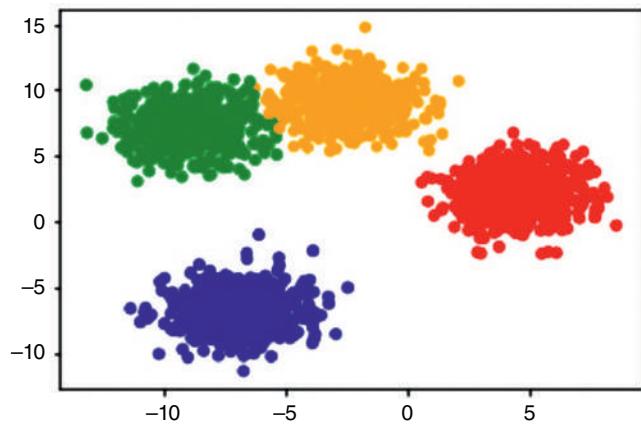
# Predict the cluster assignment for all data points
Pred = meanshift.predict(X)

# Define colors for each cluster label and generate a list of colors for the plot
colors = list(map(lambda x: 'red' if x == 1 else 'blue' if x == 2 else 'green' if x == 3
else 'orange', Pred))

# Create a scatter plot of the data points with colors based on their cluster
assignments
plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True)
plt.show()

```

Output:



As we can see in the code above, the estimate_bandwidth function plays a crucial role, as it calculates the optimal bandwidth tailored to the specific dataset under consideration. Subsequently, the estimated bandwidth is utilized during the instantiation of the mean shift algorithm. Following this, the data are fit to the model, and pertinent information such as the number of labels is derived.

3.6.1.4 Affinity Propagation

Mean-shift and affinity propagation both offer the notable benefit of obviating the need for a priori determination of cluster quantities, as they concurrently cluster data and ascertain the number of clusters. Furthermore, affinity propagation does not necessitate an estimation of cluster count before the algorithm's execution. Introduced by Frey and Dueck (2007), the affinity propagation algorithm enables the exchange of messages between data point pairs until a collection of exemplars emerges, with each exemplar symbolizing a cluster. A dataset can be delineated by a restricted set of exemplars, wherein “exemplars” refer to input set constituents that function as cluster proxies. The messages conveyed between pairs indicate the suitability of one sample to act as the exemplar for its counterpart, which is adjusted based on the values obtained from other pairs. This iterative refinement continues until convergence is achieved, at which juncture the final exemplars are chosen, resulting in the ultimate clustering. Two pivotal parameters include the preference, which dictates the number of exemplars employed, and the damping factor, which moderates the responsibility and availability of messages to prevent numerical fluctuations during message updates.

Consider a dataset D , which is comprised of data points d_1, d_2, \dots, d_n . Let s represent an $N \times N$ matrix, wherein $s(i, j)$ signifies the similarity between data points d_i and d_j . The negative squared distance between two data points is utilized as s ; for instance, for points x_i and x_j , $s(i, j)$ is equal to $-\|x_i - x_j\|^2$.

The diagonal of the matrix s , specifically $s(i, i)$, holds particular importance as it denotes the input preference, which reflects the probability of a given input serving as an exemplar. When initialized to a uniform value for all inputs, it dictates the number of classes generated by the algorithm. A value approximating the minimum possible similarity results in fewer classes, while a value that is near or surpasses the maximum possible similarity leads to an increased number of classes. Typically, the median similarity of all input pairs is employed as the initial value.

The algorithm progresses through alternating between two message-passing stages, leading to the modification of two matrices:

- The “responsibility” matrix R encompasses values $r(i, k)$, which quantify the degree to which x_k is apt to act as the exemplar for x_i in comparison to alternative candidate exemplars for x_i . The responsibility matrix is initialized to contain only zeros and updates are disseminated throughout the system:

$$r(i, k) \leftarrow s(i, k) - \max_{k' \neq k} \{a(i, k') + s(i, k')\}$$

- The “availability” matrix A comprises values $a(i, k)$, which convey the extent of “appropriateness” in x_i selecting x_k as its exemplar, considering the preferences of other data points for x_k as an exemplar. The availability matrix is also initialized to contain only zeros and updated as follows:

$$a(i, k) \leftarrow \min \left(0, r \left(k, k + \sum_{i' \notin \{i, k\}} \max(0, r(i', k)) \right) \right) \text{ for } i \neq k \text{ and } a(k, k) \leftarrow \sum_{i' \neq k} \max(0, r(i', k))$$

The iterative process continues until either the cluster boundaries exhibit consistency across multiple iterations or a pre-determined iteration count is achieved. Exemplars are derived from the ultimate matrices, identified by a positive combined value of “responsibility and availability” for themselves (specifically, $(r(i, i) + a(i, i)) > 0$).

Let us show an example in Python following the same procedure as above. After generating synthetic data, the code set up the affinity propagation algorithm by creating an instance with a preference value of -50 and fit the model to the input data X .

Input:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import AffinityPropagation
import matplotlib.pyplot as plt
from itertools import cycle

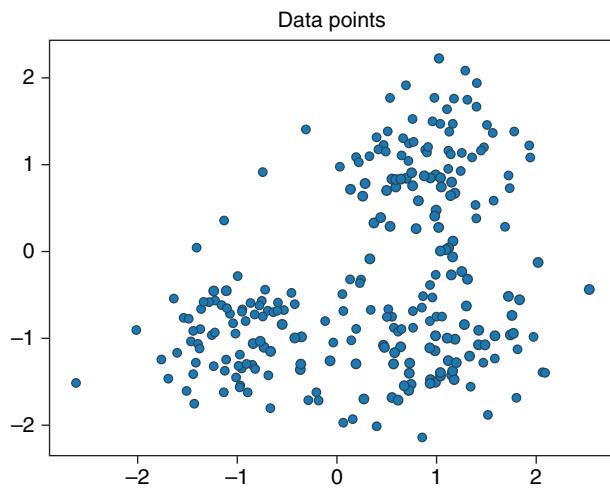
# Create the dataset
centers = [[1, 1], [-1, -1], [1, -1]]
X, y = make_blobs(n_samples=250, centers=centers, cluster_std=0.5, random_state=42)

# Setting scatter plot title.
plt.title('Data points')

# Show the scatter plot.
plt.scatter(X[:, 0], X[:, 1], edgecolors='black', linewidths=.5)

plt.show()
```

Output:



Input:

```
# Set up the Affinity Propagation algorithm by creating an instance with a preference
# value of -50 and fit the model to the input data X.
af = AffinityPropagation(preference=-50).fit(X)
# Obtain the indices of the cluster centers and the labels assigned to each data point
# by the model.
cluster_centers_indices = af.cluster_centers_indices_
# Calculate the number of clusters based on the length of the cluster centers'
# indices.
labels = af.labels_
# Print the estimated number of clusters.
no_clusters = len(cluster_centers_indices)
print('Estimated number of clusters: %d' % no_clusters)

# Plot exemplars
# Close any existing plots, create a new figure, and clear the figure
plt.close('all')
plt.figure(1)
plt.clf()
# Define a cyclic color sequence to be used in plotting the clusters.
colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')

# Iterate through the range of the number of clusters and the color sequence
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    plt.plot(X[class_members, 0], X[class_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
    markeredgecolor='k', markersize=14)
```

```

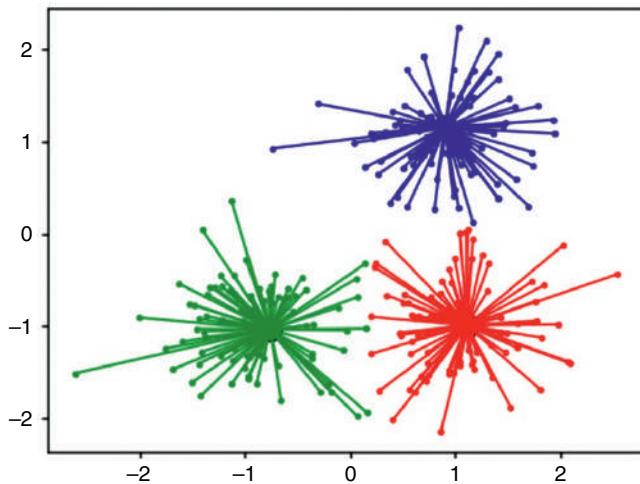
for x in X[class_members]:
    plt.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]], col)

# Display the plot showing the clusters, their centers, and the connections between
# the data points and their respective cluster centers.
plt.show()

```

Output:

Estimated number of clusters: 3



3.6.1.5 Density-based Spatial Clustering of Applications with Noise

Density-based spatial clustering of applications with noise (DBSCAN), represents an unsupervised learning algorithm extensively utilized for clustering tasks. Recognized for its ability to detect clusters of diverse shapes, this versatile clustering technique can effectively handle unique situations that other methods might not address. By employing the fundamental principle that a cluster consists of a group of densely populated data points separated from other similar groups by regions of low data point density, DBSCAN identifies distinct clusters within data. The primary goal is to discover areas characterized by high data point density and designate them as individual clusters. DBSCAN excels in uncovering clusters of various shapes and sizes within large datasets, including those with noise and outliers.

The DBSCAN algorithm is guided by two critical parameters: minPts, which is the minimum number of points (a threshold) necessary for a region to be considered dense, that is, the minimum number of data points required to form a cluster, and eps (ϵ), a distance metric used to determine the proximity of points to any given point.

Two concepts, density reachability and density connectivity, are addressed by DBSCAN. Density reachability refers to the condition where a point is considered reachable from another if it is within a specified distance (eps) from the latter, indicating the density reachability of a cluster. On the other hand, density connectivity involves DBSCAN employing a transitivity-based chaining approach to ascertain if points belong to a specific cluster. For instance, points a and d might be connected if $a \rightarrow b \rightarrow c \rightarrow d$, where $p \rightarrow q$ suggests that q is within the neighborhood of p .

In this method, data points are classified into three unique categories: core data points, which are data points with at least “minPts” neighbors within the “ ϵ ” distance; border data points, which are data points situated within the “ ϵ ” distance from a core data point but not considered core points themselves; and noise data points, which are data points that do not belong to either the core or border data point groups.

Let us apply DBSCAN to a sample dataset obtained from `sklearn.datasets`, utilizing the DBSCAN module configured with an “ ϵ ” value of 0.3 and “minPts” set to 10. In this Python code, we are comparing the performance of k-means and DBSCAN clustering algorithms on synthetic datasets.

Input:

```
# Import necessary libraries and modules
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_circles, make_moons
from sklearn.cluster import KMeans

# Generate a synthetic dataset of circles with 1000 samples, a scale factor of 0.3,
# and noise level of 0.1
X, labels_true = make_circles(n_samples=1000, factor=0.3, noise=0.1)

# Scale the features of the dataset
X = StandardScaler().fit_transform(X)

# Set the number of clusters for K-means and perform the clustering
clusters_kmeans = 2
kmeans = KMeans(n_clusters=clusters_kmeans)
y_k = kmeans.fit_predict(X)

# Perform DBSCAN clustering with an epsilon value of 0.3 and minimum samples of 10
db = DBSCAN(eps=0.3, min_samples=10).fit(X)

# Create a mask to identify core samples
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Calculate the number of clusters and noise points
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

# Set up the figure for plotting the results
fig, (ax1, ax2) = plt.subplots(ncols=2)
fig.set_figheight(10)
fig.set_figwidth(30)

# Define the colors for the clusters
colours = ['red', 'blue', 'green']

# Plot K-means clusters and centroids
for i in range(clusters_kmeans):
    ax1.scatter(X[y_k == i, 0], X[y_k == i, 1], s=100, c=colours[i],
    label='Cluster'+str(i+1))
    ax1.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
    s=100, c='black', label='Centroids')
ax1.set_title("K-means with chosen number of clusters: 2")
ax1.legend()
```

```
# Plot DBSCAN clusters
unique_labels = set(labels)
for k, col in zip(unique_labels, colours):
    if k == -1:
        col = [0, 0, 0, 1]
    class_member_mask = (labels == k)

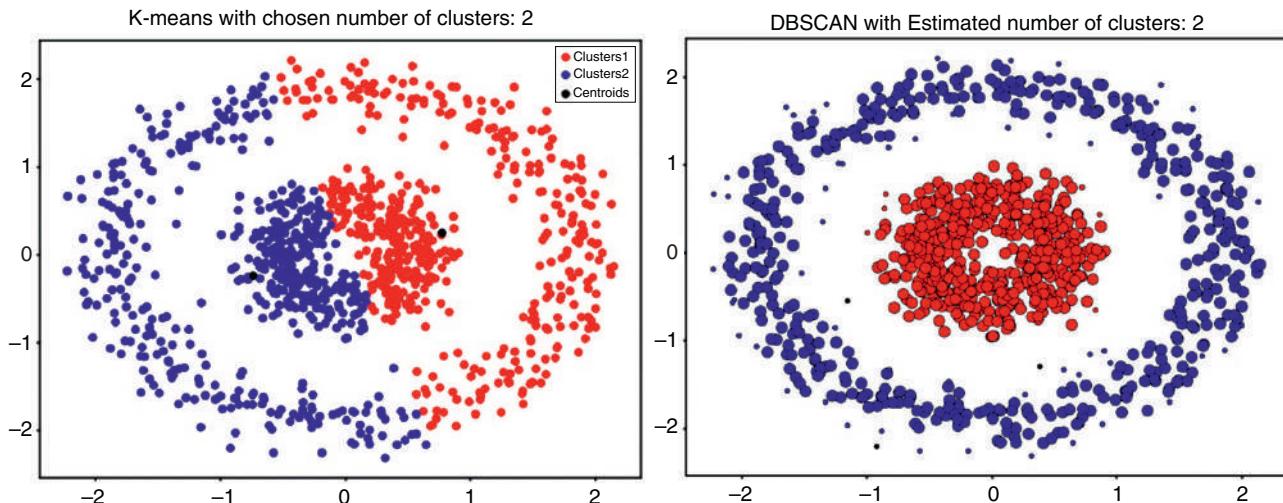
    xy = X[class_member_mask & core_samples_mask]
    ax2.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=(col),
              markeredgecolor='k', markersize=14)

    xy = X[class_member_mask & ~core_samples_mask]
    ax2.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=(col),
              markeredgecolor='k', markersize=6)

# Set the title for the DBSCAN plot
ax2.set_title('DBSCAN with Estimated number of clusters: %d' % n_clusters_)
```

Output:

Text(0.5, 1.0, 'DBSCAN with Estimated number of clusters: 2')



3.7 Machine Learning Algorithms with HephaIstos

As stated in Chapter 1 and thanks to great open-source frameworks such as scikit-learn and Keras, we can create pipelines with hephAistos for classification purposes using both CPUs and GPUs.

If we use CPUs, the *classification_algorithms* parameter can be chosen with the following options:

- *Classification_algorithms*: Classification algorithms used only with CPUs:
 - *svm_linear*
 - *svm_rbf*
 - *svm_sigmoid*
 - *svm_poly*
 - *logistic_regression*
 - *lda*
 - *qda*

- *gnb*
- *mnb*
- *kneighbors*
 - o For k-neighbors, we need to add an additional parameter to specify the number of neighbors (*n_neighbors*).
- *sgd*
- *nearest_centroid*
- *decision_tree*
- *random_forest*
 - o For random_forest, we can optionally add the number of estimators (*n_estimators_forest*).
- *extra_trees*
 - o For extra_trees, we add the number of estimators (*n_estimators_forest*).
- *mlp_neural_network*
 - o The following parameters are available: *max_iter*, *hidden_layer_sizes*, *activation*, *solver*, *alpha*, *learning_rate*, *learning_rate_init*.
 - o *max_iter*: The maximum number of iterations (default = 200).
 - o *hidden_layer_sizes*: The *i*th element represents the number of neurons in the *i*th hidden layer.
 - o *mlp_activation*: The activation function for the hidden layer (“identity,” “logistic,” “relu,” “softmax,” “tanh”; default = “relu”).
 - o *solver*: The solver for weight optimization (“lbfgs,” “sgd,” “adam”; default = “adam”).
 - o *alpha*: Strength of the l2 regularization term (default = 0.0001).
 - o *mlp_learning_rate*: Learning rate schedule for weight updates (“constant,” “invscaling,” “adaptive”; default = “constant”).
 - o *learning_rate_init*: The initial learning rate used (for sgd or Adam). It controls the step size in updating the weights.
- *mlp_neural_network_auto*: This option allows us to find the optimal parameters for the neural network
 - o For each classification algorithm, we also need to add the number of k-folds for cross-validation (*cv*).

Here is an example:

```
from ml_pipeline_function import ml_pipeline_function

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', classification_algorithms=
['svm_rbf','lda', 'random_forest', 'gpu_logistic_regression'], n_estimators_forest =
100, gpu_logistic_activation = 'adam', gpu_logistic_optimizer = 'adam',
gpu_logistic_epochs = 50, cv = 5)
```

The above code will print the steps of the processes and provide the metrics of our models such as the following:

	SVM_rbf	lda	random_forest
Rescaling Method	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal
Extraction Method	None	None	None
Accuracy	0.982456	0.938596	0.95614
Precision	0.982456	0.938596	0.95614
Recall	0.982456	0.938596	0.95614
F1 Score	0.982456	0.938596	0.95614
Cross-validation mean	0.975824	0.947253	0.958242
Cross-validation std	0.012815	0.018906	0.015541

We can also use classification algorithms that use GPUs:

- *gpu_logistic_regression*: We need to add parameters to use *gpu_logistic_regression*:
 - *gpu_logistic_optimizer*: The model optimizers such as stochastic gradient descent (SGD [learning_rate = 1e-2]), adam (“adam”), or RMSprop (“RMSprop”)
 - *gpu_logistic_loss*: The loss functions such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
 - *gpu_logistic_epochs*: The number of epochs
- *gpu_mlp*: We need to add parameters to use *gpu_mlp*:
 - *gpu_mlp_optimizer*: The model optimizers such as stochastic gradient descent (SGD [learning_rate = 1e-2]), adam (“adam”), or RMSprop (“RMSprop”)
 - *gpu_mlp_activation*: The activation functions such as softmax, sigmoid, linear, or tanh
 - *gpu_mlp_loss*: The loss functions such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
 - *gpu_mlp_epochs*: The number of epochs
- *gpu_rnn*: Recurrent neural network for classification. We need to set the following parameters:
 - *rnn_units*: A positive integer, the dimensionality of the output space
 - *rnn_activation*: The activation function to use (softmax, sigmoid, linear, or tanh)
 - *rnn_optimizer*: The optimizer (adam, sgd, RMSprop)
 - *rnn_loss*: The loss function such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
 - *rnn_epochs*: The number of epochs (integer)

Let us view an example:

```
from ml_pipeline_function import ml_pipeline_function

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', classification_algorithms=
['svm_rbf', 'lda', 'random_forest', 'gpu_logistic_regression'], n_estimators_forest =
100, gpu_logistic_activation = 'adam', gpu_logistic_optimizer = 'adam',
gpu_logistic_epochs = 50, cv = 5)
```

The above code will print the steps of the processes and provide the metrics of our models such as the following:

	SVM_rbf	lda	random_forest	gpu_logistic_regression
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal	row_removal
Extraction Method	None	None	None	None
Accuracy	0.982456	0.938596	0.964912	0.982456
Precision	0.982456	0.938596	0.964912	0.982456
Recall	0.982456	0.938596	0.964912	0.982456
F1 Score	0.982456	0.938596	0.964912	0.982456
Cross-validation mean	0.975824	0.947253	0.956044	NaN
Cross-validation std	0.012815	0.018906	0.014906	NaN

Here is another example with an SGD optimizer:

```
from ml_pipeline_function import ml_pipeline_function
from tensorflow.keras.optimizers import SGD

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', classification_algorithms=
['svm_rbf','lda', 'random_forest', 'gpu_logistic_regression'], n_estimators_forest =
100, gpu_logistic_optimizer = SGD(learning_rate = 0.001), gpu_logistic_epochs = 50,
cv = 5)
```

The above code produces the following metrics at the end:

	SVM_rbf	lda	random_forest	gpu_logistic_regression
Rescaling Method	StandardScaler	StandardScaler	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal	row_removal	row_removal
Extraction Method	None	None	None	None
Accuracy	0.982456	0.938596	0.964912	0.991228
Precision	0.982456	0.938596	0.964912	0.991228
Recall	0.982456	0.938596	0.964912	0.991228
F1 Score	0.982456	0.938596	0.964912	0.991228
Cross-validation mean	0.975824	0.947253	0.958242	NaN
Cross-validation std	0.012815	0.018906	0.020382	NaN

In addition, and with the same philosophy, regression algorithms are also available for both CPUs and GPUs:

- Regression algorithms used only with CPUs:
 - *linear_regression*
 - *svr_linear*
 - *svr_rbf*
 - *svr_sigmoid*
 - *svr_poly*
 - *mlp_regression*
 - *mlp_auto_regression*
- Regression algorithms that use GPUs, if available:
 - *gpu_linear_regression*: Linear regression using an SGD optimizer. For classification, we need to add some parameters:
 - *gpu_linear_activation*: “linear”
 - *gpu_linear_epochs*: An integer to define the number of epochs
 - *gpu_linear_learning_rate*: The learning rate for the SGD optimizer
 - *gpu_linear_loss*: The loss functions such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
 - *gpu_mlp_regression*: MLP neural network using GPUs for regression with the following parameters to set:
 - *gpu_mlp_epochs_r*: The number of epochs (integer)
 - *gpu_mlp_activation_r*: The activation function such as softmax, sigmoid, linear, or tanh
 - The chosen optimizer is “adam.” Note that no activation function is used for the output layer because it is a regression. We use *mean_squared_error* for the loss function

- *gpu_rnn_regression*: Recurrent neural network for regression. We need to set the following parameters:
 - o *rnn_units*: A positive integer, the dimensionality of the output space
 - o *rnn_activation*: The activation function to use (softmax, sigmoid, linear, or tanh)
 - o *rnn_optimizer*: The optimizer (adam, sgd, RMSprop)
 - o *rnn_loss*: The loss function such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
 - o *rnn_epochs*: The number of epochs (integer)

Let us view another example:

```
from ml_pipeline_function import ml_pipeline_function
from tensorflow.keras.optimizers import SGD

from data.datasets import breastcancer
df = breastcancer()
df = df.drop(["id"], axis = 1)

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = 'standard_scaler', regression_algorithms=
['linear_regression','svr_linear', 'svr_rbf', 'gpu_linear_regression'],
gpu_linear_epochs = 50, gpu_linear_activation = 'linear', gpu_linear_learning_rate =
0.01, gpu_linear_loss = 'mse')
```

The above code will print the steps of the processes and provide the metrics of our models such as the following:

	linear_regression	gpu_linear_regression
Rescaling Method	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal
Extraction Method	None	None
MSE	0.051602	0.068005
R-squared	0.778978	0.708721

	svr_linear	svr_rbf
Rescaling Method	StandardScaler	StandardScaler
Missing Method	row_removal	row_removal
Extraction Method	None	None
MSE	0.055885	0.016993
R-squared	0.760633	0.927214

Here is another example with an RNN:

```
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Load data
DailyDelhiClimateTrain = './data/datasets/DailyDelhiClimateTrain.csv'
df = pd.read_csv(DailyDelhiClimateTrain, delimiter=',')

# define time format
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')
```

```

# create a DataFrame with new columns (year, month and day)
df['year']=df['date'].dt.year
df['month']=df['date'].dt.month
df['day']=df['date'].dt.day

# Delete column 'date'
df.drop('date', inplace=True, axis=1)

# Rename column meantemp to Target
df = df.rename(columns={"meantemp": "Target"})

# Drop row having at least 1 missing value
df = df.dropna()

# Run ML Pipeline
ml_pipeline_function(df, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, rescaling = 'standard_scaler', regression_algorithms=
['gpu_rnn_regression'], rnn_units = 500, rnn_activation = 'tanh' , rnn_optimizer =
'RMSprop', rnn_loss = 'mse', rnn_epochs = 50)

```

To use CNNs, we can include the *conv2d* option, which will apply a two-dimensional CNN using GPUs if they are available. The parameters are the following:

- *conv_kernel_size*: The kernel_size is the size of the filter matrix for the convolution (*conv_kernel_size* × *conv_kernel_size*)
- *conv_activation*: The activation function to use (softmax, sigmoid, linear, relu, or tanh)
- *conv_optimizer*: The optimizer (adam, sgd, RMSprop)
- *conv_loss*: The loss function such as the mean squared error (“mse”), the binary logarithmic loss (“binary_crossentropy”), or the multi-class logarithmic loss (“categorical_crossentropy”)
- *conv_epochs*: The number of epochs (integer)

Let us view a short example:

```

from ml_pipeline_function import ml_pipeline_function
import pandas as pd

import tensorflow as tf
from keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

df = mnist.load_data()
(X, y), (_, _) = mnist.load_data()
(X_train, y_train), (X_test, y_test) = df

# Here we reshape the data to fit model with X_train.shape[0] images for training,
# image size is X_train.shape[1] x X_train.shape[2]
# 1 means that the image is greyscale.
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], X_train.shape[2], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], X_test.shape[2], 1)
X = X.reshape(X.shape[0], X.shape[1], X.shape[2], 1)

ml_pipeline_function(df, X, y, X_train, y_train, X_test, y_test, output_folder =
'./Outputs/', convolutional=['conv2d'], conv_activation='relu', conv_kernel_size =
3, conv_optimizer = 'adam', conv_loss='categorical_crossentropy', conv_epochs=1)

```

References

- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* 2: 303–314. <https://doi.org/10.1007/BF02551274>.
- Henseler, J., Ringle, C., and Sinkovics, R. (2009). The use of partial least squares path modeling in international marketing. *Advances in International Marketing (AIM)* 20: 277–320.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation* 9 (8): 1735–1780.
- Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Network* 4 (2): 251–257. [https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T).
- Le Cun, Y., Boser, B., Denker, J.S. et al. (1989). Handwritten digit recognition with a back-propagation network. In: *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS'89)*, pp. 396–404. MIT Press.
- Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.
- Parker, D.B. (1985). *Learning Logic*. Tech. Rep. MIT.
- Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, pp. 318–362. Foundations. MIT Press.
- Werbos, P.J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Doctoral Dissertation, Boston, MA: Applied Mathematics, Harvard University.

Further Reading

- de Amorim, R.C. (2015). Feature relevance in ward's hierarchical clustering using the L_p norm. *Journal of Classification* 32: 46–62. <https://doi.org/10.1007/s00357-015-9167-1>.
- de Amorim, C.R. and Mirkin, B. (2012). Minkowski metric, feature weighting and anomalous cluster initializing in K-Means clustering. *Pattern Recognition* 45: 1061–1075. <https://doi.org/10.1016/j.patcog.2011.08.012>.
- de Amorim, R.C. and Hennig, C. (2015). Recovering the number of clusters in data sets with noise features using feature rescaling factors. *Information Science* 324: 126–145. <https://doi.org/10.1016/j.ins.2015.06.039>.
- Ben-Hur, A., Horn, D., Siegelmann, H., and Vapnik, V.N. (2001). Support vector clustering. *Journal of Machine Learning Research* 2: 125–137.
- Berwick, R. An idiot's guide to support vector machines (SVMs). Village Idiot. <https://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf> (accessed 2022).
- Boser, B.E., Guyon, I.M., and Vapnik, V.N. (1992). A training algorithm for optimal margin classifiers. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory – COLT '92*, 144. CiteSeerX 10.1.1.21.3818. Association for Computing Machinery. <https://doi.org/10.1145/130385.130401>. S2CID 207165665.
- Besse, P. Neural Networks and Introduction to Deep Learning, wikistat, <https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-m-hdstat-rnn-deep-learning.pdf> (accessed 2022).
- Chang, C.-C. and Lin, C.-J. (2011). LIBSVM : A Library for Support Vector Machines. ACM Transactions on Intelligent Systems and Technology, 2: 1–27: 27. <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- Cheng, Y. (1995). Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17: 790–799. <https://doi.org/10.1109/34.400568>.
- Comaniciu, D. and Meer, P. (2002). Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24: 603–619. <https://doi.org/10.1109/34.1000236>.
- Cortes, C. and Vapnik, V.N. (1995). Support-vector networks. *Machine Learning* 20 (3): 273–297. CiteSeerX 10.1.1.15.9362. <https://doi.org/10.1007/BF00994018>. S2CID 206787478.
- Ding, C. and He, X. (2004). K-means clustering via principal component analysis. In: *Proceedings of the 21st International Conference on Machine Learning*, Banff, Alberta, Canada. New York, NY, USA: Association for Computing Machinery, 29. <https://doi.org/10.1145/1015330.1015408>.
- Drucker, H., Burges, C.J.C., Kaufman, L. et al. (1997). Support vector regression machines. *Advances in Neural Information Processing Systems* 9: 155–161.
- Duda, R.O. (2001). *Pattern Classification*, 2e. Wiley.
- Elman, J.L. (1990). Finding structure in time. *Cognitive Science* 14 (2): 179–211.

- Everitt, B. (2001). *Cluster Analysis*, 4e. Oxford University Press.
- Fletcher, T. (2008). Support vector machines explained. <https://api.semanticscholar.org/CorpusID:623796> (accessed 2022).
- Frey, B.J. and Dueck, D. (2007). Clustering by passing messages between data points. *Science* 315: 972–976. <https://doi.org/10.1126/science.1136800>.
- Fukunaga, K. and Hostetler, L. (1975). The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory* 21: 32–40. <https://doi.org/10.1109/TIT.1975.1055330>.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2e. Springer.
- Hinton, G.E., Srivastava, N., Krizhevsky, A. et al. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv*, abs/1207.0580, CorpusID:14832074, 1–18. <https://api.semanticscholar.org/>
- Jain, A.K. (2010). Data clustering: 50 years beyond K-means. *Pattern Recognition Letters* 31: 651–666. <https://doi.org/10.1016/j.patrec.2009.09.011>.
- Jordan, M.I. (1990). *Artificial Neural Network*. IEEE Press.
- Jurafsky, D. and Martin, J.H. (2009). Speech and Language Processing, 2e. USA: Prentice-Hall, Inc. <https://web.stanford.edu/~jurafsky/slp3/5.pdf>. Copyright © 2021. All rights reserved. Draft of December 29, 2021.
- Kanungo, T., Mount, D.M., Netanyahu, N.S. et al. (2002). An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24: 881–892. <https://doi.org/10.1109/TPAMI.2002.1017616>.
- Kingma, D. and Ba, J. (2014). Adam: a method for stochastic optimization. *Arxiv* abs/1412.6980, 1–13. <https://api.semanticscholar.org/CorpusID:6628106>.
- Krizhevsky, A., Sutskever, I., and Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25 (2): 1097–1105.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *IEEE Communications Magazine* 27 (11): 41–46.
- LeCun, Y., Jackel, L., Boser, B. et al. (1998b). Handwritten digit recognition: applications of neural networks chips and automatic learning. *Proceedings of the IEEE* 86 (11): 2278–2324.
- Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28: 129–137. <https://doi.org/10.1109/TIT.1982.1056489>.
- MacKay, D.J.C. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.
- MacQueen, J.B. (1967). Some methods for classification and analysis of multivariate observations. In: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297. University of California Press.
- McCulloch, W.S. and Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology* 52 (1/2): 99–115.
- Nesterov, Y. (1983). A method of solving a complex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady* 27: 372–376.
- Platt, J. (2000). Probabilistic outputs for support vector machines and comparison to regularized likelihood methods. In: *Advances in Large Margin Classifiers*.
- Polyak, B.T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics* 4 (5): 1–17.
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65 (6): 386.
- Santana, R., McGarry, L.M., Bielza, C. et al. (2013). Classification of neocortical interneurons using affinity propagation. *Frontiers in Neural Circuits* 7: 185. <https://doi.org/10.3389/fncir.2013.00185>.
- Sculley, D. (2010). Web-scale k-means clustering. In: *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, pp. 1177–1178. ACM.
- Smith, J.W., Everhart, J.E., Dickson, W.C. et al. (1988). Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. In: *Proceedings of the Symposium on Computer Applications and Medical Care*, pp. 261–265. IEEE Computer Society Press.
- Smola, A.J. and Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing* 14: 199–222. <https://doi.org/10.1023/b:stco.0000035301.49549.88>.
- Sutskever, I., Martens, J., Dahl, G.E., and Hinton, G.E. (2013). On the importance of initialization and momentum in deep learning. *ICML* 28 (3): 1139–1147.
- Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. (2016). Inception-ResNet and the impact of residual connections on learning. *Arxiv*, abs/1602.07261, CorpusID:1023605, 1–12. <https://api.semanticscholar.org/>.

- Vattani, A. (2011). K-means requires exponentially many iterations even in the plane. *Discrete & Computational Geometry* 45: 596–616. <https://doi.org/10.1007/s00454-011-9340-1>.
- Vlasblom, J. and Wodak, S.J. (2009). Markov clustering versus affinity propagation for the partitioning of protein interaction graphs. *BMC Bioinformatics* 10: 99. <https://doi.org/10.1186/1471-2105-10-99>.
- Ward, J.H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association* 58: 236–244. <https://doi.org/10.1080/01621459.1963.10500845>.
- https://en.wikipedia.org/wiki/K-means_clustering
- https://github.com/kavyagajjar/Clustering/blob/main/DBSCAN/Cluster_Analysis_with_DBSCAN.ipynb
- <https://machinelearningmedium.com/2017/08/11/cost-function-of-linear-regression/>
- <https://medium.com/analytics-vidhya/logistic-regression-b35d2801a29c>
- <https://medium.com/geekculture/installing-cudnn-and-cuda-toolkit-on-ubuntu-20-04-for-machine-learning-tasks-f41985fcf9b2>
- <https://python3.foobrdigital.com/machine-learning-algorithms/>
- <https://python-bloggers.com/2022/03/multiple-linear-regression-using-tensorflow/>
- <https://towardsai.net/p/machine-learning/logistic-regression-with-mathematics>
- <https://towardsdatascience.com/a-gentle-introduction-to-gradient-descent-thru-linear-regression-fb0fc86482a3>
- <https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9>
- <https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21>
- <https://www.analyticsvidhya.com/blog/2020/02/mathematics-behind-convolutional-neural-network/>
- <https://www.analyticsvidhya.com/blog/2021/08/conceptual-understanding-of-logistic-regression-for-data-science-beginners/>
- <https://www.analyticsvidhya.com/blog/2021/08/understanding-linear-regression-with-mathematical-insights/>
- <https://www.geeksforgeeks.org/affinity-propagation-in-ml-to-find-the-number-of-clusters/>
- <https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/>
- <https://www.geeksforgeeks.org/linear-regression-using-tensorflow/>
- <https://www.geeksforgeeks.org/ml-mean-shift-clustering/>
- <https://www.hindawi.com/journals/cmim/2021/8500314/>
- <https://www.kaggle.com/datasets/zalando-research/fashionmnist>
- <https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-m-hdstat-rnn-deep-learning.pdf>

4

Natural Language Processing



Photo by Annamária Borsos

Natural language processing (NLP) refers to the branch of artificial intelligence (AI) focused on giving computers the ability to understand text and spoken words. It combines computational linguistics with statistical, machine learning, and deep learning models. The main idea is to process human language in the form of text or voice data. NLP has many applications such as understanding the intent and sentiment of a speaker or a writer, translating text from one language to another, responding to spoken commands, or summarizing large volumes of text. NLP can be found in voice-operated global positioning systems (GPS), digital assistants, speech-to-text services, chatbots, named entity recognition, sentiment analysis, and text generation. Understanding human language is not an easy task, as a dialog contains many ambiguities such as sarcasms, metaphors, variations, and homonyms. When we say “understanding,” we need to be clear about the lack of consciousness of a machine. Alan Turing decided to dismiss these types of inquiries and ask a simple question: Can a computer talk like a human? To answer this question, he imagined the famous Turing Test in a paper published in 1950. A program called ELIZA subsequently succeeded in misleading people by mimicking a psychologist.

Python provides a wide range of tools and libraries for processing natural language. Popular ones are the Natural Language Toolkit (NLTK), which is an open-source collection of libraries for building NLP programs, and SpaCy, which is a free open-source library for advanced NLP. We can find routines for sentence parsing, word segmentation, tokenization, and other purposes.

A few concepts need to be understood before exploring coding of NLP. We have mentioned tokenization, which refers to a sequence of characters in a document that is used for analytical purposes. In general, we need effective tokens to allow our program to process data. Tokens should be stored in an iterable data structure such as a list or generator to facilitate the analysis or should be free of non-alphanumeric characters. For example, the following list represents tokens:

```
['My', 'name', 'is', 'Xavier', 'Vasques', 'and', 'I', 'speak', 'about', 'NLP']
```

When we have our tokens, it is easier to perform some basic analytics such as a simple word count. We can also imagine the need to create a pandas DataFrame containing some features such as the number of documents in which a token appears, a count of appearances of a token, or the rank of a token relative to other tokens. If we are in a context in which we desire to analyze emails and predict which ones are spam and which are not, we can use historical data and tag the emails that are spam in order to run classification algorithms such as KNN. We will need to convert our text into numerical data (vector format). The bag-of-words method is an approach in which each unique word in a text is represented by a number. In general, there is a need to clean our dataset by removing elements such as punctuation or common words ("a," "I," "the," etc.). NLTK includes punkt, providing a pre-trained tokenizer for English, averaged_perceptron_tagger, providing a pre-trained parts-of-speech tagger for English, and stopwords, which is a list of 179 English stop words such as "I," "a," and "the." Another concept often seen in NLP is "stemming," which consists of reducing words to the root form. If we consider the word "action," we can find in a text the following words: "action," "actionable," "actioned," "actioning," and so on. This needs to be carefully studied, as we can be in a situation of over-stemming or under-stemming. In the case of over-stemming, two different stems are integrated such as "univers" for "universe," "universal," or "university." In the case of under-stemming, two words are not stemmed even though they have the same root (e.g., "alumnus," "alumni," "alumnae"). When we need to tag each word (noun, verb, adjective, etc.), we consider parts-of-speech tagging and named entity recognition to obtain textual mentions of named entities (person, place, organization, etc.).

There are a few approaches to work on NLP, such as symbolic, statistical, and neural NLP. The symbolic approach refers to a hand-coded set of rules associated with dictionary lookup, which is different from the approaches of systems that can learn rules automatically. NLP applications face some challenges, such as handling increasing volumes of text and voice data or difficulty in scaling. The symbolic approach can be used in combination with others or when the training dataset is not large enough. Statistical NLP refers to the use of statistical inferences to learn rules automatically by analyzing large corpora. Many machine learning algorithms have been applied to handle natural language by examining a large set of features. The challenge with this approach is often the requirement for elaborate feature engineering. Deep learning approaches such as those based on convolutional neural networks and recurrent neural networks enable NLP systems to process large volumes of raw, unstructured, and unlabeled text and voice datasets.

In NLP, there are two important terms to know: natural language understanding (NLU), which is the process of making the NLP system understand a natural language input, and natural language generation (NLG), which is the process of producing sentences in the form of natural language that makes sense. NLU is composed of several steps including lexical analysis, syntactic analysis, semantic analysis, discourse analysis, and pragmatic analysis. Like NLU, NLG also contains several procedural steps such as discourse generation, sentence planning, lexical choice, sentence structuring, and morphological generation.

Let us start a classic NLP use case, described in Kaggle and using the UCI datasets, which include a collection of more than 5000 SMS phone messages. The objective of this use case is to train a machine learning model on labeled data and use it to classify unlabeled messages as spam or ham.

4.1 Classifying Messages as Spam or Ham

The dataset we will use is composed of 3375 SMS ham messages randomly chosen from 10,000 legitimate messages collected at the Department of Computer Science at the National University of Singapore and 450 SMS ham messages collected from a PhD thesis. The dataset also contains 425 SMS spam messages manually extracted from the Grumbletext Web site as well as 1002 SMS ham messages and 322 spam messages from the SMS Spam Corpus.

After installing NLTK (using pip install, for example), we import NLTK.

Input:

```
# Import libraries
import nltk
```

We can then list all the lines of text messages.

Input:

```
# Let's list of all the lines of text messages using rstrip
messages = [line.rstrip() for line in open("")]

print(len(messages))
```

Output:

5574

We check the first five messages.

Input:

```
# Print the first 5 messages
for message_no, message in enumerate(messages[:5]):
    print(message_no, message)

    print('\n')
```

Output:

0 ham Go until jurong point, crazy.. Available only in bugis n great world la e buffet...
Cine there got amore wat...

1 ham Ok lar... Joking wif u oni...

2 spam Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810075over18's

3 ham U dun say so early hor... U c already then say...

4 ham Nah I don't think he goes to usf, he lives around here though

We will use pandas, Matplotlib, and Seaborn to manipulate and visualize data.

Input:

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

# Use read_csv
messages = pd.read_csv('../data/datasets/SMSSpamCollection', sep='\t', names=['label', 'message'])

messages.head()
```

Output:

	Label	Message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina....
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro....

We can start a descriptive analysis of the data and visualize data through plots.

Input:

```
messages.describe()
messages.groupby('label').describe()
```

Output:

Label	Message			Freq
	Count	Unique	Top	
ham	4825	4516	Sorry, I'll call later	30
spam	747	653	Please call our customer service representativ...	4

Let us add a new feature called length, telling us how long the text messages are.

Input:

```
# We add a new feature called length telling us how long the text messages are
messages['length'] = messages['message'].apply(len)

messages.head()
```

Output:

	Label	Message	Length
0	ham	Go until jurong point, crazy.. Available only...	111
1	ham	Ok lar... Joking wif u oni...	29
2	spam	Free entry in 2 a wkty comp to win FA Cup fina...	155
3	ham	U dun say so early hor... U c already then say...	49
4	ham	Nah I don't think he goes to usf, he lives aro...	61

Now, we will plot the frequency versus the length of the messages.

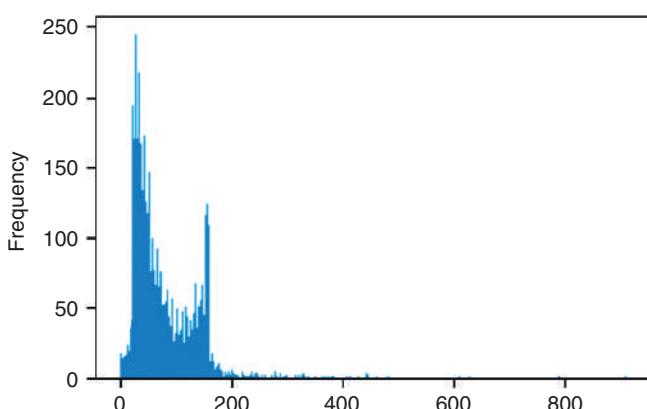
Input:

```
# Plot the frequency versus length of messages
messages['length'].plot(bins=400, kind='hist')
# Message length description

messages.length.describe()
```

Output:

```
count      5572.000000
mean       80.489950
std        59.942907
min        2.000000
25%       36.000000
50%       62.000000
75%      122.000000
max       910.000000
Name: length, dtype: float64
```



Let us be curious and read the message with 910 characters as well as the one with only two.

Input:

```
# Look at the message with 910 characters
messages[messages['length'] == 910]['message'].iloc[0]

# look at the message with 2 characters
messages[messages['length'] == 2]['message'].iloc[0]
```

Output:

"For me the love should start with attraction. i should feel that I need her every time around me. she should be the first thing which comes in my thoughts. I would start the day and end it with her. she should be there every time I dream. love will be then when my every breath has her name. my life should happen around her. my life will be named to her. I would cry for her. will give all my happiness and take all her sorrows. I will be ready to fight with anyone for her. I will be in love when I will be doing the craziest things for her. love will be when I don't have to prove anyone that my girl is the most beautiful lady on the whole planet. I will always be singing praises for her. love will be when I start up making chicken curry and end up making sambar. life will be the most beautiful then. will get every morning and thank god for the day because she is with me. I would like to say a lot..will tell later.."

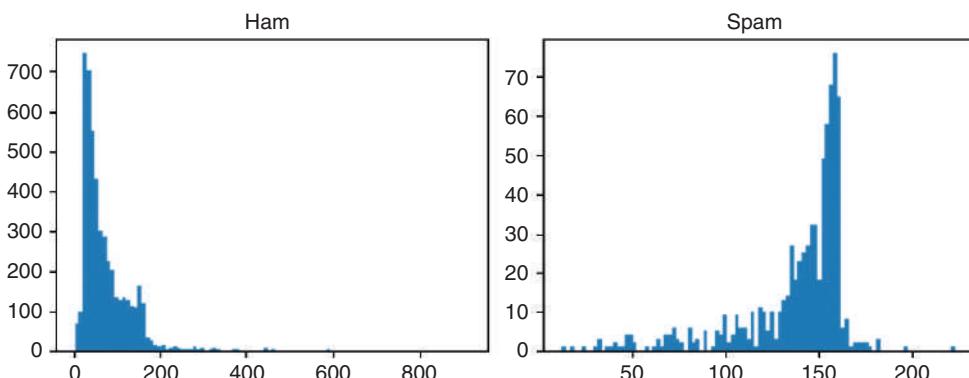
'Ok'

Let us check whether length could be a useful feature to distinguish ham and spam messages.

Input:

```
messages.hist(column='length', by='label', bins=100, figsize=(12, 4))
```

Output:



A first impression is that spam messages seem to have more characters.

As described above, we need to preprocess the data by converting the corpus to a vector format (with the bag-of-words approach, for example), split messages into individual words, remove punctuation by importing a list of English stopwords from NLTK, and remove common words. To convert the corpus to a vector format, we will use CountVectorizer from scikit-learn. We will produce a matrix of two dimensions, with the first dimension being the entire vocabulary (one row per word) and the second dimension the actual document (one column per text message). Another important concept to consider is the term frequency-inverse document frequency (TF-IDF). With the approach above using bag-of-words, we now have the word count for all messages. The challenge here is that we do not know how important those words are to the whole document. Term frequency will measure how frequently a term occurs in a document. We also need to consider the length of the document, as it will influence the frequency of a term. To normalize it, we divide by the total number of terms in the document:

$$TF = \frac{\text{Number of times term } t \text{ appears in a document}}{\text{Total number of terms in the document}}$$

Inverse document frequency measures how important a term is by weighting down the frequent terms while weighting up the rare ones:

$$IDF = \log \frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}}$$

Input:

```
import string
from nltk.corpus import stopwords

def text_process(mess):

    # Check if punctuation is present
    nopunc = [char for char in mess if char not in string.punctuation]

    # Join the characters again to form the string.
    nopunc = ".join(nopunc)

    # Remove any stopwords
    return [word for word in nopunc.split() if word.lower() not in stopwords.words
('english')]

# Look at the first five messages
messages['message'].head(5).apply(text_process)
```

Output:

```
0  [Go, jurong, point, crazy, Available, bugis, n...
1                  [Ok, lar, Joking, wif, u, oni]
2  [Free, entry, 2, wkly, comp, win, FA, Cup, fin...
3      [U, dun, say, early, hor, U, c, already, say]
4  [Nah, dont, think, goes, usf, lives, around, t...
```

Input:

```
from sklearn.feature_extraction.text import CountVectorizer

bag_transformer = CountVectorizer(analyzer=text_process).fit(messages['message'])

# Number of vocab words
print(len(bag_transformer.vocabulary_))

# Transform bag-of-words
messages_bag = bag_transformer.transform(messages['message'])

# Matrix size and amount of non-zero occurrences
print('Matrix Shape: ', messages_bag.shape)
print('Non-zero occurrences: ', messages_bag.nnz)

# TF-IDF
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer().fit(messages_bag)
messages_tfidf = tfidf_transformer.transform(messages_bag)
print(messages_tfidf.shape)
```

Output:

```
11425
Matrix Shape: (5572, 11425)
Non-zero occurrences: 50548
(5572, 11425)
```

As we can see above, we have 11,425 columns (unique words) and 5572 rows (messages). This matrix is called the bag of words.

It is now time to train a model and apply it to the test SMS dataset (20% of the entire dataset) to predict whether SMS messages are ham or spam. We will use naive Bayes with scikit-learn.

Input:

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB

# Split dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = \
train_test_split(messages_tfidf, messages['label'], test_size=0.2)

# Model fit
spam_detect_model = MultinomialNB().fit(X_train, y_train)

# Predict testint data
predictions = spam_detect_model.predict(X_test)
print(predictions)

from sklearn.metrics import classification_report
print(classification_report(y_test, predictions))
```

Output:

	precision	recall	f1-score	support
ham	0.97	1.00	0.98	977
spam	1.00	0.78	0.87	138
accuracy			0.97	1115
macro avg	0.98	0.89	0.93	1115
weighted avg	0.97	0.97	0.97	1115

Let us also test with an SVM using a linear kernel.

Input:

```
from sklearn.model_selection import train_test_split
from sklearn import svm
```

```
# Split dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = \
train_test_split(messages_tfidf, messages['label'], test_size=0.2)

# Model fit
spam_detect_model = svm.SVC(kernel='linear').fit(X_train, y_train)

# Predict testint data
predictions = spam_detect_model.predict(X_test)
print(predictions)

from sklearn.metrics import classification_report

print(classification_report(y_test, predictions))
```

Output:

	precision	recall	f1-score	support
ham	0.98	1.00	0.99	975
spam	1.00	0.84	0.91	140
accuracy			0.98	1115
macro avg	0.99	0.92	0.95	1115
weighted avg	0.98	0.98	0.98	1115

4.2 Sentiment Analysis

A very common use case in NLP is sentiment analysis. Its goal is to provide an idea regarding what people think about a topic by analyzing, for example, tweets (now called tweets), articles, or comments. In this section, we will explore the domain of sentiment analysis using a dataset from what was previously called Twitter, now X. This is possible by using Tweepy, which is a Python library for accessing X API. For accessing X data, it is necessary to have a Twitter Developer Account. The objective of our use case is to analyze what people think about AI, but of course it could be any topic. For this use case, we need to install TextBlob, Tweepy, and WordCloud.

If we go to hephaistos/Notebooks with a terminal and open a Jupyter Notebook, we will see in a browser a file named Natural Language Processing.ipynb that contains all code examples shown in this section. The notebooks can also be downloaded here:

<https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Natural%20Language%20Processing.ipynb>.

Let us first import the necessary libraries and obtain the keys and access tokens for authentication purposes.

Input:

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

import tweepy
from textblob import TextBlob
```

```

import re
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from wordcloud import WordCloud
import json
from collections import Counter

# Keys and Access Token for authentication
APIKey = "YOUR API Key"
APIKeySecret = "YOUR API KEY SECRET"
accessToken = "YOUR ACCESS TOKEN"
accessTokenSecret = "YOUR ACCESS TOKEN SECRET"
bearer_token = "YOUR BEARER TOKEN"

auth = tweepy.OAuthHandler(APIKey, APIKeySecret)
auth.set_access_token(accessToken, accessTokenSecret)
api = tweepy.API(auth, wait_on_rate_limit=True)

```

We can now define the search keyword (artificial intelligence) and the number of tweets to obtain the last 5000. This may take some time.

Input:

```

query = 'artificial intelligence'
max_tweets = 5000
searched_tweets = [status for status in tweepy.Cursor(api.search_tweets, q=query).
items(max_tweets)]
searched_tweets

```

Now, we can start some routines for sentiment analysis. The most common method is to analyze whether the tweets are generally positive, negative, or neutral.

Input:

```

# Sentiment analysis: Positive, Negative and Neutral Tweets
positive = 0
negative = 0
neutral = 0

for tweet in searched_tweets:
    analysis = TextBlob(tweet.text)
    if analysis.sentiment[0]>0:
        positive = positive +1
    elif analysis.sentiment[0]<0:
        negative = negative + 1
    else:
        neutral = neutral + 1

print("Total Positive = ", positive)
print("Total Negative = ", negative)
print("Total Neutral = ", neutral)

```

Output:

```
Total Positive = 1004
Total Negative = 3327
Total Neutral = 669
```

Next, we will create a Dataframe of tweets to facilitate operations and clean the searched tweets.

Input:

```
my_list_of_dicts = []
for each_json_tweet in searched_tweets:
    my_list_of_dicts.append(each_json_tweet._json)

with open('tweet_json_data.txt', 'w') as file:
    file.write(json.dumps(my_list_of_dicts, indent=4))

my_demo_list = []
with open('tweet_json_data.txt', encoding='utf-8') as json_file:
    all_data = json.load(json_file)
    for each_dictionary in all_data:
        tweet_id = each_dictionary['id']
        text = each_dictionary['text']
        favorite_count = each_dictionary['favorite_count']
        retweet_count = each_dictionary['retweet_count']
        created_at = each_dictionary['created_at']
        my_demo_list.append({'tweet_id': str(tweet_id),
                             'text': str(text),
                             'favorite_count': int(favorite_count),
                             'retweet_count': int(retweet_count),
                             'created_at': created_at,
                             })

tweet_dataset = pd.DataFrame(my_demo_list, columns =
                            ['tweet_id', 'text',
                             'favorite_count', 'retweet_count',
                             'created_at'])

#Writing tweet dataset to csv file for future reference
tweet_dataset.to_csv('tweet_data.csv')
```

Let us now check the shape and the first 10 tweets of our Dataframe.

Input:

```
tweet_dataset.shape
```

Output:

```
(5000, 5)
```

Input:

```
tweet_dataset.head()
```

Output:

	tweet_id	text	favorite_count	retweet_count	created_at
0	1599404573386412032	RT @RealDMitchell: My @ObsNewReview column tod...	0	1	Sun Dec 04 14:05:40 +0000 2022
1	1599404503803236352	@renoomokri Peter Obi's Manifesto is the most...	0	0	Sun Dec 04 14:05:23 +0000 2022
2	1599404474904514560	My @ObsNewReview column today is about surveys...	3	1	Sun Dec 04 14:05:16 +0000 2022
3	1599404456277577733	RT @Nilofer_tweets: Harvard University is offe...	0	12	Sun Dec 04 14:05:12 +0000 2022
4	1599404421943013377	RT @VLSorrelllImages: #JADEL\n@CASEfromASE EIC...	0	1	Sun Dec 04 14:05:04 +0000 2022

Let us now clean our dataset.

Input:

```
def remove_pattern(input_txt, pattern):
    r = re.findall(pattern, input_txt)
    for i in r:
        input_txt = re.sub(i, " ", input_txt)

    return input_txt

tweet_dataset['text'] = np.vectorize(remove_pattern)(tweet_dataset['text'],
                                             "@[\w]*")
```

Input:

```
corpus = []
for i in range(0, 1000):
    tweet = re.sub('^[a-zA-Z0-9]', ' ', tweet_dataset['text'][i])
    tweet = tweet.lower()
    tweet = re.sub('rt', " ", tweet)
    tweet = re.sub('http', " ", tweet)
    tweet = re.sub('https', " ", tweet)
    tweet = tweet.split()
    ps = PorterStemmer()
    tweet = [ps.stem(word) for word in tweet if not word in set(stopwords.words('english'))]
    tweet = ' '.join(tweet)
    corpus.append(tweet)
```

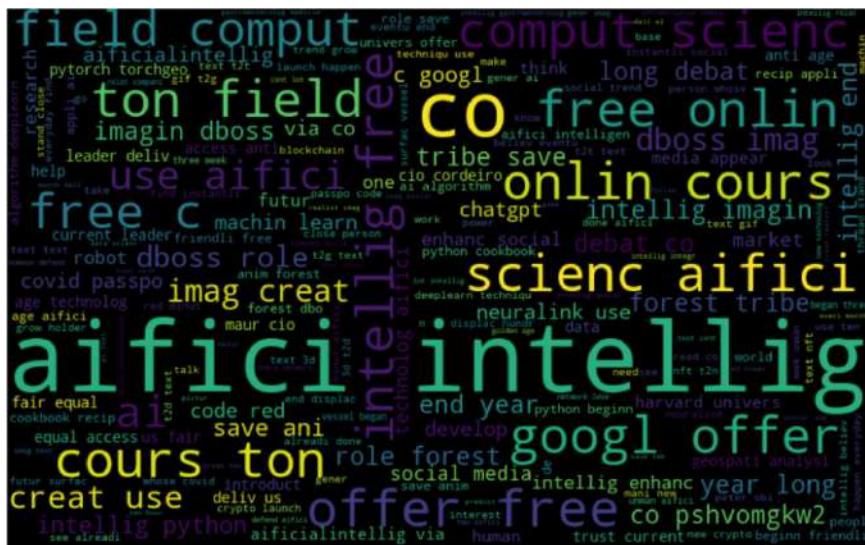
We can also visualize the data with WordCloud.

Input:

```
all_words = ' '.join([text for text in corpus])
wordcloud = WordCloud(width=800, height=500, random_state=21, max_font_size=110).generate(all_words)
```

```
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

Output:



We can also check for term frequency.

Input:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.90, min_df=2, max_features=1000,
stop_words='english')
tfidf = tfidf_vectorizer.fit_transform(tweet_dataset['text'])
#Count Most Frequent Words
Counter = Counter(corpus)
most_occur = Counter.most_common(10)
print(most_occur)
```

Output:

```
[('googl offer free onlin cours ton field comput scienc aifici intellig 10 free c', 134),  
 ('aifici intellig end 14 year long debat co pshvomgkw2', 32), ('dboss imag creat use aifici  
 intellig imagin dboss role forest tribe save ani', 28), ('dboss imag creat use aifici intellig  
 dboss role save anim forest dbo', 13), ('aifici intellig python cookbook recip appli ai  
 algorithm deeplearn techniqu use ten', 13), ('4 trust current leader deliv us faire equal access  
 1 anti age technolog 2 aifici intelligen', 13), ('harvard univers offer free cours aifici  
 intellig python beginn friendli 100 free', 12), ('thread stand close person whose covid passpo  
 code red 10 minut aifici intellig', 12), ('see alreadi done aifici intellig believ eventu  
 end displac hundr', 12), ('mani new crypto launch happen everyday find instantli social  
 trend grow holder', 10)]
```

Interestingly, the 5000 tweets we have analyzed are more negative than positive about AI. In the WordCloud image, we see words and phrases such as science, free, computer science, role, social media, machine learning, google, code, python, debate, use, research, and others.

4.3 Bidirectional Encoder Representations from Transformers

In recent years, significant advancements in NLP have been observed, commencing in 2018 with the introduction of two large-scale deep learning models: Generative Pre-Training (GPT) and Bidirectional Encoder Representations from Transformers (BERT), which encompass BERT-Base and BERT-Large. Contrary to earlier NLP models, BERT is an open-source, highly bidirectional, unsupervised language representation that is pre-trained solely on plain text corpora. This period also witnessed the emergence of other substantial deep learning language models such as GPT-2, RoBERT, ESIM+GloVe, and GPT-3, fueling extensive technological discourse.

Foundation models (FMs) are defined as expansive AI models trained on large-scale, unlabeled datasets, typically through self-supervised learning. This process results in a model capable of adapting to various downstream tasks. Alternative terms for foundation models include Large Models, Large Language Models (LMs/LLMs), State-of-the-Art (SOTA) models, or “Generative AI.” The development of such models is partly due to the challenge of limited training data in NLP. While an abundance of textual data exists globally, creating task-specific datasets requires segmenting this vast collection into numerous distinct domains. This process yields only a few thousand or hundred thousand human-annotated training examples. Unfortunately, deep learning NLP models require significantly larger data quantities for optimal performance, with improvements observed when trained on millions or billions of labeled examples.

Researchers have developed strategies for training universal language representation models using the extensive amounts of unlabeled text available online, a process referred to as pre-training. These versatile pre-trained models can then be fine-tuned using smaller, task-specific datasets for applications such as question-answering and sentiment analysis. This approach yields considerable accuracy improvements compared to training exclusively on smaller, specialized datasets. BERT, a recent development in NLP pre-training methodologies, has attracted attention in the deep learning community due to its exceptional performance across various NLP tasks, including question-answering. These characteristics contribute to the increasing importance and applicability of foundation models in AI. One approach to utilizing foundation models targets enterprises or research institutions that require precise, domain-specific models for purposes like automating workflows or accelerating scientific advancements. However, training a foundation model on the internet does not inherently make it a domain expert, even if it appears credible to non-specialists in a particular field. While the AI revolution brings considerable enthusiasm, managing foundation models is highly complex. The extensive process of converting data into a functional model ready for deployment may necessitate weeks of manual labor and substantial computational resources. Significant investments across all aspects, not just the models themselves, are crucial for fully harnessing the potential of foundation models. Foundation models are expected to accelerate AI integration within businesses substantially. By alleviating labeling requirements, organizations will find AI adoption considerably more straightforward. Ensuring the capabilities of foundation models are accessible to all businesses in a hybrid cloud environment is of paramount importance. Multiple use cases include training a foundation model on various technical documents (e.g., equipment manuals, product catalogs, how-to guides), using the model to label diverse images via a truly domain-expert AI, or generating lines of code.

ChatGPT, an AI application developed by OpenAI, has garnered significant global interest. Designed as a chatbot-style interface, ChatGPT allows users to pose open-ended questions or prompts for the model to answer, showcasing the potential of Large Language models. While ChatGPT primarily serves as a tool for tasks such as simplifying copywriting or composing vows, it represents a critical milestone in the lengthy history of AI. Foundation models exhibit five distinct characteristics:

- **They are trained on vast amounts of unlabeled data:** GPT-3, developed by OpenAI, is a well-known FM trained on 499 billion tokens of text, encompassing web crawling, Reddit, and Wikipedia data, which equates to roughly 375 billion words.

- **They are sizable:** GPT-3 comprises 175 billion parameters, where parameters are adjustable weights used to determine a model's output. In comparison, linear regression models have two parameters, while GPT-3 possesses nearly 100 billion times that amount.
- **They are self-supervised:** Self-supervised learning is a methodology in which a model is trained to recognize specific data patterns autonomously, without annotated or labeled datasets, akin to how children acquire language with minimal instruction.
- **They are general:** FMs are applicable across multiple tasks and do not require explicit training for a single task. They can scale their performance without restarting for each task. For instance, GPT-3 can be employed for various tasks such as question-answering, language translation, sentiment analysis, and more, without retraining.
- **They are generative:** FMs can generate novel ideas or content, particularly useful in knowledge tasks.

In this discussion, we will examine BERT. Its most attractive features are affordability and accessibility, as it can be downloaded and used without any cost. BERT models can be employed to extract high-quality linguistic features from text data or fine-tuned for specific tasks such as question-answering, abstract summarization, sentence prediction, conversational response generation, and using users' data to produce state-of-the-art predictions. Language modeling, at its core, focuses on predicting missing words within a given context. The primary objective of language models is to complete sentences by estimating the probability of a word filling a blank space. For example, in the sentence, "Laura and Elsa traveled to Montpellier and purchased a _____ of shoes," a language model might estimate an 80% probability for the word "pair" and a 20% probability for the word "cart."

Before BERT, language models would analyze text sequences during training in a unidirectional manner, either from left to right or by combining both left-to-right and right-to-left perspectives. This one-directional approach is effective for generating sentences, as the model can predict subsequent words and progressively construct a complete sentence.

BERT, however, introduces a bidirectional training approach, which is its key technical innovation. This bidirectionality allows BERT to have a more profound understanding of language context and flow compared to single-direction language models.

Instead of predicting the next word in a sequence, BERT employs a groundbreaking technique known as Masked Language Modeling (MLM). This technique involves randomly masking words within a sentence and then predicting them. By masking words, BERT is able to consider both the left and right contexts of a sentence when predicting a masked word. This distinguishes BERT from previous language models such as LSTM-based models, which lacked the simultaneous consideration of both preceding and following tokens. It may be more precise to describe BERT as non-directional rather than bidirectional.

BERT's bidirectional training is grounded in the Transformer architecture, which leverages self-attention mechanisms to process input sequences in parallel, rather than sequentially. This allows BERT to efficiently capture long-range dependencies and complex relationships between words, providing a richer understanding of the sentence's meaning. By employing this advanced mathematical framework, BERT has achieved state-of-the-art results on various NLP tasks, revolutionizing the field of NLP. BERT relies on the Transformer model architecture, as opposed to LSTMs. A Transformer operates by executing a fixed, minimal number of steps. In each step, it applies an attention mechanism to identify relationships among all words in a sentence, independent of their respective positions. For example, consider the sentence, "The bat flew swiftly in the dim cave." To comprehend that "bat" denotes an animal rather than a piece of baseball equipment, the Transformer can promptly focus on the words "flew" and "cave," and deduce the correct meaning in just one step.

4.4 BERT's Functionality

Upon searching "BERT" online, numerous variations can be found. BERT was originally designed for the English language in two specific model dimensions: (i) BERT-BASE, featuring 12 encoders with 12 bidirectional self-attention heads, resulting in 110 million parameters, and (ii) BERT-LARGE, comprising 24 encoders with 16 bidirectional self-attention heads, which accumulates to 340 million parameters. Both versions were pre-trained utilizing the Toronto BookCorpus (800 million words) and English Wikipedia (2500 million words).

BERT is grounded in a Transformer architecture, an attention mechanism that discerns contextual relationships among words in a text. A fundamental Transformer comprises an encoder, which processes the text input, and a decoder, which yields a prediction for a given task. BERT's objective is to establish a language representation model, so it solely requires the

encoder component. The encoder accepts a sequence of tokens as input, which are initially converted into vectors and subsequently analyzed within the neural network. In preparation for processing, the input is enriched with the addition of supplementary metadata. Token embeddings are utilized by incorporating a [CLS] token at the beginning of the first sentence and appending a [SEP] token at the end of each sentence. Segment embeddings are employed by marking each token with either Sentence A or Sentence B, allowing the encoder to distinguish between sentences. Lastly, positional embeddings are assigned to every token to indicate its position within the sentence.

In essence, the Transformer constructs layers that map sequences to sequences, resulting in an output sequence of vectors that correspond on a 1 : 1 basis with input and output tokens at identical indices. As previously noted, BERT does not aim to predict subsequent words in a sentence.

Input	[CLS]	Mg	daughter	in	smart	[SEP]	She	has	good	grades	[SEP]
Token Embeddings	$E_{[CLS]}$ +	E_{Mg} +	E_{daughter} +	E_{in} +	E_{smart} +	$E_{[\text{SEP}]}$ +	E_{She} +	E_{has} +	E_{good} +	E_{grades} +	$E_{[\text{SEP}]}$ +
Segment Embeddings	E_A +	E_A +	E_A +	E_A +	E_A +	E_A +	E_B +	E_B +	E_B +	E_B +	E_B +
Position Embeddings	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}

The training process incorporates a duo of primary methodologies. First, the Masked Language Model (MLM) strategy is employed. It is founded on the idea of obfuscating 15% of input words by substituting them with a [MASK] token. The entire sequence is then channeled through BERT's attention-based encoder, with predictions made solely for the masked words based on the context provided by the remaining non-masked words in the sequence. However, this elementary masking approach encounters a limitation: the model only attempts predictions when the [MASK] token is present in the input, whereas the objective is to predict the accurate tokens irrespective of the input token. To resolve this, the 15% of tokens chosen for masking undergo a series of adjustments. Eighty percent of these tokens are replaced with the [MASK] token, 10% are exchanged with a random token, and the remaining 10% are left unchanged. During training, the BERT loss function concentrates solely on the prediction of masked tokens, disregarding the predictions of non-masked tokens. Consequently, the model converges at a notably slower pace compared to left-to-right or right-to-left models.

Second, the Next Sentence Prediction (NSP) technique is utilized to understand the relationship between two sentences, which proves beneficial for tasks such as question answering. In the training phase, the model is presented with pairs of sentences, learning to predict whether the second sentence is a continuation of the first in the original text. BERT employs a distinctive [SEP] token to delineate sentences. While training, the model is provided with two input sentences simultaneously, such that 50% of the time, the second sentence follows the first one directly, and 50% of the time, it is a random sentence from the entire corpus. BERT is then required to predict whether the second sentence is random or not, presuming that the random sentence is disconnected from the first. In order to ascertain whether the second sentence is linked to the first, the complete input sequence is processed through the Transformer-based model. The output of the [CLS] token is converted into a 2×1 shaped vector using a rudimentary classification layer, and the IsNext-Label is assigned via softmax. The model is trained by combining both the MLM and NSP methodologies to minimize the joint loss function of the two approaches.

4.5 Installing and Training BERT for Binary Text Classification Using TensorFlow

Configuring your Python TensorFlow environment is quite straightforward:

- Clone the BERT GitHub repository onto your computer. In your terminal, enter the following command: `git clone https://github.com/google-research/bert.git`.
- Obtain the pre-trained BERT model files from the official BERT GitHub page. These files contain the weights, hyperparameters, and other essential information BERT acquired during pre-training. Save these files in the directory where you cloned the GitHub repository and then extract them. The following links provide access to different files:
 - BERT-Large, Uncased (Whole Word Masking): 24-layer, 1024-hidden, 16-heads, 340M parameters: https://storage.googleapis.com/bert_models/2019_05_30/wwm_uncased_L-24_H-1024_A-16.zip

- BERT-Large, Cased (Whole Word Masking): 24-layer, 1024-hidden, 16-heads, 340M parameters: https://storage.googleapis.com/bert_models/2019_05_30/wwm_cased_L-24_H-1024_A-16.zip
- BERT-Base, Multilingual Cased: 104 languages, 12-layer, 768-hidden, 12-heads, 110M parameters: https://storage.googleapis.com/bert_models/2018_11_23/multi_cased_L-12_H-768_A-12.zip
- BERT-Base, Multilingual (**Not recommended, use Multilingual Cased instead**): 102 languages, 12-layer, 768-hidden, 12-heads, 110M parameters: https://storage.googleapis.com/bert_models/2018_11_03/multilingual_L-12_H-768_A-12.zip

More can be found on GitHub: <https://github.com/google-research/bert.git>.

We observe that there are files designated as “cased” and “uncased,” indicating whether letter casing is deemed beneficial for the task in question. In our example, we opted to download the BERT-Base-Cased model.

To utilize BERT, it is necessary to transform our data into the format BERT anticipates. BERT requires data to be in a TSV file with a specific structure with four columns:

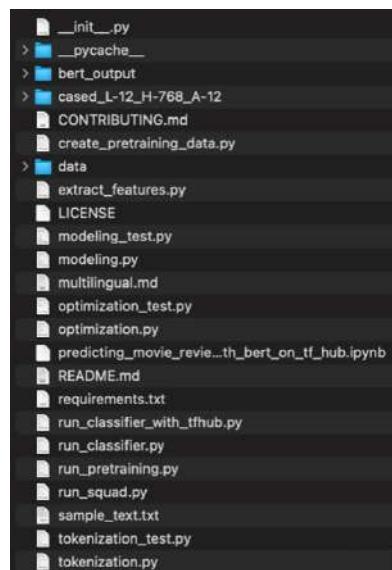
- **Column 0:** A unique identifier for each row.
- **Column 1:** An integer label for the row (class labels: 0, 1, 2, 3, etc.).
- **Column 2:** A consistent letter for all rows, included solely because BERT expects it, though it serves no purpose.
- **Column 3:** The text samples we aim to classify.

In the subsequent analysis, we will interact with the Yelp Reviews Polarity dataset. By leveraging the pandas library, we will import and meticulously analyze this information. The dataset comprises user-generated assessments and ratings for a wide variety of businesses, primarily centered on restaurants and local services, as featured on Yelp’s platform. This aggregation of data provides crucial insights into consumer preferences, experiences, and viewpoints, thus facilitating businesses and researchers in deciphering customer behavior and enhancing their offerings. The dataset can be accessed at https://www.tensorflow.org/datasets/catalog/yelp_polarity_reviews. Designed for binary sentiment classification, the Yelp Reviews Polarity dataset includes 560,000 highly polarized Yelp reviews for training and an additional 38,000 for testing. This dataset originates from Yelp reviews and constitutes a portion of the Yelp Dataset Challenge 2015 data. For additional details, please visit <http://www.yelp.com/dataset>. We can find the Jupyter Notebook at <https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/BERT.ipynb>.

As delineated above, it is necessary to create a folder within the directory where BERT was cloned, which will house three distinct files: train.tsv, dev.tsv, and test.tsv (where TSV denotes tab-separated values). Both train.tsv and dev.tsv should encompass all four columns, while test.tsv ought to contain only two columns, specifically the row ID and the text designated for classification.

Additionally, we should create a folder named “data” within the “bert” directory to store the .tsv files and another folder called “bert_output” where the fine-tuned model will be saved. The pre-trained BERT model should be stored in the “bert” directory as well.

The “bert” folder looks like this:



Let us prepare our data.

Input:

```
# Import the necessary libraries
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Use pandas read_csv function to load the Yelp Reviews Polarity dataset into a
DataFrame
# The dataset is stored in the file 'train.csv' and 'test.csv' with comma-separated
values
# Assign column names 'label' and 'text' to the respective columns in the DataFrames
df_bert_train = pd.read_csv('../data/datasets/yelp_review_polarity_csv/train.csv',
names=['label', 'text'])
df_bert_test = pd.read_csv('../data/datasets/yelp_review_polarity_csv/test.csv',
names=['label', 'text'])

# Display the first five rows of the training DataFrame to verify the data import
df_bert_train.head()
```

Output:

	label	text
0	1	Unfortunately, the frustration of being Dr. Go...
1	2	Been going to Dr. Goldberg for over 10 years. ...
2	1	I don't know what Dr. Goldberg was like before...
3	1	I'm writing this review to give you a heads up...
4	2	All the food is great here. But the best thing...

Input:

```
# Instantiate a LabelEncoder object
labelencoder = LabelEncoder()

# Use the LabelEncoder object to fit and transform the 'label' column in the DataFrame
# This converts the original labels into integer-encoded labels
df_bert_train['label'] = labelencoder.fit_transform(df_bert_train['label'])
df_bert_test['label'] = labelencoder.fit_transform(df_bert_test['label'])

# Show the first five rows of the DataFrame, displaying the transformed 'label' column
df_bert_train.head()
```

Output:

	label	text
0	0	Unfortunately, the frustration of being Dr. Go...
1	1	Been going to Dr. Goldberg for over 10 years. ...
2	0	I don't know what Dr. Goldberg was like before...
3	0	I'm writing this review to give you a heads up...
4	1	All the food is great here. But the best thing...

Input:

```
# Construct a new DataFrame for training in compliance with BERT's specifications
df_bert_train = pd.DataFrame({
    'id': range(len(df_bert_train)), # Create an 'id' column with a sequence of integers
    # from 0 to the length of df_bert_train
    'label': df_bert_train['label'], # Incorporate the integer-encoded 'label' column
    # from the existing df_bert_train DataFrame
    'alpha': ['a'] * df_bert_train.shape[0], # Generate a dummy 'alpha' column with the
    # same letter 'a' for all rows
    'text': df_bert_train['text'].replace(r'\n', ' ', regex=True) # Introduce a 'text'
    # column containing the text from the 'text' column, substituting newline characters
    # with spaces
})

# Showcase the first five rows of the newly established DataFrame
df_bert_train.head()
```

Output:

	id	label	alpha	text
0	0	0	a	Unfortunately, the frustration of being Dr. Go...
1	1	1	a	Been going to Dr. Goldberg for over 10 years. ...
2	2	0	a	I don't know what Dr. Goldberg was like before...
3	3	0	a	I'm writing this review to give you a heads up...
4	4	1	a	All the food is great here. But the best thing...

Input:

```
# Construct a new DataFrame for testing in compliance with BERT's specifications
df_bert_test = pd.DataFrame({
    'id': range(len(df_bert_test)), # Create an 'id' column with a sequence of integers
    # from 0 to the length of df_bert_test
    'label': df_bert_test['label'], # Incorporate the integer-encoded 'label' column
    # from the existing df_bert_test DataFrame
    'alpha': ['a'] * df_bert_test.shape[0], # Generate a dummy 'alpha' column with the
    # same letter 'a' for all rows
})
```

```
'text': df_bert_test['text'].replace(r'\n', ' ', regex=True) # Introduce a 'text' column containing the text from the 'text' column, substituting newline characters with spaces
})

# Showcase the first five rows of the newly established DataFrame
df_bert_test.head()
```

Output:

	id	label	alpha	text
0	0	1	a	Contrary to other reviews, I have zero complai...
1	1	0	a	Last summer I had an appointment to get new ti...
2	2	1	a	Friendly staff, same starbucks fair you get an...
3	3	0	a	The food is good. Unfortunately the service is...
4	4	1	a	Even when we didn't have a car Filene's Baseme...

Input:

```
# Split the train set further into train and dev (development/validation) sets
df_bert_train, df_bert_dev = train_test_split(df_bert_train, test_size=0.01)

# Display the first five rows of each set (train, test, and dev) by calling the head() method
df_bert_train.head(), df_bert_test.head(), df_bert_dev.head()
```

Output:

(id	label	alpha	text
417837	417837	1	a	Really good food and great service. I wouldn't...	
383622	383622	1	a	Ah, happiness is a new buffet restaurant in to...	
200378	200378	1	a	Sure there's lots of stuff to do inside this e...	
216246	216246	0	a	Sylvia, what happened?\nWe normally do take ou...	
439684	439684	0	a	First of all, after being told \"Oh my Gosh, I....,	
		id	label	alpha	text
0	0	1	a	Contrary to other reviews, I have zero complai...	
1	1	0	a	Last summer I had an appointment to get new ti...	
2	2	1	a	Friendly staff, same starbucks fair you get an...	
3	3	0	a	The food is good. Unfortunately the service is...	
4	4	1	a	Even when we didn't have a car Filene's Baseme...,	
		id	label	alpha	text
90605	90605	0	a	We are here for spring training. The Potsticke...	
16654	16654	1	a	A really nice place, attentive service, and no...	
507856	507856	1	a	The location is superb of this frozen yogurt p...	
103976	103976	0	a	I have to give a star but if I could, I would ...	
181086	181086	1	a	Normally stop in here after eating at Aloha Sp...	

Input:

```
# Save the DataFrames to .tsv format, as required by BERT
df_bert_train.to_csv('.../bert/data/train.tsv', sep='\t', index=False, header=False)
df_bert_dev.to_csv('.../bert/data/dev.tsv', sep='\t', index=False, header=False)
df_bert_test.to_csv('.../bert/data/test.tsv', sep='\t', index=False, header=False)
```

We saved the *.tsv file in the “bert/data” folder.

With our terminal, we can go to the “bert” folder and type the following command line:

```
python run_classifier.py --task_name=cola -do_train=true -do_eval=true
-do_predict=true -data_dir=./data/ -vocab_file=./cased_L-12_H-768_A-12/vocab.txt
-bert_config_file=./cased_L-12_H-768_A-12/bert_config.json -init_checkpoint=./
cased_L-12_H-768_A-12/bert_model.ckpt -max_seq_length=128 -train_batch_size=32
-learning_rate=2e-5 -num_train_epochs=5.0 -output_dir=./bert_output/
-do_lower_case=False
```

The training process of the model may require some time, depending on our computer’s capabilities, and the parameters we have chosen. While training, the outputs displayed in your terminal will resemble the following:

```
I0501 09:03:06.609829 4403074560 tpu_estimator.py:2307] global_step/sec: 0.0889941
INFO:tensorflow:examples/sec: 2.84781
I0501 09:03:06.609952 4403074560 tpu_estimator.py:2308] examples/sec: 2.84781
INFO:tensorflow:global_step/sec: 0.0870115
I0501 09:03:18.102550 4403074560 tpu_estimator.py:2307] global_step/sec: 0.0870115
INFO:tensorflow:examples/sec: 2.78437
I0501 09:03:18.102677 4403074560 tpu_estimator.py:2308] examples/sec: 2.78437
INFO:tensorflow:global_step/sec: 0.0868483
I0501 09:03:29.616887 4403074560 tpu_estimator.py:2307] global_step/sec: 0.0868483
INFO:tensorflow:examples/sec: 2.77915
I0501 09:03:29.617010 4403074560 tpu_estimator.py:2308] examples/sec: 2.77915
INFO:tensorflow:global_step/sec: 0.0891478
I0501 09:03:40.834223 4403074560 tpu_estimator.py:2307] global_step/sec: 0.0891478
INFO:tensorflow:examples/sec: 2.85273
I0501 09:03:40.834474 4403074560 tpu_estimator.py:2308] examples/sec: 2.85273
```

We can find the results of our model in “bert_output.”

The file test_results.tsv is created in the output folder following the prediction process on the test dataset. This file includes the predicted probability values associated with each class label. In case we aim to generate predictions on a new test dataset, test.tsv, we must first complete the model training. Afterward, navigate to the bert_output directory and identify the model.ckpt file with the highest number, as these files hold the weights of the trained model. With the highest checkpoint number in hand, execute the run_classifier.py once more, but this time, set the init_checkpoint to the highest model checkpoint.

Then, we can type the following command lines in our terminal:

```
export TRAINED_MODEL_CKPT=./bert_output/model.ckpt-413

python run_classifier.py --task_name=cola -do_predict=true -data_dir=./data
-vocab_file=./cased_L-12_H-768_A-12/vocab.txt -bert_config_file=./cased_L-12_H-
768_A-12/bert_config.json -init_checkpoint=$TRAINED_MODEL_CKPT -max_seq_length=128
-output_dir=./bert_output
```

Upon execution, a file named test_results.tsv will be produced, containing a number of columns equivalent to the total count of class labels.

For a comprehensive insight into BERT, it is highly recommended to consult the initial research paper, Devlin et al. (2019), as well as the accompanying open-source GitHub repository (<https://github.com/google-research/bert>). Furthermore, an alternative implementation of BERT can be found within the PyTorch framework. We can establish a virtual environment containing the necessary packages. We may use any package or environment manager. For example, Conda can be utilized.

```
conda create -n bert python pytorch pandas tqdm
conda install -c anaconda scikit-learn
```

We can also install the PyTorch variant of BERT provided by Hugging Face.

```
pip install pytorch-pretrained-bert
```

4.6 Utilizing BERT for Text Summarization

Text Summarization refers to the computational technique of condensing a large body of data into a concise subset (a summary) that encapsulates the most crucial and pertinent information found in the original content. In this example, we will use Bert Extractive Summarizer (<https://pypi.org/project/bert-extractive-summarizer/>) that we need to install:

```
pip install bert-extractive-summarize
```

The HuggingFace Pytorch transformers library is utilized by this tool to execute extractive summarizations by embedding sentences and subsequently implementing a clustering algorithm. This technique pinpoints sentences in close proximity to the cluster centroids, which are considered the most representative. To offer supplementary context, the library integrates coreference resolution methodologies using the neuralcoref library, accessible at <https://github.com/huggingface/neural-coref>. The CoreferenceHandler class facilitates the modification of the neuralcoref library's greediness level, allowing for the customization of the coreference resolution approach. In the most recent version of the bert-extractive-summarizer, if a GPU is present, CUDA is employed by default to ensure optimal computational performance.

Let us summarize the following text that we can find in Wikipedia (https://en.wikipedia.org/wiki/Machine_learning):

Learning algorithms work on the basis that strategies, algorithms, and inferences that worked well in the past are likely to continue working well in the future. These inferences can sometimes be obvious, such as ‘since the sun rose every morning for the last 10,000 days, it will probably rise tomorrow morning as well.’ Other times, they can be more nuanced, such as ‘X % of families have geographically separate species with color variants, so there is a Y% chance that undiscovered black swans exist’.

Machine learning programs can perform tasks without being explicitly programmed to do so. It involves computers learning from data provided so that they carry out certain tasks. For simple tasks assigned to computers, it is possible to program algorithms telling the machine how to execute all steps required to solve the problem at hand; on the computer’s part, no learning is needed. For more advanced tasks, it can be challenging for a human to manually create the needed algorithms. In practice, it can turn out to be more effective to help the machine develop its own algorithm, rather than having human programmers specify every needed step.

The discipline of machine learning employs various approaches to teach computers to accomplish tasks where no fully satisfactory algorithm is available. In cases where vast numbers of potential answers exist, one approach is to label some of the correct answers as valid. This can then be used as training data for the computer to improve the algorithm(s) it uses to determine correct answers. For example, to train a system for the task of digital character recognition, the MNIST dataset of handwritten digits has often been used.”

Input:

```
# Import the Summarizer class from the summarizer module
from summarizer import Summarizer

# Define a multi-line string variable 'body' containing text
body = '''

Learning algorithms work on the basis that strategies, algorithms, and inferences that worked well in the past are likely to continue working well in the future. These inferences can sometimes be obvious, such as "since the sun rose every morning for the last 10,000 days, it will probably rise tomorrow morning as well". Other times, they can be more nuanced, such as "X% of families have geographically separate species with color variants, so there is a Y% chance that undiscovered black swans exist". Machine learning programs can perform tasks without being explicitly programmed to do so. It involves computers learning from data provided so that they carry out certain tasks. For simple tasks assigned to computers, it is possible to program algorithms telling the machine how to execute all steps required to solve the problem at hand; on the computer's part, no learning is needed. For more advanced tasks, it can be challenging for a human to manually create the needed algorithms. In practice, it can turn out to be more effective to help the machine develop its own algorithm, rather than having human programmers specify every needed step.

The discipline of machine learning employs various approaches to teach computers to accomplish tasks where no fully satisfactory algorithm is available. In cases where vast numbers of potential answers exist, one approach is to label some of the correct answers as valid. This can then be used as training data for the computer to improve the algorithm(s) it uses to determine correct answers. For example, to train a system for the task of digital character recognition, the MNIST dataset of handwritten digits has often been used.

'''

# Instantiate the Summarizer class
model = Summarizer()

# Call the summarizer model on the 'body' text with a specified minimum summary length
# (in this case, 60 characters)
result = model(body, min_length=60)

# Join the resulting summarized text and store it in the 'full' variable
full = ".join(result)

# Print the final summarized text
print(full)
```

Output:

Learning algorithms work on the basis that strategies, algorithms, and inferences that worked well in the past are likely to continue working well in the future. For more advanced tasks, it can be challenging for a human to manually create the needed algorithms. For example, to train a system for the task of digital character recognition, the MNIST dataset of handwritten digits has often been used.

4.7 Utilizing BERT for Question Answering

A variety of machine learning techniques have been developed to tackle the problem of question answering through NLP. Previously, the bag-of-words method was prevalent, centering on responding to pre-established queries designed by developers. This approach necessitated considerable effort on the part of developers to generate questions and their respective answers. Although beneficial for chatbots, the bag-of-words method faced difficulties in managing inquiries related to extensive databases. Presently, the domain of NLP is predominantly influenced by transformer-based models such as BERT, which have notably enhanced the field's capabilities.

Let us take an example.

First, we install transformers through our terminal:

```
pip install transformers
```

Input:

```
import torch
from transformers import BertForQuestionAnswering
from transformers import BertTokenizer

# Load the pre-trained BERT model for Question Answering
model = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-
masking-finetuned-squad')

# Load the pre-trained BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased-whole-word-masking-
finetuned-squad')

# Define the question and context (body) as strings
question = "What is Machine Learning?"
body = ''' Learning algorithms work on the basis that strategies, algorithms, and
inferences that worked well in the past are likely to continue working well in the
future. These inferences can sometimes be obvious, such as "since the sun rose every
morning for the last 10,000 days, it will probably rise tomorrow morning as well".
Other times, they can be more nuanced, such as "X% of families have geographically
separate species with color variants, so there is a Y% chance that undiscovered black
swans exist".

Machine learning programs can perform tasks without being explicitly programmed to do
so. It involves computers learning from data provided so that they carry out certain
tasks. For simple tasks assigned to computers, it is possible to program algorithms
telling the machine how to execute all steps required to solve the problem at hand; on
the computer's part, no learning is needed. For more advanced tasks, it can be
challenging for a human to manually create the needed algorithms. In practice, it can
turn out to be more effective to help the machine develop its own algorithm, rather
than having human programmers specify every needed step.

The discipline of machine learning employs various approaches to teach computers to
accomplish tasks where no fully satisfactory algorithm is available. In cases where
vast numbers of potential answers exist, one approach is to label some of the correct
answers as valid. This can then be used as training data for the computer to improve
the algorithm(s) it uses to determine correct answers. For example, to train a system
for the task of digital character recognition, the MNIST dataset of handwritten digits
has often been used. '''
```

```

# Encode the question and context using the tokenizer
encoding = tokenizer.encode_plus(text=question, text_pair=body)

# Extract the input IDs (token embeddings) and token type IDs (segment embeddings)
inputs = encoding['input_ids']
sentence_embedding = encoding['token_type_ids']

# Convert input IDs to tokens
tokens = tokenizer.convert_ids_to_tokens(inputs)

# Obtain the start and end scores for the answer span from the BERT model
start_scores, end_scores = model(input_ids=torch.tensor([inputs]),
token_type_ids=torch.tensor([sentence_embedding]), return_dict=False)

# Find the indices of the highest start and end scores
start_index = torch.argmax(start_scores)
end_index = torch.argmax(end_scores)

# Extract the answer tokens from the token list
answer = ' '.join(tokens[start_index:end_index+1])

# Initialize an empty string for the corrected answer
corrected_answer = ""

# Iterate through each word in the answer and correct subword tokens
for word in answer.split():
    # If it's a subword token, remove the '##' prefix and append to the corrected answer
    if word[0:2] == '##':
        corrected_answer += word[2:]
    # Otherwise, add a space before the word and append to the corrected answer
    else:
        corrected_answer += ' ' + word

# Print the final corrected answer
print(corrected_answer)

```

Output:

computers learning from data provided so that they carry out certain tasks . for simple tasks assigned to computers , it is possible to program algorithms telling the machine how to execute all steps required to solve the problem at hand ; on the computer ' s part , no learning is needed . for more advanced tasks , it can be challenging for a human to manually create the needed algorithms . in practice , it can turn out to be more effective to help the machine develop its own algorithm , rather than having human programmers specify every needed step . the discipline of machine learning employs various approaches to teach computers to accomplish tasks where no fully satisfactory algorithm is available

Further Reading

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational*

Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186. Minneapolis, Minnesota: Association for Computational Linguistics.

Miller, D. (2019). Leveraging BERT for extractive text summarization on lectures. *Computer Science*, arXiv, <https://doi.org/10.48550/arXiv.1906.04165>.

<https://Www.Nltk.Org>.

<https://Spacy.Io>.

<https://Www.Kaggle.Com/Code/Ranjitmishra/Sms-Spam-Collection-Natural-Language-Processing>.

<https://Archive.Ics.Uci.Edu/MI/Datasets/SMS+Spam+Collection>.

<http://Www.Grubbletext.Co.Uk>.

<http://Www.Esp.Uem.Es/Jmgomez/Smsspamcorpus/>

<https://www.kaggle.com/code/drvaibhavkumar/twitter-data-analysis-using-tweepy>.

<https://research.ibm.com/blog?tag=foundation-models>.

<https://medium.com/@samia.khalid/bert-explained-a-complete-guide-with-theory-and-tutorial-3ac9ebc8fa7c>.

[https://en.wikipedia.org/wiki/BERT_\(language_model\)](https://en.wikipedia.org/wiki/BERT_(language_model)).

<https://github.com/google-research/bert>.

<https://medium.com/swlh/a-simple-guide-on-using-bert-for-text-classification-bbf041ac8d04>.

<https://pypi.org/project/bert-extractive-summarizer/>

5

Machine Learning Algorithms in Quantum Computing



Photo by Annamária Borsos

The idea of designing and developing quantum computers first emerged in the early 1980s under the impetus of physicist and Nobel Prize winner Richard Feynman. While a “classical” computer operates with bits of values 0 or 1, a quantum computer uses the fundamental properties of quantum physics and relies on “quantum bits (qubits).” Beyond this technological feat, quantum computing opens the way to processing computational tasks whose complexity is beyond the reach of our current computers.

At the beginning of the twentieth century, the theories of so-called classical physics were unable to explain certain problems observed by physicists. Therefore, they had to be reformulated and enriched. With the impetus of scientists, physics evolved in the first place toward a “new mechanics” that would become “wave mechanics” and finally “quantum mechanics.”

Quantum mechanics is the mathematical and physical theory that describes the fundamental structure of matter and the evolution in time and space of phenomena on a microscopic scale. An essential notion of quantum mechanics is the “wave–particle duality.” Until the 1890s, physicists considered that the world was composed of two types of objects or particles: those that have mass (such as electrons, protons, neutrons, and atoms) and those that do not (such as photons, waves, etc.). For the physicists of the time, these particles were governed by the laws of Newtonian mechanics for those that have mass and by the laws of electromagnetism for waves. Therefore, we had two theories of “physics” to describe two different types of objects.

Quantum mechanics invalidated this dichotomy and introduced the fundamental idea of “wave–particle duality”: a particle of matter or a wave must be treated by the same laws of physics. This was the advent of wave mechanics, which

would become quantum mechanics a few years later. The development of quantum mechanics was associated with great names such as Niels Bohr, Paul Dirac, Albert Einstein, Werner Heisenberg, Max Planck, Erwin Schrödinger, and many others.

Max Planck and Albert Einstein, by studying the radiation emitted by a heated body and the photoelectric effect, were the first to understand that exchanges of light energy could only occur in “packets” and not with any value. This is similar to a staircase that only allows you to go up by the height of one step (or more) but not to reach any height between two steps. Moreover, Albert Einstein was awarded the Nobel Prize in Physics for his theory on the quantized aspect of energy exchanges in 1921 and not for the theory of relativity.

Niels Bohr extended the quantum postulates of Planck and Einstein from light to matter, proposing a model that reproduced the spectrum of the hydrogen atom. He was awarded the Nobel Prize in Physics in 1922, defining a model of the atom that dictates the behavior of light quanta. Step by step, rules were found to calculate the properties of atoms, molecules, and their interactions with light. From 1925 to 1927, a series of works by several physicists and mathematicians led to two general theories applicable to these problems: Louis de Broglie’s wave mechanics and especially Erwin Schrödinger’s wave mechanics, and Werner Heisenberg, Max Born, and Pascual Jordan’s matrix mechanics. These two mechanics were unified by Erwin Schrödinger from a physical point of view and by John von Neumann from a mathematical point of view. Finally, Paul Dirac formulated the complete synthesis or rather generalization of these two mechanics, which is now called quantum mechanics. The fundamental equation of quantum mechanics is the Schrödinger equation:

$$H(t)|\psi(t)\rangle = i\hbar \frac{d}{dt} |\psi(t)\rangle$$

Prior to delving into the development of a “Quantum Information Theory,” let us revisit the operation of a standard computer and the “Classical Information Theory” that governs the functioning of computers as we know them today. The first binary computers were built in the 1940s, including Colossus (1943) and ENIAC (IBM – 1945). Colossus was designed to decipher secret German messages, while ENIAC was created for calculating ballistic trajectories. In 1945, ENIAC (an acronym for Electronic Numerical Integrator and Computer) became the first entirely electronic computer constructed to be “Turing-complete,” meaning it could be reprogrammed to solve any computational problem, in principle. ENIAC was programmed by women, referred to as the “ENIAC women,” with the most renowned being Kay McNulty, Betty Jennings, Betty Holberton, Marlyn Wescoff, Frances Bilas, and Ruth Teitelbaum. Prior to this, these women had performed ballistic calculations using mechanical desktop computers for the military. ENIAC weighed 30 tons, occupied an area of 72 m², and consumed 140 kW.

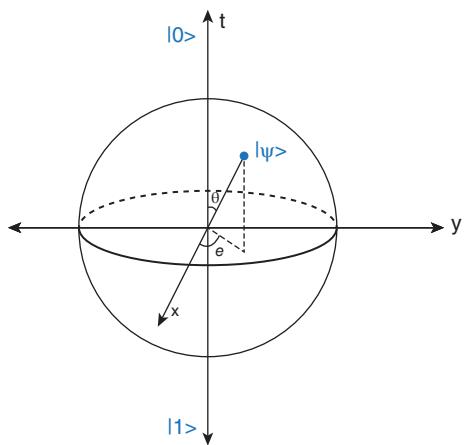
Regardless of the task performed by a computer, the underlying process remains consistent: an instance of the task is described by an algorithm that is translated into a series of 0s and 1s, which is then executed in the computer’s processor, memory, and input/output devices. This is the foundation of binary calculation, which practically relies on electrical circuits equipped with transistors that can be in two modes: “ON” allowing current to flow and “OFF” preventing current flow.

Over the past 80 years, a classical information theory has been developed based on these 0s and 1s, constructed from Boolean operators (XAND, XOR), words (bytes), and simple arithmetic operations such as “0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1 = 0 (with a carry),” and verifying if 1 = 1, 0 = 0, and 1 ≠ 0. Naturally, more complex operations can be constructed from these basic operations, allowing computers to perform trillions of such operations per second.

These 0s and 1s are contained in “BInary digiTs” or “bits,” which represent the smallest quantity of information within a computer system.

For a quantum computer, the “qubit (quantum bit)” serves as the fundamental entity, representing the smallest unit capable of manipulating information, analogous to the “bit.” A qubit possesses key properties of quantum mechanics such as superposition, entanglement, measurement, or inference.

Superposition is about creating a quantum state that is a combination of |0> and |1>. A quantum object (at the microscopic scale) can exist in an infinite number of states (as long as its state is not measured). Consequently, a qubit can exist in any state between 0 and 1. Qubits can simultaneously assume the value 0 and 1, or more accurately, “a certain amount of 0 and a certain amount of 1,” as a linear combination of two states denoted |0> and |1>, with coefficients α and β . Hence, while a classical bit can only represent two states (0 or 1), a qubit can represent an “infinity” of states. This is one of the potential advantages of quantum computing from an information theory perspective (**n-qubits** – 2^n quantum state dimensions).



The vectors $|0\rangle$ and $|1\rangle$ are situated in a two-dimensional complex vector space, known as C^2 :

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

As a result, we can represent any vector in C^2 using the following formula:

$$a |0\rangle + b |1\rangle$$

The terms $|0\rangle$ and $|1\rangle$ are pronounced as “ket zero” and “ket one,” correspondingly, and are collectively recognized as the computational basis.

Superposition involves generating a quantum state that constitutes a fusion of $|0\rangle$ and $|1\rangle$:

$$a |0\rangle + b |1\rangle$$

in which a and b denote complex numbers:

$$|a|^2 + |b|^2 = 1$$

Two quantum states are considered identical if they diverge solely by a constant multiple, designated as u, where $|u| = 1$. This is attributed to the fact that

$$|a|^2 + |b|^2 = |au|^2 + |bu|^2 = 1$$

Under these constraints, we can represent the qubit on the Bloch Sphere. Observe that when both a and b are non-zero, the qubit's state comprises both $|0\rangle$ and $|1\rangle$. This concept is what is commonly referred to when people mention that a qubit can simultaneously exist as “0 and 1.”

The act of measuring forces the qubit's state, which is a $|0\rangle + b |1\rangle$, to collapse into either $|0\rangle$ or $|1\rangle$ upon observation. In this scenario, $|a|^2$ represents the likelihood of obtaining $|0\rangle$ upon measurement and $|b|^2$ represents the likelihood of obtaining $|1\rangle$ upon measurement.

For instance:

$$\frac{\sqrt{2}}{2} |0\rangle + \frac{\sqrt{2}}{2} |1\rangle$$

possesses an equal probability of collapsing into either $|0\rangle$ or $|1\rangle$.

Another example:

$$\frac{\sqrt{3}}{2} |0\rangle - \frac{1}{2} i |1\rangle$$

has a 75% chance of collapsing into $|0\rangle$.

Involving two qubits results in combinations such as

$$a |00\rangle + b |01\rangle + c |10\rangle + d |11\rangle,$$

where $|01\rangle$ implies the first qubit is in state $|0\rangle$ while the second is in state $|1\rangle$; a, b, c , and d denote complex numbers:

$$|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$$

If two or more of the values a, b, c , and d are non-zero, and it is not possible to separate the qubits, they become entangled, exhibiting perfect correlation and losing their independence.

For example,

$$\frac{\sqrt{2}}{2} |00\rangle + \frac{\sqrt{2}}{2} |01\rangle \quad \text{not entangled}$$

$$\frac{\sqrt{2}}{2} |01\rangle - \frac{\sqrt{2}}{2} |10\rangle, \frac{\sqrt{2}}{2} |00\rangle + \frac{\sqrt{2}}{2} |11\rangle \quad \text{entangled}$$

We can express

$$\frac{\sqrt{2}}{2} |00\rangle + \frac{\sqrt{2}}{2} |01\rangle$$

as

$$|0\rangle (\frac{\sqrt{2}}{2} |0\rangle + \frac{\sqrt{2}}{2} |1\rangle)$$

but it is impossible to represent

$$\frac{\sqrt{2}}{2} |00\rangle + \frac{\sqrt{2}}{2} |11\rangle$$

as a combination of two individual qubit states, which means they are entangled. Upon measuring the initial qubit, the second qubit becomes distinctly defined.

Quantum gates are the fundamental building blocks of quantum computing, analogous to classical logic gates in classical computing. They are used to manipulate quantum bits, or qubits, by applying specific transformations to their quantum states. Quantum gates operate on the principles of quantum mechanics, such as superposition and entanglement, allowing for more complex and powerful computations than their classical counterparts.

Some examples of common quantum gates are:

- 1) **Pauli-X gate:** This is the quantum equivalent of the classical NOT gate. It flips the state of a qubit, transforming $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$.
- 2) **Hadamard gate:** This gate is used to create superposition in a qubit. When applied to a qubit in either the $|0\rangle$ or $|1\rangle$ state, it generates an equal superposition of both states, resulting in $(|0\rangle + |1\rangle)/\sqrt{2}$ or $(|0\rangle - |1\rangle)/\sqrt{2}$, respectively.
- 3) **Pauli-Y gate:** Similar to the Pauli-X gate, the Pauli-Y gate flips the qubit states but also adds a complex phase. The transformation maps $|0\rangle$ to $i|1\rangle$ and $|1\rangle$ to $-i|0\rangle$, where i is the imaginary unit.
- 4) **Pauli-Z gate:** The Pauli-Z gate is a phase-flip gate that adds a phase of π to the $|1\rangle$ state without affecting the $|0\rangle$ state. It maps $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $-|1\rangle$.
- 5) **CNOT gate (Controlled-NOT):** This is a two-qubit gate where the first qubit acts as the control, and the second qubit is the target. If the control qubit is in state $|1\rangle$, the target qubit's state is flipped. If the control qubit is in state $|0\rangle$, the target qubit remains unchanged.

These quantum gates, among others, can be combined and applied to qubits to perform various quantum computing tasks and algorithms.

In quantum computing, the term “inference” generally refers to the process of extracting useful information or making predictions based on the manipulation and measurement of qubits in a quantum system. This concept is particularly relevant in the context of quantum machine learning, where quantum algorithms are used to analyze and process data to gain insights, make predictions, or classify new data points. Quantum inference leverages the principles of quantum mechanics

such as superposition and entanglement to enable more efficient data processing and potentially faster solutions than classical methods. Quantum algorithms such as Grover's search algorithm and quantum phase estimation can be applied to perform inference tasks, which may provide speedup over their classical counterparts.

5.1 Quantum Machine Learning

Machine learning is changing the way businesses operate in fundamental ways and bringing new opportunities for progress, alongside challenges. The capabilities of machine learning to interpret and analyze data have greatly increased. Yet, machine learning is also demanding in terms of computing power because of more and more data to process and the complexity of workflows. Machine learning and quantum computing are two technologies that can potentially allow us to solve complex problems, previously untenable, and accelerate areas such as model training or pattern recognition. The future of computing will certainly be made up of classical, biologically inspired, and quantum computing. The intersection between quantum computing and machine learning has received considerable attention in recent years and has allowed the development of quantum machine learning algorithms such as quantum-enhanced support vector machine (QSVM), QSVM multiclass classification, variational quantum classifier (VQC), or quantum generative adversarial networks (qGANs). Since the birth of quantum computing, scientists have been searching for the best places to apply quantum algorithms. The first two quantum algorithms published by Shor (1994) and Grover (1996) demonstrated that applying them to factorization and theoretical searching could produce an advantage in comparison with classical computing. The study of machine learning problems with quantum techniques is a new and active area of research. For example, a quantum neural network (QNN) had been defined in general terms but was only defined at the physical level in 2000 by Ezhov and Ventura (2000). In 2003, an approach was proposed by Rick and Ventura to train QNNs, but the method was exponentially complex. The introduction of quantum computing into clustering, distributed semantics, and SVMs by Lloyd et al. (2013), Blacoe et al. (2013), and Rebentrost et al. (2014) was also limited too much to theory. The emergence of physical implementation of quantum computers such as those of IBM has made it possible to translate research theory on quantum machine learning algorithms into practice.

It is clear now that quantum computers have the potential to boost the performance of machine learning algorithms and may contribute to breakthroughs in different fields such as drug discovery or fraud detection. Data can exhibit structures that are difficult to identify, reducing classification accuracy. The idea is to find better patterns within machine learning/deep learning processes by leveraging quantum systems that map data to higher dimensions for training and use. In a recent paper, Havlíček et al. (2019) propose and describe the experimental implementation of two quantum algorithms on a superconducting processor. Both algorithms solve a problem of supervised learning, which is the construction of a classifier. Like conventional SVMs, the quantum variational classifier uses a variational quantum circuit to classify data and the quantum kernel estimator estimates the kernel function and optimizes a classical SVM. The reason we are exploring this area is because when we use classical SVMs, we can be limited if the feature space is very large. In this case, the kernel functions are computationally expensive to estimate. A key element in the paper is the use of the quantum state space as feature space, as we can exploit the exponentially large quantum state space through controllable entanglement and interference. This is in preparation for the quantum advantage, which refers to solving practically relevant challenges better or faster with quantum computing in comparison to classical computers with the best-known hardware and the best-known classical solutions. It is still an open question whether near-term quantum computers can be advantageous for machine learning tasks. Clearly, many elements need to be resolved, but there is a path to improve training by considering more dimensions than possible today and decreasing computational time, for example, by algorithms using kernel methods (linear, exponential, Gaussian, hyperbolic, angular functions, etc.).

There are also hyperparameters in the machine learning models that need to be addressed, including regularization, batch size, learning speed, and others. For example, part of establishing an artificial neural network is deciding how many layers of hidden nodes will be used between the input layer and the output layer. Hyperparameter optimization is also an area in which quantum computing could potentially assist.

Liu et al. (2017) provided more than assumptions by describing the construction of a classification problem and rigorously proving a quantum speedup. The authors show that no classical learner can classify data inverse-polynomially better than random guessing. They found mathematical proof of a quantum advantage for machine learning by developing a specific

task for which quantum kernel methods are better than classical methods. What does “better” mean here? *Its quantum advantage comes from the fact that we can construct a family of datasets for which only quantum computers can recognize the intrinsic labeling patterns, while for classical computers the dataset looks like random noise.* When they visualized the data in a quantum feature map, the quantum algorithm was able to predict the labels very quickly and accurately. The idea was to create classification problems based on discrete log, compute logarithms in a cyclic group, in which it is possible to generate all the members of the group using a single mathematical operation. This specific problem can be solved by using Shor’s algorithm, which would take a superpolynomial amount of time on a classical computer. Of course, we are talking about a very specific problem in which the classification problem needs to fit into the cyclical structure. In real life, most quantum algorithms do not perform better than conventional ones run on classical computers; there is room for improvement.

In the machine learning/deep learning worlds, we can evoke the following potential use cases for quantum computing:

- **Chemicals and petroleum:** drilling locations, seismic imaging.
- **Distribution and logistics:** recommendations of consumer offers, freight forecasting, irregular behaviors.
- **Financial services:** recommendations of finance offers, credit and asset scoring, irregular behaviors (fraud).
- **Health care and life science:** accelerated diagnosis, genomic analysis, clinical trial enhancements, medical image processing.
- **Manufacturing:** quality control, structural design, fluid dynamics.

The Large Hadron Collider accelerates beams of protons that collide a billion times per second at near-light speed, creating particles in the proton chaos measured by detectors. It produces 1 petabyte of data per second, which requires a million classical CPU cores in 170 locations across the world. To be detectable, data from detectors should be processed by complex computation. In a recent study, CERN and IBM investigated how to use quantum machine learning to detect and analyze the Higgs boson (a particle that helps understand the origin of mass). Finding occurrences of the Higgs boson within a maelstrom of data seriously challenges the limits of classical computers. In their study, the scientists showed that a general-purpose quantum kernel algorithm that had not been optimized for particle physics, contrary to the classical machine learning algorithms that had been optimized by CERN, was able to match CERN’s best classical algorithms. The algorithm used was a QSVM.

To better understand quantum machine learning, we will use the open-source SDK Qiskit (<https://qiskit.org>) that allows working with quantum computers and provides many algorithms and use cases. Qiskit supports Python version 3.6 or later. The goal of this chapter is to apply quantum machine learning algorithms to real datasets. To understand the different concepts, we can explore qiskit.org.

To install the environment, we can go to https://qiskit.org/documentation/getting_started.html.

In our environment, we can install Qiskit with pip:

```
pip install qiskit
```

We can also install qiskit-machine-learning using pip:

```
pip install qiskit-machine-learning
```

Documentation can be found on GitHub: <https://github.com/Qiskit/qiskit-machine-learning>.

To run our code, we can use either simulators or real hardware. For example, we could use the 127-qubit superconducting quantum computers available at IBM Quantum Services to run quantum algorithms such as `ibm_sherbrooke`. The qubit connectivity is shown in Figure 5.1 and the characteristics of each quantum system in Table 5.1.

If we go to hephAistos/Notebooks with a terminal and open a Jupyter Notebook, we will see in a browser two files named `Quantum_Kernel_Methods.ipynb` and `Quantum_Neural_Network.ipynb` that contain all code examples shown in this section. The notebooks can also be downloaded at https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Quantum_Kernel_Methods.ipynb and https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/Quantum_Neural_Network.ipynb.

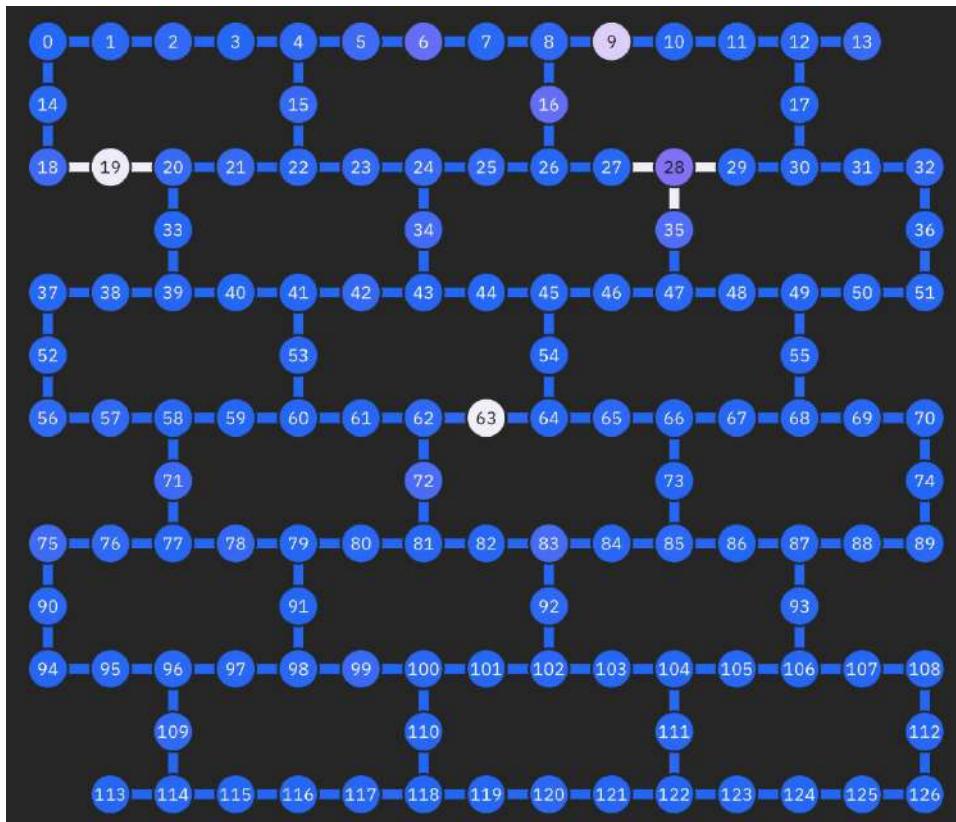


Figure 5.1 Qubit connectivity of the 127-qubit superconducting quantum computer (ibm_sherbrooke).

Table 5.1 Characteristics of each quantum system.

	ibmq_kolkata	ibmq_montreal	ibmq_mumbai	ibm_auckland
Processor type	Falcon r5.11	Falcon r4	Falcon r5.10	Falcon r5.11
Qubits	27	27	27	27
Quantum volume	128	128	128	128
CLOPS	2K	2K	1.8K	2.4K
Basis gates	CX, ID, RZ, SX, X			
Average CNOT error	8.092e-2	1.285e-2	1.184e-1	8.634e-2
Average readout error	2.912e-2	3.485e-2	3.178e-2	1.977e-2
Average T1	132.12 us	121.24 us	117.96 us	157.84 us
Average T2	90.01 us	100.6 us	100.22 us	130.58 us

While studying the Qiskit documentation, you will encounter references to the Qiskit Runtime primitives, which serve as implementations of the Sampler and Estimator interfaces found in the `qiskit.primitives` module. These interfaces facilitate the seamless interchangeability of primitive implementations with minimal code modifications. Various primitive implementations can be found within the `qiskit`, `qiskit_aer`, and `qiskit_ibm_runtime` libraries, each designed to cater to specific requirements:

- The primitives in the `qiskit` library enable efficient local state vector simulations, offering expedient algorithm prototyping capabilities.

- The primitives in the qiskit_aer library provide access to local Aer simulators, which are instrumental in conducting simulations involving noise.
- The primitives in the qiskit_ibm_runtime library grant access to cloud simulators and real quantum hardware through the Qiskit Runtime service. These primitives incorporate exclusive features such as built-in circuit optimization and error mitigation support.

Primitives constitute core functions that facilitate the construction of modular algorithms and applications, delivering outcomes that transcend mere count values by providing more immediate and meaningful information. Moreover, they offer a seamless pathway to harness the latest advancements in IBM Quantum hardware and software.

The initial release of Qiskit Runtime comprises two essential primitives:

- **Sampler:** This primitive generates quasi-probabilities based on input circuits.
- **Estimator:** This primitive calculates expectation values derived from input circuits and observables.

For more comprehensive insights, detailed information is available in the following resource: <https://qiskit.org/ecosystem/ibm-runtime/tutorials/how-to-getting-started-with-sampler.html>.

5.2 Quantum Kernel Machine Learning

Since the beginning of their studies, scientists have been looking for the best applications and output of quantum algorithms. As described above, the intersection between quantum computing and machine learning has received widespread attention in recent years and has allowed the development of quantum machine learning algorithms such as QSVM, QVC, and QNN.

Solving supervised machine learning problems with quantum techniques is a new area of research. In classical machine learning, kernel methods are widely used, and the use of support vector machine (SVM) for classification is one of the most frequent applications. SVMs have been widely used as binary classifiers and applied in recent years to solving multiclass problems. In binary SVM, the objective is to create a hyperplane that linearly divides n -dimensional data points into two components by searching for an optimal margin that correctly segregates the data into different classes. The hyperplane that divides the input dataset into two groups can be either in the original feature space or in a higher-dimensional kernel space. The selected optimal hyperplane from among many hyperplanes that might classify the data corresponds to the hyperplane with the largest margin that allows the largest separation between the different classes. It is an optimization problem under constraints in which the distance between the nearest data point and the optimal hyperplane (on each side) is maximized. The hyperplane is then called the maximum-margin hyperplane, allowing creation of a maximum-margin classifier. The closest data points are known as support vectors, and the margin is an area that generally does not contain any data points. If the hyperplane defined as optimal is too close to the data points and the margin too small, it will be difficult to predict new data and the model will fail to generalize well.

Several methods have been proposed to build multiclass SVMs based on binary SVM such as the all-pair approach in which a binary classification problem for each pair of classes is used. In addition to linear classification, it is also possible to compute a nonlinear classification using what is commonly called the kernel trick (a kernel function) that maps inputs into high-dimensional feature spaces. The kernel function corresponds to an inner product of vectors in a potentially high-dimensional Euclidian space referred to as the feature space. The objective of nonlinear SVM is to gain separation by mapping data to higher-dimensional space because many classification or regression problems are not linearly separable or regressable in the space of the inputs \mathbf{x} . The aim is to use the kernel trick to move to a higher-dimensionality feature space given a suitable mapping $\mathbf{x} \rightarrow \phi(\mathbf{x})$. To address data not tractable by linear methods, we need to choose a suitable feature map. By combining classical kernel methods and quantum models, quantum kernel methods can shape new approaches in machine learning. In early versions of quantum kernel methods, quantum feature maps encode the data points into inner products or amplitudes in the Hilbert space. The number of features determines the number of qubits, and the quantum circuit used to implement the feature map is of a length that is a linear or polylogarithmic function of the size of the dataset. The work provided thus far to document or prove the advantage of a quantum feature map has been performed by carefully choosing synthetic datasets or by application to small, binary classification problems. Data can exhibit structures that are difficult to identify; therefore classification accuracy may be reduced.

Our goal is to find better patterns using machine learning processes by leveraging quantum systems that map data to higher dimensions for training purposes and use. The principle, considering a classical data vector $\mathbf{x} \in \chi$, is to map \mathbf{x} to an n -qubit quantum feature state $|\phi(\mathbf{x})\rangle$ by a unitary encoding circuit $U(\mathbf{x})$ such as $\phi(\mathbf{x}) = U(\mathbf{x})|0^n\rangle\langle 0^n|U^\dagger(\mathbf{x})$. For two samples $\mathbf{x}, \tilde{\mathbf{x}}$, the quantum kernel function K is defined as the inner products of two quantum feature states in the Hilbert–Schmidt space $K(\mathbf{x}, \tilde{\mathbf{x}}) = \text{tr}[\phi^\dagger(\mathbf{x})\phi(\tilde{\mathbf{x}})]$ and translated as the transition amplitude $K(\mathbf{x}, \tilde{\mathbf{x}}) = |\langle 0^n | U^\dagger(\mathbf{x})U(\tilde{\mathbf{x}}) | 0^n \rangle|^2$. For instance, the kernel function can be estimated on a quantum computer by a procedure called quantum kernel estimation (QKE), which consists of evolving the initial state $|0^n\rangle$ with $U^\dagger(\mathbf{x})U(\tilde{\mathbf{x}})$ and recording the frequency of the all-zero outcome 0^n . The constructed kernel is then injected into a standard SVM. By replacing the classical kernel function with QKE, it is possible to classify data in a quantum feature space with SVM. Rebentrost et al. (2013) proposed a theoretically feasible quantum kernel approach based on SVM. In 2019, Havlíček et al. and Schuld and Killoran (2019) proposed two implementations of quantum kernel methods. The authors experimentally implemented two quantum algorithms into a superconducting processor. Like a conventional SVM, the quantum variational classifier uses a variational quantum circuit to classify data, and the quantum kernel estimator estimates the kernel function and optimizes a classical SVM. Classifying data using quantum algorithms may provide possible advantages such as increased speed for kernel computing compared to classical computing and a potential improvement in classification accuracy. To achieve these advantages, it is mandatory to find and explicitly specify a suitable feature map. This operation is not straightforward when compared to classic kernel specification. Although theoretical work has shown a demonstrable advantage on synthetically generated datasets, concrete and specific applications are needed to empirically study whether a quantum advantage may be reached and, if so, for what kinds of datasets and applications. It is now a major challenge to find quantum kernels that could potentially provide advantages on real-world datasets.

Richard Feynman suggested the use of quantum systems to efficiently simulate nature. Despite over a century of research on cortical circuits, it is still unclear and not broadly accepted how many classes of cortical neurons exist. The continuous development of new techniques and the availability of more and more data regarding phenotypes does not allow the maintenance of a unit classification system that can be simple to update and that may take into consideration all of the different characteristics of neurons. Neuronal classification remains a topic in progress because it is still unclear how to designate a neuronal cell class and what are the best features to define it. The cortical circuit is composed of glutamatergic neurons (~80 to 90%) and GABAergic neurons (~10 to 20%). GABAergic interneurons play a critical role within cortical networks although they are a minority. Interneuron phenotypes are diverse, with more than 20 distinct inhibitory interneuron cell types in rodents. They can be divided into several subtypes according to their morphology, expression of histochemical markers, molecular functions, electrophysiology, and synaptic properties.

In this chapter, we will apply quantum kernel methods to the quantitative characterization of neuronal morphologies from histological neuronal reconstructions, which represent a primary resource to address anatomical comparisons and morphometric analysis. Morphology-based classification of neuron types at the whole-brain level in the rat remains a challenge, as it is not clear how to designate a neuronal cell given the significant number of neuron types, the limited samples (reconstructed neuron), the best features by which to define them, and diverse data formats such as two- and three-dimensional images (structured with high dimensions and fewer samples than the complexity of morphologies) and SWC-format files (low-dimensional and unstructured). There are several reasons why neuroscientists are interested in this topic. Some brain diseases affect specific cell types, and current knowledge may be improved by correlating some disorders to the underlying vulnerable neuronal types. Neuron morphology studies can lead to the identification of genes to target for specific cell morphology and functions linked to them. The discovery of gene functions in specific cell types can be used as entry points. Before acquiring its form and function, a neuron goes through different stages of development that need to be understood to identify new markers, marker combinations, or mediators of developmental choices. The understanding of neuron morphologies often represents the basis of modeling efforts and data-driven modeling approaches to study the impact of a cell's morphology on its electrical behavior and on the network in which it is embedded. Classification by neuron subtypes represents a way to reduce dimensionality for modeling purposes.

Even if its principle is almost the same as the classical kernel method, the quantum kernel method is based on quantum computing properties and maps a data point from an original space to a quantum Hilbert space. The quantum mapping function is critically important in the process of quantum kernel methods, with a direct impact on model's performance. Finding a suitable feature map in the context of gate-based quantum computers is less trivial than simply specifying a suitable kernel in classical algorithms. In quantum kernel machine learning, a classical feature vector \vec{x} is mapped to a quantum Hilbert space using a quantum feature map $\Phi(\vec{x})$ such that $K_{ij} = \left| \langle \Phi^\dagger(\vec{x}_j) | \Phi(\vec{x}_i) \rangle \right|^2$. There are important

factors to evaluate when considering a feature map such as the feature map circuit depth, the data map function for encoding the classical data, the quantum gate set, and the order expansion. We can find different types of feature maps such as ZFeatureMap, which implements a first-order diagonal expansion where $|S| = 1$. We need to establish different parameters such as the feature dimensions (dimensionality of the data, which is equal to the number of required qubits), the number of times the feature map circuit is repeated (reps), and a function encoding the classical data. Here, there is no entanglement because there are no interactions between features. Another example is the ZZFeatureMap, which is a second-order Pauli-Z evolution circuit allowing $|S| \leq 2$ and Φ , a classical nonlinear function. Here, interactions in the data are encoded in the feature map according to the connectivity graph and the classical data map. Like ZFeatureMap, ZZFeatureMap needs the same parameters as well as an additional one, which is the entanglement that generates connectivity (“full,” “linear,” or its own entanglement structure). The PauliFeatureMap is the general form of the feature map that allows us to create feature maps using different gates. It transforms input data $\vec{x} \in \mathbb{R}^n$ such that the following is true:

$$U_{\Phi(\vec{x})} = \exp\left(i \sum_{S \subseteq [n]} \phi_S(\vec{x}) \prod_{i \in S} P_i\right)$$

where $P_i \in \{I, X, Y, Z\}$ denotes the Pauli matrices and S denotes the connectivities between different qubits or data points: $S \in \{\binom{n}{k} \text{ combinations}, k = 1, \dots, n\}$. For $k = 1$ and P_0 , we can refer to ZFeatureMap and ZZFeatureMap for $k = 2$ and $P_0 = Z$ and $P_0, 1 = ZZ$.

In this chapter, we test eight quantum kernel algorithms. The first one, named q_kernel_zz, applies a ZZFeatureMap. As described by Havlíček et al. (2019), we define a feature map on n -qubits generated by the unitary:

$$\mathcal{U}_{\Phi}(\vec{x}) = U_{\Phi(\vec{x})} H^{\otimes n} U_{\Phi(\vec{x})} H^{\otimes n}$$

where H denotes the conventional Hadamard gate and $U_{\phi(\vec{x})}$ is a diagonal gate in the Pauli-Z basis.

$$U_{\Phi(\vec{x})} = \exp\left(i \sum_{S \subseteq [n]} \phi_S(\vec{x}) \prod_{i \in S} Z_i\right)$$

Taking an example with two qubits, the general expression is as follows:

$$\mathcal{U}_{\Phi}(\vec{x}) = U_{\Phi(\vec{x})} H^{\otimes 2} U_{\Phi(\vec{x})} H^{\otimes 2}$$

where

$$U_{\Phi(\vec{x})} = \exp(i\phi_1(x)ZI + i\phi_2(x)IZ + i\phi_{1,2}(x)ZZ)$$

The encoding function that transforms the input data into a higher-dimensional feature space is given by the following:

$$\Phi(x) = \{\phi_1(x), \phi_2(x), \phi_{1,2}(x)\}$$

To create a feature map and test different encoding functions, we use the encoding function from Havlíček et al. (2019):

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = (\pi - x_1)(\pi - x_2) \quad (\text{q_kernel_zz})$$

For q_kernel_zz, we use the ZZFeatureMap.

A second algorithm (q_kernel_default) can be tested by applying a PauliFeatureMap (paulis = ['ZI', 'IZ', 'ZZ']) with the default data mapping ϕ_S :

$$\phi_S(\vec{x}) = \begin{cases} x_0 & \text{if } k = 1 \\ \prod_{j \in S} (\Pi - x_j) & \text{otherwise} \end{cases}$$

For q_kernel_default, we use the PauliFeatureMap.

In addition, we can use five encoding functions presented by Suzuki et al. (2020) For a two-qubit example, they are the following:

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = \pi x_1 x_2 \quad (\text{q_kernel_8})$$

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = \frac{\pi}{2}(1-x_1)(1-x_2) \quad (\text{q_kernel_9})$$

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = \exp\left(\frac{|x_1 - x_2|^2}{8/\ln(\pi)}\right) \quad (\text{q_kernel_10})$$

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = \frac{\pi}{3\cos(x_1)\cos(x_2)} \quad (\text{q_kernel_11})$$

$$\phi_{\{i\}}(x) = x_i \text{ and } \phi_{\{1,2\}}(x) = \pi \cos(x_1) \cos(x_2) \quad (\text{q_kernel_12})$$

For `q_kernel_8`, `q_kernel_9`, `q_kernel_10`, `q_kernel_11`, and `q_kernel_12`, we use the `PauliFeatureMap` (paulis = ['ZI', 'IZ', 'ZZ']).

It is also possible to train a quantum kernel with quantum kernel alignment (QKA), which iteratively adapts a parameterized quantum kernel to a dataset and converges to the maximum SVM margin at the same time (we have named it `q_kernel_training`). The algorithm introduced by Glick et al. (2021) allows, from a family of kernels (covariant quantum kernels that are related to covariant quantum measurements), the learning of a quantum kernel; at the same time, converging to the maximum SVM margin optimizes the parameters in a quantum circuit. To implement it, we prepare the dataset as usual and define the quantum feature map. We then use the `QuantumKernelTrained.fit` method to train the kernel parameters and pass it to a machine learning model. In covariant quantum kernels, the feature map is defined by a unitary representation $D(\mathbf{x})$ for $\mathbf{x} \in \chi$ and a state $|\psi\rangle = U|0^n\rangle$. The kernel matrix is given as follows:

$$K(\mathbf{x}, \tilde{\mathbf{x}}) = |\langle 0^n | U^\dagger D^\dagger(\mathbf{x}) D(\tilde{\mathbf{x}}) U | 0^n \rangle|^2$$

For a given group, the quantum kernel alignment is used to find the optimal fiducial state. In the context of covariant quantum kernels, the equation can be extended to the following:

$$K_\lambda(\mathbf{x}, \tilde{\mathbf{x}}) = |\langle 0^n | U_\lambda^\dagger D^\dagger(\mathbf{x}) D(\tilde{\mathbf{x}}) U_\lambda | 0^n \rangle|^2$$

where quantum kernel alignment will learn an optimal fiducial state parametrized by λ for a given group.

In order to train the quantum kernel, we will employ an instance of `TrainableFidelityQuantumKernel`, encompassing both the feature map and its associated parameters. Following the training of the quantum kernels, we will proceed to utilize Qiskit's Quantum Variational Classifier (QVC) for classification tasks (available at <https://qiskit.org>). The `TrainableFidelityQuantumKernel` and `QuantumKernelTrainer` will be employed for managing the training process within the context of `q_kernel_training`. Specifically, the Quantum Kernel Alignment technique will be employed during training, with the selection of the kernel loss function `SVC` as the input to `QuantumKernelTrainer`.

Given that `SVC` is a supported loss function in Qiskit, we can utilize the string representation "svc_loss"; however, it is important to note that default settings will be employed when passing the loss as a string. Should custom settings be desired, it is necessary to explicitly instantiate the desired options and pass the `KernelLoss` object to the `QuantumKernelTrainer`.

To optimize the training process, the SPSA optimizer will be selected, and the trainable parameters will be initialized using the `initial_point` argument. It is worth mentioning that the length of the list provided as the `initial_point` argument must be equal to the number of trainable parameters present in the feature map.

Let us start coding and first import the necessary libraries.

Input:

```
# Importing utilities for data processing and visualization
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

The computational execution can be tailored to utilize either local simulators (Aer simulators) or online quantum simulators and real quantum hardware. The following lines can be commented or uncommented as per the desired choice. If the intention is to run the code on a local simulator, the following lines can be added.

Input:

```
# Compute code with local simulator (Aer simulators)
from qiskit.primitives import Sampler
sampler = Sampler()
```

Alternatively, if our preference is to execute the code using online quantum simulators or real quantum hardware, the following lines can be added:

Input:

```
## Compute code with online quantum simulators or quantum hardware from the cloud
# Import QiskitRuntimeService and Sampler
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
# Define service
service = QiskitRuntimeService(channel = 'ibm_quantum', token = " YOUR TOKEN ",
instance = 'ibm-q/open/main')
# Get backend
quantum_backend = "ibmq_bogota"
backend = service.backend(quantum_backend) # Use a simulator or hardware from the
cloud
# Define Sampler with different options
# resilience_level=1 adds readout error mitigation
# execution.shots is the number of shots
# optimization_level=3 adds dynamical decoupling
from qiskit_ibm_runtime import Options
options = Options()
options.resilience_level = 1
options.execution.shots = 1024
options.optimization_level = 3
sampler = Sampler(session=backend, options = options)
```

To utilize this final option, it is essential to possess an IBM Quantum account, which can be obtained through the following link: <https://quantum-computing.ibm.com>. Signing in to the account will grant access to a personal token that can be directly incorporated into our code. Additionally, by visiting <https://quantum-computing.ibm.com>, we gain visibility into all the available online simulators and hardware systems.

We need also to set a seed for randomization to keep outputs consistent.

Input:

```
# seed for randomization, to keep outputs consistent
from qiskit.utils import algorithm_globals
seed = 123456
algorithm_globals.random_seed = seed
```

Now, we can define our encoding functions.

Input:

```
# Encoding Functions
from functools import reduce
```

```

def data_map_8(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*(m * n), x)
    return coeff

def data_map_9(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: (np.pi/2)*(m * n), 1 - x)
    return coeff

def data_map_10(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*np.exp(((n - m)*(n - m))/8), x)
    return coeff

def data_map_11(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: (np.pi/3)*(m * n), 1/(np.cos(x)))
    return coeff

def data_map_12(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*(m * n), np.cos(x))
    return coeff

```

We select quantum feature mapping with two feature dimensions.

Input:

```

# Quantum Feature Mapping with feature_dimension = 2 and reps = 2
from qiskit.circuit.library import PauliFeatureMap

qfm_default = PauliFeatureMap(feature_dimension=2,
                               paulis = ['ZI','IZ','ZZ'],
                               reps=2, entanglement='full')
print(qfm_default)

qfm_8 = PauliFeatureMap(feature_dimension=2,
                        paulis = ['ZI','IZ','ZZ'],
                        reps=2, entanglement='full', data_map_func=data_map_8)
print(qfm_8)

qfm_9 = PauliFeatureMap(feature_dimension=2,
                        paulis = ['ZI','IZ','ZZ'],
                        reps=2, entanglement='full', data_map_func=data_map_9)
print(qfm_9)

qfm_10 = PauliFeatureMap(feature_dimension=2,
                         paulis = ['ZI','IZ','ZZ'],
                         reps=2, entanglement='full', data_map_func=data_map_10)
print(qfm_10)

qfm_11 = PauliFeatureMap(feature_dimension=2,
                         paulis = ['ZI','IZ','ZZ'],
                         reps=2, entanglement='full', data_map_func=data_map_11)
print(qfm_11)

qfm_12 = PauliFeatureMap(feature_dimension=2,
                         paulis = ['ZI','IZ','ZZ'],
                         reps=2, entanglement='full', data_map_func=data_map_12)
print(qfm_12)

```

Output:

```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```



```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```



```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```



```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```



```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```



```

q_0: - 0  PauliFeatureMap(x[0], x[1])
q_1: - 1

```

The code makes use of the default implementation of the Sampler primitive and employs the ComputeUncompute fidelity measure to calculate the overlaps between states. In the event that specific instances of Sampler or Fidelity are not provided, the code will automatically generate these objects with the default values.

Input:

```

from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit_machine_learning.kernels import FidelityQuantumKernel
fidelity=ComputeUncompute(sampler=sampler)
Q_Kernel_8=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_8)
Q_Kernel_9=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_9)
Q_Kernel_10=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_10)
Q_Kernel_11=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_11)
Q_Kernel_12=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_12)
Q_Kernel_default=FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_default)

```

We process our data as we have done previously in classical computing (load data, process missing data, split data, normalize, use PCA):

Input:

```

from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Import dataset
data = '../data/datasets/neurons_binary.csv'
df = pd.read_csv(data, delimiter=';')

```

```

# Drop row having at least 1 missing value
df = df.dropna()

# Creating an instance of Labelencoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[["Target"]] = df[["Target"]].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=seed)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_train = pd.DataFrame(X_train, columns = X.columns)
X_test = Normalize.fit_transform(X_test)
X_test = pd.DataFrame(X_test, columns = X.columns)

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
# transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)
# Define a new DataFrame with two column (the principal components)
component_columns = []
for x in (n+1 for n in range(2)):
    component_columns = component_columns + ['PCA_%i'%x]
X_train = pd.DataFrame(data = X_train, columns = component_columns)
X_test = pd.DataFrame(data = X_test, columns = component_columns)

```

Then, we iterate the different classifiers.

Input:

```

from qiskit_machine_learning.algorithms import QSVC

names = ["Q_Kernel_default", "Q_Kernel_8", "Q_Kernel_9",
        "Q_Kernel_10", "Q_Kernel_11", "Q_Kernel_12"]

classifiers = [
    QSVC(quantum_kernel=Q_Kernel_default),
    QSVC(quantum_kernel=Q_Kernel_8),
    QSVC(quantum_kernel=Q_Kernel_9),
    QSVC(quantum_kernel=Q_Kernel_10),

```

```

QSVC(quantum_kernel=Q_Kernel_11),
QSVC(quantum_kernel=Q_Kernel_12),
]

# Iterate over classifiers and give classification test score
for name, clf in zip(names, classifiers):
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    print(f'Callable kernel classification test score for {name}: {score}')

```

Output:

```

Callable kernel classification test score for Q_Kernel_default: 0.53
Callable kernel classification test score for Q_Kernel_8: 0.46
Callable kernel classification test score for Q_Kernel_9: 0.38
Callable kernel classification test score for Q_Kernel_10: 0.67
Callable kernel classification test score for Q_Kernel_11: 0.44
Callable kernel classification test score for Q_Kernel_12: 0.62

```

As we have done previously in classical computing, we can provide metrics about our model (accuracy, precision, recall, f1 score, cross-validation, classification report).

Input:

```

from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report

# Provide metrics over classifiers
for name, clf in zip(names, classifiers):
    print("\n")
    print(name)
    print("\n")
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    print("\n")
    print("Print predicted data coming from X_test as new input data")
    print(y_pred)
    print("\n")
    print("Print real values\n")
    print(y_test)
    print("\n")

    print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
    print("Precision:", metrics.precision_score(y_test, y_pred, average='micro'))
    print("Recall:", metrics.recall_score(y_test, y_pred, average='micro'))
    print("f1 Score:", metrics.f1_score(y_test, y_pred, average='micro'))
    print("Cross Validation Mean:", cross_val_score(clf, X_train, y_train, cv=5).mean())
    print("Cross Validation Std:", cross_val_score(clf, X_train, y_train, cv=5).std())

    print('Classification Report: \n')
    print(classification_report(y_test,y_pred))

```

Output:

Q_Kernel_default

Print predicted data coming from X_test as new input data
[1 1 1 1 1 1 1 1]

Print real values

9 0
30 3
19 1
35 3
0 0
21 2
3 0
29 2
Name: Target, dtype: int64

Accuracy: 0.125
Precision: 0.125
Recall: 0.125
f1 Score: 0.125
Cross Validation Mean: 0.11904761904761904
Cross Validation Std: 0.10858813572372743

Q_Kernel_8

Print predicted data coming from X_test as new input data
[1 2 2 1 2 1 1 2]

Print real values

9 0
30 3
19 1
35 3
0 0
21 2
3 0
29 2
Name: Target, dtype: int64

Accuracy: 0.125
Precision: 0.125
Recall: 0.125
f1 Score: 0.125
Cross Validation Mean: 0.21904761904761902
Cross Validation Std: 0.07589227357385346

Q_Kernel_9

```
Print predicted data coming from X_test as new input data  
[1 1 1 1 1 1 1]
```

```
Print real values
```

```
9      0  
30     3  
19     1  
35     3  
0      0  
21     2  
3      0  
29     2  
Name: Target, dtype: int64
```

```
Accuracy: 0.125  
Precision: 0.125  
Recall: 0.125  
f1 Score: 0.125  
Cross Validation Mean: 0.2523809523809524  
Cross Validation Std: 0.08192690730516787
```

Q_Kernel_10

```
Print predicted data coming from X_test as new input data  
[1 1 1 1 1 1 1]
```

```
Print real values
```

```
9      0  
30     3  
19     1  
35     3  
0      0  
21     2  
3      0  
29     2  
Name: Target, dtype: int64
```

```
Accuracy: 0.125  
Precision: 0.125  
Recall: 0.125  
f1 Score: 0.125  
Cross Validation Mean: 0.1571428571428571  
Cross Validation Std: 0.09110060223670947
```

Q_Kernel_11

```
Print predicted data coming from X_test as new input data  
[1 2 1 2 1 2 1 1]
```

Print real values

```
9      0  
30     3  
19     1  
35     3  
0      0  
21     2  
3      0  
29     2  
Name: Target, dtype: int64
```

```
Accuracy: 0.25  
Precision: 0.25  
Recall: 0.25  
f1 Score: 0.25  
Cross Validation Mean: 0.28095238095238095  
Cross Validation Std: 0.1846735183777649
```

Q_Kernel_12

```
Print predicted data coming from X_test as new input data  
[1 3 1 1 3 3 1 1]
```

Print real values

```
9      0  
30     3  
19     1  
35     3  
0      0  
21     2  
3      0  
29     2  
Name: Target, dtype: int64
```

```
Accuracy: 0.25  
Precision: 0.25  
Recall: 0.25  
f1 Score: 0.25  
Cross Validation Mean: 0.22380952380952376  
Cross Validation Std: 0.08984743935292004  
Classification Report:
```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	3
1	0.20	1.00	0.33	1
2	0.00	0.00	0.00	2
3	0.33	0.50	0.40	2
accuracy			0.25	8
macro avg	0.13	0.38	0.18	8
weighted avg	0.11	0.25	0.14	8

Another coding example with `q_kernel_zz` is described below. We will use the ZZFeatureMap with linear entanglement, we will repeat the data encoding step two times, and we will use feature selection (embedded decision tree) to select five features (on a very small dataset of 260 neurons). We will use five qubits on the StatevectorSimulator from the IBM Quantum framework (<https://qiskit.org/>). The simulator models the noiseless execution of quantum computer hardware, representing the ideal. It evaluates the resulting quantum state vector. For each experiment, to assess whether patterns are identifiable, we will train the supervised classification algorithms using 80% of each sample, randomly chosen, and we will assess the accuracy in predicting the remaining 20%. The accuracy of the classification will be assessed by performing cross-validation on the training dataset. We will average the fivefold cross-validation scores resulting in a mean \pm standard deviation score. For data rescaling, we will use QuantileTransformer. We will also transform the dataset using a Mahalanobis transformation with suppression of a neuron when the surface of the soma equals 0. The Mahalanobis distance is a multivariate metric measuring the distance between a point and a distribution. Applying the Mahalanobis distance allows reduction of the standard deviation for each feature by deleting neurons from the dataset. The datasets will be preprocessed to address missing values. If a value within the features is missing, the neuron will be deleted from the dataset. Categorical features such as morphology types will be encoded, transforming each categorical feature with m possible values.

Input:

```
# Import utilities
import numpy as np
import pandas as pd

# seed for randomization, to keep outputs consistent
from qiskit.utils import algorithm_globals
seed = 123456
algorithm_globals.random_seed = seed

# Define parameters
cv = 5 # 5-fold cross-validation
feature_dimension = 5 # Features dimension
k_features = 5 # Feature selection
reps = 2 # Repetition
ibm_account = 'YOUR TOKEN'
quantum_backend = 'simulator_statevector'

# Import dataset
data = '../data/datasets/neurons_maha_soma.csv'
neuron = pd.read_csv(data, delimiter=',')

print(neuron)

df = neuron.head(22).copy() # Ganglion
df = pd.concat([df, neuron.iloc[320:340]]) # Granule
```

```

df = pd.concat([df, neuron.iloc[1493:1513]])    # Medium Spiny
df = pd.concat([df, neuron.iloc[1171:1191]])    # Parachromaffin
df = pd.concat([df, neuron.iloc[10031:10051]])   # Pyramidal

df = pd.concat([df, neuron.iloc[2705:2725]])    # Basket
df = pd.concat([df, neuron.iloc[22589:22609]])  # Bitufted
df = pd.concat([df, neuron.iloc[3175:3195]])    # Chandelier
df = pd.concat([df, neuron.iloc[22644:22664]])  # Double bouquet
df = pd.concat([df, neuron.iloc[3199:3219]])    # Martinotti
df = pd.concat([df, neuron.iloc[8260:8280]])    # Nitrergic

df = pd.concat([df, neuron.iloc[2255:2275]])    # Astrocytes
df = pd.concat([df, neuron.iloc[3306:3326]])    # Microglia

# Drop row having at least 1 missing value
df = df.dropna()

# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[["Target"]] = df[["Target"]].apply(enc.fit_transform)

from sklearn.model_selection import train_test_split
# We split our data to y (Target) and X (features)
y = df.loc[:, df.columns == 'Target']
# Features variables
X = df.loc[:, df.columns != ('Target')]
# Split data into train and test
# Option test_size = 0.2 means that we take 20% of the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scaling the data
from sklearn.preprocessing import QuantileTransformer
Normalize = QuantileTransformer(n_quantiles=1000, output_distribution="uniform")
# Transform data
X_train = Normalize.fit_transform(X_train)
X_train = pd.DataFrame(X_train, columns = X.columns)
X_test = Normalize.fit_transform(X_test)
X_test = pd.DataFrame(X_test, columns = X.columns)

from sklearn.tree import DecisionTreeClassifier

def embedded_decision_tree_classifier(X, y, k_features, output_folder=None):
    """
    Here we use decision tree classifier to select features. We select the k best
    features (k_features)
    """

```

```

Inputs:
- X (features) DataFrame
- y (target) DataFrame
"""

print("\n")
print("Decision Tree Regressor Features Importance: started")
print("\n")

# define the model
model = DecisionTreeClassifier()
# fit the model
model.fit(X, y)
# get importance
importance = model.feature_importances_
# Get features name
feature_names = [f"{i}" for i in X.columns]

# create a data frame to visualize features importance
features_importance = pd.DataFrame({"Features": feature_names, "Importances": importance})
features_importance.set_index('Importances')

# Print features importance
print("\n")
print("Features Importances:")
print("\n")
print(features_importance)
if output_folder is not None:
    features_importance.to_csv(output_folder+'Decision_Tree_Classifier_Features_Importance.csv', index=False)

if output_folder is not None:
    # plot feature importance
    features_importance.plot(kind='bar', x='Features', y='Importances')
    pyplot.title('Decision Tree Classifier Features Importance')
    pyplot.tight_layout()
    pyplot.savefig(output_folder+'Decision_Tree_Classifier_Features_Importance.png')

# Select the k most important features
features_columns = []
# Order the features importance dataframe
df = pd.DataFrame(data = features_importance.sort_values(by='Importances', key=abs, ascending=False))
# Put the k most important features in features_columns
for x in range(k_features):
    features_columns = features_columns + [df.iloc[x][0]]

# Create a new DataFrame with selected features
df_data = pd.DataFrame(data = X, columns = features_columns)

```

```

print("\n")
print("Decision Tree Classifier Features Importance: DataFrame")
print("\n")
print(df_data)

return df_data

X_train = embedded_decision_tree_classifier(X_train, y_train, k_features)
X_test = pd.DataFrame(data = X_test, columns = X_train.columns)

feature_dimension = X_train.shape[1] # Number of features
multiclass = None
output_folder = None

# We convert pandas DataFrame into numpy array
X_train = X_train.to_numpy()
X_test = X_test.to_numpy()

y_train = y_train.values.ravel()
y_test = y_test.values.ravel()

# seed for randomization, to keep outputs consistent
seed = 123456
algorithm_globals.random_seed = seed

# Quantum Feature Mapping with feature_dimension = 5 and reps = 2
from qiskit.circuit.library import ZZFeatureMap
qfm_zz = ZZFeatureMap(feature_dimension=feature_dimension, reps=reps,
entanglement="linear")

print(qfm_zz)

if quantum_backend is not None:
    # Compute code with online quantum simulators or quantum hardware from the cloud
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
    # Define service
    service = QiskitRuntimeService(channel = 'ibm_quantum', token = ibm_account,
instance = 'ibm-q/open/main')
    # Get backend
    backend = service.backend(quantum_backend) # Use a simulator or hardware from the cloud
    # Define Sampler with different options
    # resilience_level=1 adds readout error mitigation
    # execution.shots is the number of shots
    # optimization_level=3 adds dynamical decoupling
    from qiskit_ibm_runtime import Options
    options = Options()
    options.resilience_level = 1
    options.execution.shots = 1024
    options.optimization_level = 3
    sampler = Sampler(session=backend, options = options)
else:

```

```

# Compute code with local simulator (Aer simulators)
from qiskit.primitives import Sampler
sampler = Sampler()

# After preparing our training and testing datasets, we configure the
FidelityQuantumKernel class to compute a kernel matrix using the ZZFeatureMap.
# We utilize the default implementation of the Sampler primitive and the
ComputeUncompute fidelity, which calculates the overlaps between states.
# If you do not provide specific instances of Sampler or Fidelity, the code will
automatically create these objects with the default values.
from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit_machine_learning.kernels import FidelityQuantumKernel
fidelity = ComputeUncompute(sampler=sampler)
Q_Kernel_zz = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_zz)

# QSVC model
model = QSVC(quantum_kernel=Q_Kernel_zz)
model.fit(X_train,y_train)
score = model.score(X_test, y_test)
print(f'Callable kernel classification test score for q_kernel_zz: {score}')

y_pred = model.predict(X_test)

print("\n")
print("Print predicted data coming from X_test as new input data")
print(y_pred)
print("\n")
print("Print real values\n")
print(y_test)
print("\n")

# K-Fold Cross Validation
from sklearn.model_selection import KFold
k_fold = KFold(n_splits=cv)
score = np.zeros(cv)
i = 0
print(score)
for indices_train, indices_test in k_fold.split(X_train):
    #print(indices_train, indices_test)
    X_train_ = X_train[indices_train]
    X_test_ = X_train[indices_test]
    y_train_ = y_train[indices_train]
    y_test_ = y_train[indices_test]

    # fit classifier to data
    model.fit(X_train_, y_train_)

    # score classifier
    score[i] = model.score(X_test_, y_test_)

```

```

    i = i + 1
import math
print("cross validation scores: ", score)
cross_mean = sum(score) / len(score)
cross_var = sum(pow(x - cross_mean,2) for x in score) / len(score) # variance
cross_std = math.sqrt(cross_var) # standard deviation
print("cross validation mean: ", cross_mean)

results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score(y_test,
y_pred, average='micro'), metrics.recall_score(y_test, y_pred, average='micro'),
metrics.f1_score(y_test, y_pred, average='micro'), cross_mean, cross_std]

metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
['q_kernel_zz'])

print('Classification Report: \n')
print(classification_report(y_test,y_pred))

print(metrics_dataframe)

```

Output:

Target	Soma_Surface	N_stems	N_bifs	N_branch	N_tips	\
0	ganglion	1149.320	4.0	101.0	206.0	106.0
1	ganglion	1511.830	3.0	70.0	143.0	74.0
2	ganglion	1831.530	3.0	13.0	29.0	17.0
3	ganglion	1291.270	6.0	109.0	224.0	116.0
4	ganglion	3064.340	4.0	60.0	124.0	65.0
...
22686	double_bouquet	605.067	5.0	132.0	269.0	138.0
22687	double_bouquet	920.949	6.0	121.0	248.0	128.0
22688	double_bouquet	770.529	3.0	104.0	211.0	108.0
22689	double_bouquet	478.078	4.0	158.0	320.0	163.0
22690	double_bouquet	629.470	4.0	65.0	134.0	70.0
	Width	Height	Depth	Type	...	Bif_ampl_remote Bif_tilt_local \
0	249.09	493.80	33.63	1806.0	...	9132.460 9543.61
1	453.80	390.94	35.00	1504.0	...	6034.870 6750.98
2	282.23	324.06	29.24	699.0	...	835.754 1215.33
3	228.86	616.17	42.19	2131.0	...	9696.640 10160.10
4	264.09	364.24	44.37	1568.0	...	5084.620 5927.53
...
22686	222.40	1212.65	120.80	6247.0	...	11348.400 13220.00
22687	328.02	980.65	227.04	32561.0	...	9996.370 12886.10
22688	247.39	1322.26	96.81	5746.0	...	7972.950 10343.50
22689	343.21	813.14	115.01	9878.0	...	11555.700 15740.30
22690	305.82	1164.29	183.69	15343.0	...	5267.240 6223.78

```

      Bif_tilt_remote  Bif_torque_local  Bif_torque_remote  Last_parent_diam \
0           9714.80          7413.60          7348.63          11.24
1            6781.43          6896.19          7620.16           7.75
2            1478.25          1192.02          1001.38           4.06
3           10281.70          9992.69          10400.00           9.38
4            5852.84          5128.55          5371.86           7.73
...
22686         12887.70          12790.80          12031.90          11.41
22687         12170.90          10669.40          10279.20          28.52
22688         10984.10          9389.51          10205.70          10.84
22689         17768.70          14536.90          13063.80          17.55
22690         6298.83          5738.15          6259.83           6.12

      Diam_threshold  HillmanThreshold  Helix  Fractal_Dim
0           114.0480          123.3370       -0.33        56.4171
1           101.5980          103.2830       -0.01        46.9175
2            59.7340          72.2750       -0.64        20.7496
3           125.3340          145.4360       -5.34        49.3177
4           123.6620          140.4000      -1.27        51.0624
...
22686         73.7220          78.2270      -1.86        205.7210
22687         63.8622          95.7028       1.93        239.6990
22688         71.3840          79.0890      -2.71        142.2360
22689         73.8240          81.1340      -3.20        271.7430
22690         44.6384          53.5180     -11.67        124.9970

```

[22691 rows x 44 columns]

Decision Tree Regressor Features Importance: started

Features Importances:

	Features	Importances
0	Soma_Surface	0.132398
1	N_stems	0.019728
2	N_bifs	0.000000
3	N_branch	0.000000
4	N_tips	0.000000
5	Width	0.000000
6	Height	0.174589
7	Depth	0.070581
8	Type	0.115796
9	Diameter	0.000000
10	Diameter_pow	0.000000
11	Length	0.000000
12	Surface	0.000000
13	SectionArea	0.022928
14	Volume	0.000000
15	EucDistance	0.000000
16	PathDistance	0.095161
17	Branch_Order	0.000000
18	Terminal_degree	0.000000
19	TerminalSegment	0.009220
20	Taper_1	0.076445

```

21             Taper_2      0.000000
22     Branch_pathlength  0.044744
23             Contraction 0.000000
24             Fragmentation 0.000000
25             Daughter_Ratio 0.024061
26 Parent_Daughter_Ratio 0.000000
27 Partition_asymmetry 0.000000
28             Rall_Power  0.014306
29                 Pk       0.008298
30             Pk_classic 0.000000
31                 Pk_2    0.000000
32     Bif_ampl_local   0.066446
33     Bif_ampl_remote  0.027766
34     Bif_tilt_local   0.000000
35     Bif_tilt_remote  0.000000
36     Bif_torque_local 0.000000
37     Bif_torque_remote 0.000000
38     Last_parent_diam 0.016321
39     Diam_threshold   0.020529
40 HillmanThreshold    0.045931
41             Helix     0.014752
42     Fractal_Dim     0.000000

```

Decision Tree Classifier Features Importance: DataFrame

	Height	Soma_Surface	Type	PathDistance	Taper_1
0	0.980769	0.649038	0.701923	0.774038	0.725962
1	0.697115	0.711538	0.754808	0.730769	0.850962
2	0.533654	0.216346	0.348558	0.461538	0.423077
3	0.105769	0.245192	0.456731	0.187500	0.932692
4	0.235577	0.485577	0.500000	0.495192	0.375000
..
204	0.658654	0.913462	0.326923	0.504808	0.673077
205	0.947115	0.596154	0.899038	0.889423	0.923077
206	0.278846	0.947115	0.009615	0.062500	0.014423
207	0.480769	0.658654	0.956731	0.918269	0.894231
208	0.817308	0.437500	0.913462	0.879808	0.870192

[209 rows x 5 columns]

```

q_0: 0
|
|
q_1: 1
|
|
q_2: ZZFeatureMap(x[0],x[1], x[2], x[3], x[4])
|
|
q_3: 3
|
|
q_4: 4

```

```

ibmq_qasm_simulator
ibmq_lima
ibmq_belem
ibmq_quito
simulator_statevector
simulator_mps
simulator_extended_stabilizer
simulator_stabilizer
ibmq_manila
ibm_nairobi
ibm_oslo
Callable kernel classification test score for q_kernel_zz: 0.4716981132075472

Print predicted data coming from X_test as new input data
[ 9 10 7 4 11 8 1 9 12 3 12 0 6 10 3 9 11 5 2 11 12 12 3 10
 7 7 5 11 7 8 6 3 0 7 6 1 11 9 10 5 0 8 3 11 12 3 9 3
 1 12 0 4 1]

Print real values

[ 9 10 7 4 11 2 4 9 5 1 6 0 8 10 7 9 11 5 10 11 2 3 5 10
 1 3 11 11 12 5 6 7 0 1 6 7 11 0 10 5 9 8 1 11 6 3 0 5
 12 2 9 4 1]

[0. 0. 0. 0. 0.]

cross validation scores: [0.38095238 0.5           0.57142857 0.42857143 0.58536585]
cross validation mean: 0.4932636469221835
Classification Report:

      precision    recall   f1-score   support

          0         0.50     0.50     0.50        4
          1         0.25     0.20     0.22        5
          2         0.00     0.00     0.00        3
          3         0.14     0.33     0.20        3
          4         1.00     0.67     0.80        3
          5         0.67     0.33     0.44        6
          6         0.67     0.50     0.57        4
          7         0.20     0.25     0.22        4
          8         0.33     0.50     0.40        2
          9         0.60     0.60     0.60        5
         10         1.00     0.80     0.89        5
         11         1.00     0.86     0.92        7
         12         0.00     0.00     0.00        2

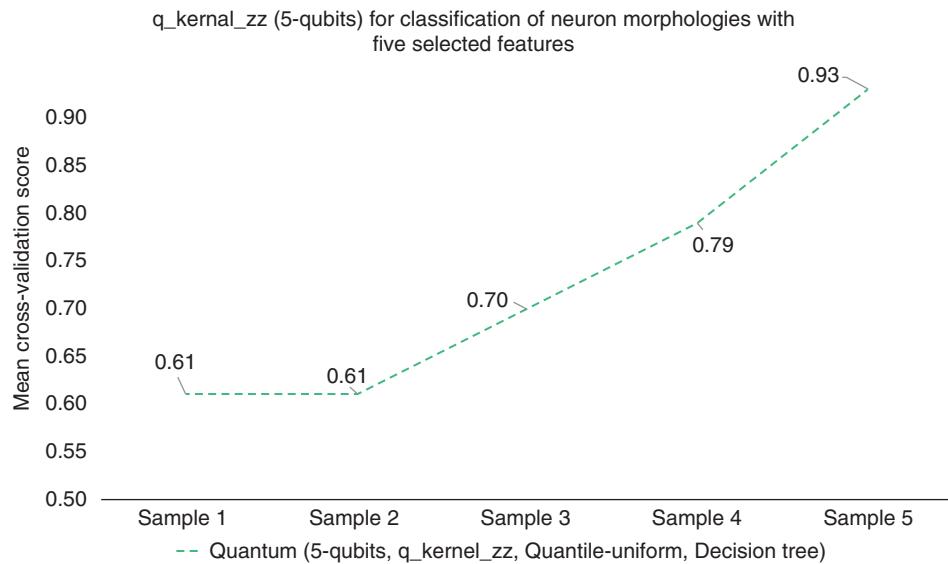
    accuracy                           0.47          53
   macro avg       0.49       0.43       0.44          53
weighted avg       0.56       0.47       0.50          53

q_kernel_zz
Accuracy           0.471698
Precision          0.471698
Recall             0.471698
F1 Score           0.471698
Cross-validation mean 0.493264
Cross-validation std 0.079293

```

As we can see, the results improved. If we now apply the same algorithm by varying the size of the dataset, the cross-validation score will improve (Table 5.2).

When applying the q_kernel_zz on the entire dataset, we have the following results:



As we can see, the classification accuracy has improved significantly.

Table 5.2 Dataset with the number of neuron morphologies for multiclass classification. From the 27,881 extracted neurons, 22,691 neurons (Sample 5) remained after the application of Mahalanobis distance transformation and suppression of all neurons with a soma surface equal to 0.

	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
<i>Principal cells</i>					
Ganglion	20	50	100	200	318
Granule	20	50	100	200	851
Medium Spiny	20	50	100	200	762
Parachromaffin	20	50	100	200	322
Pyramidal	20	50	100	200	12,558
<i>Interneurons</i>					
Basket	20	50	100	200	470
Bitufted	20	50	67	67	55
Chandelier	20	26	26	26	24
Double bouquet	20	50	50	50	49
Martinotti	20	50	100	137	107
Nitrogenic	20	50	100	200	1771
<i>Glial cells</i>					
Astrocytes	20	50	100	200	450
Microglia	20	50	100	200	4954
Total	260	626	1143	2080	22691

5.3 Quantum Kernel Training

It is also possible to train a quantum kernel with quantum kernel alignment (QKA), which iteratively adapts a parameterized quantum kernel to a dataset and converges to the maximum SVM margin at the same time. To implement it, we prepare the dataset as usual and define the quantum feature map. Then, we use the `QuantumKernelTrained.fit` method to train the kernel parameters and pass them to a machine learning model.

Input:

```
# Import utilities
import numpy as np
import pandas as pd

# Import dataset
data = '../data/datasets/neurons.csv'
neuron = pd.read_csv(data, delimiter=';')

# Select a subset of the data composed of three classes
df = neuron.head(20).copy() # Ganglion
df = pd.concat([df, neuron.iloc[373:393]]) # Granule
df = pd.concat([df, neuron.iloc[17033:17053]]) # Basket

# seed for randomization, to keep outputs consistent
from qiskit.utils import algorithm_globals
seed = 123456
algorithm_globals.random_seed = seed

# Drop row having at least 1 missing value
df = df.dropna()

# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[['Target']] = df[['Target']].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

# Scaling the data
from sklearn import preprocessing
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_train = pd.DataFrame(X_train, columns = X.columns)
X_test = Normalize.fit_transform(X_test)
X_test = pd.DataFrame(X_test, columns = X.columns)
```

```

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
# transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)

# Define a new DataFrame with two column (the principal components)
component_columns = []
for x in (n+1 for n in range(2)):
    component_columns = component_columns + ['PCA_%i'%x]
X_train = pd.DataFrame(data = X_train, columns = component_columns)
X_test = pd.DataFrame(data = X_test, columns = component_columns)

# Define some parameters
feature_dimension = 2 # number of features
reps = 2 # number of repetitions
quantum_backend = 'ibmq_qasm_simulator' # quantum backend
cv = 2 # Cross-validation

# Define a callback class for our optimizer
class QKTCallback:
    """Callback wrapper class."""

    def __init__(self) -> None:
        self._data = [[] for i in range(5)]

    def callback(self, x0, x1=None, x2=None, x3=None, x4=None):
        """
        Args:
            x0: number of function evaluations
            x1: the parameters
            x2: the function value
            x3: the stepsize
            x4: whether the step was accepted
        """
        self._data[0].append(x0)
        self._data[1].append(x1)
        self._data[2].append(x2)
        self._data[3].append(x3)
        self._data[4].append(x4)

    def get_callback_data(self):
        return self._data

    def clear_callback_data(self):
        self._data = [[] for i in range(5)]

# Qiskit imports
from qiskit import QuantumCircuit
from qiskit.circuit import ParameterVector

```

```

from qiskit.visualization import circuit_drawer
from qiskit.circuit.library import ZZFeatureMap
from qiskit_machine_learning.kernels import TrainableFidelityQuantumKernel
from qiskit_machine_learning.kernels.algorithms import QuantumKernelTrainer

# Create a rotational layer to train. We will rotate each qubit the same amount.
training_params = ParameterVector("θ", 1)
fm0 = QuantumCircuit(feature_dimension)
for qubit in range(feature_dimension):
    fm0.ry(training_params[0], qubit)

# Use ZZFeatureMap to represent input data
fm1=ZZFeatureMap(feature_dimension=feature_dimension, reps=reps, entanglement='linear')

# Create the feature map, composed of our two circuits
fm = fm0.compose(fm1)

print(circuit_drawer(fm))
print(f"Trainable parameters: {training_params}")

if quantum_backend is not None:
    # Compute code with online quantum simulators or quantum hardware from the cloud
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
    # Define service
    service = QiskitRuntimeService(channel = 'ibm_quantum', token = "YOUR TOKEN",
instance = 'ibm-q/open/main')
    # Get backend
    backend = service.backend(quantum_backend) # Use a simulator or hardware from the cloud
    # Define Sampler: With our training and testing datasets ready, we set up the
FidelityQuantumKernel class to calculate a kernel matrix using the ZZFeatureMap. We
use the reference implementation of the Sampler primitive and the ComputeUncompute
fidelity that computes overlaps between states. These are the default values and if
you don't pass a Sampler or Fidelity instance, the same objects will be created
automatically for you.
    # Run Quasi-Probability calculation
    # optimization_level=3 adds dynamical decoupling
    # resilience_level=1 adds readout error mitigation
    from qiskit_ibm_runtime import Options
    options = Options()
    options.resilience_level = 1
    options.execution.shots = 1024
    options.optimization_level = 3
    sampler = Sampler(session=backend, options = options)
else:
    # Compute code with local simulator (Aer simulators)
    from qiskit.primitives import Sampler
    sampler = Sampler()

```

```
# We utilize the default implementation of the Sampler primitive and the
ComputeUncompute fidelity, which calculates the overlaps between states.
# If you do not provide specific instances of Sampler or Fidelity, the code will
automatically create these objects with the default values.
from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit_machine_learning.kernels import FidelityQuantumKernel
fidelity = ComputeUncompute(sampler=sampler)

# Instantiate quantum kernel
quant_kernel = TrainableFidelityQuantumKernel(fidelity = fidelity, feature_map=fm,
training_parameters=training_params)

# Set up the optimizer
cb_qkt = QKTCallback()
spsa_opt = SPSA(maxiter=10, callback=cb_qkt.callback, learning_rate=0.05,
perturbation=0.05)

# Instantiate a quantum kernel trainer
qkt = QuantumKernelTrainer(
    quantum_kernel=quant_kernel, loss="svc_loss", optimizer=spsa_opt, initial_point=
    [np.pi / 2]
)

# Train the kernel using QKT directly
qka_results = qkt.fit(X_train, y_train)
optimized_kernel = qka_results.quantum_kernel
print(qka_results)

# Use QSVC for classification
qsvc = QSVC(quantum_kernel=optimized_kernel)

# Fit the QSVC
qsvc.fit(X_train, y_train)

# Predict the labels
y_pred = qsvc.predict(X_test)

# Evaluate the test accuracy
accuracy_test = metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred)
print(f"accuracy test: {accuracy_test}")

# Print predicted values and real values of the X_test dataset
print("\n")
print("Print predicted data coming from X_test as new input data")
print(y_pred)
print("\n")
print("Print real values\n")
print(y_test)
print("\n")
```

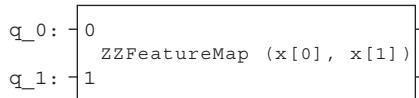
```
# Print accuracy metrics of the model
results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score(y_test,
y_pred, average='micro'), metrics.recall_score(y_test, y_pred, average='micro'),
metrics.f1_score(y_test, y_pred, average='micro'), cross_val_score(qsvc, X_train,
y_train, cv=cv).mean(), cross_val_score(qsvc, X_train, y_train, cv=cv).std()]
metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision", "Recall",
"F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
['q_kernel_training'])
print('Classification Report: \n')
print(classification_report(y_test,y_pred))

metrics_dataframe
```

Output:



Trainable parameters: θ , [' $\theta[0]$ ']



```
{
    'optimal_parameters': {ParameterVectorElement(theta[0]): 1.3059540613374132},
    'optimal_point': array([1.30595406]),
    'optimal_value': 18.27409165606242,
    'optimizer_evals': 30,
    'optimizer_time': None,
    'quantum_kernel': <qiskit_machine_learning.kernels.quantum_kernel.QuantumKernel
object at 0x12e514f40>}
accuracy test: 0.5595238095238095
```

Print predicted data coming from X_{test} as new input data
[2 2 2 1 0 0 1 1 0 1 1 2]

Print real values

377	2
17034	0
381	2
17051	0
17052	0
374	2
9	1
17041	0

```

388      2
378      2
382      2
384      2
Name: Target, dtype: int64

```

Classification Report:

	precision	recall	f1-score	support
0	0.33	0.25	0.29	4
1	0.20	1.00	0.33	1
2	0.75	0.43	0.55	7
accuracy			0.42	12
macro avg	0.43	0.56	0.39	12
weighted avg	0.57	0.42	0.44	12

5.4 Pegasos QSVC: Binary Classification

There is an alternative method to QSVC (which uses the dual optimization from scikit-learn), namely the Pegasos algorithm from Shalev-Shwartz in which another SVM-based algorithm benefits from the quantum kernel method. PegasosQSVC yields a training complexity that is independent of the size of the training set, meaning that this method can train faster than QSVC with large training sets. We need to optimize some hyperparameters.

Input:

```

# Import utilities
import numpy as np
import pandas as pd

# seed for randomization, to keep outputs consistent
from qiskit.utils import algorithm_globals
seed = 123456
algorithm_globals.random_seed = seed

# number of steps performed during the training procedure
tau = 100

# regularization parameter
C = 1000

# Encoding Functions
from functools import reduce

def data_map_8(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*(m * n), x)
    return coeff

```

```

def data_map_9(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: (np.pi/2)*(m * n), 1 - x)
    return coeff

def data_map_10(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*np.exp(((n - m)*(n - m))/8), x)
    return coeff

def data_map_11(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: (np.pi/3)*(m * n), 1/(np.cos(x)))
    return coeff

def data_map_12(x: np.ndarray) -> float:
    coeff = x[0] if len(x) == 1 else reduce(lambda m, n: np.pi*(m * n), np.cos(x))
    return coeff

# Quantum Feature Mapping with feature_dimension = 2 and reps = 2
from qiskit.circuit.library import PauliFeatureMap

qfm_default = PauliFeatureMap(feature_dimension=2,
                                paulis = ['ZI','IZ','ZZ'],
                                reps=2, entanglement='full')
print(qfm_default)
qfm_8 = PauliFeatureMap(feature_dimension=2,
                        paulis = ['ZI','IZ','ZZ'],
                        reps=2, entanglement='full', data_map_func=data_map_8)
print(qfm_8)
qfm_9 = PauliFeatureMap(feature_dimension=2,
                        paulis = ['ZI','IZ','ZZ'],
                        reps=2, entanglement='full', data_map_func=data_map_9)
print(qfm_9)
qfm_10 = PauliFeatureMap(feature_dimension=2,
                          paulis = ['ZI','IZ','ZZ'],
                          reps=2, entanglement='full', data_map_func=data_map_10)
print(qfm_10)
qfm_11 = PauliFeatureMap(feature_dimension=2,
                          paulis = ['ZI','IZ','ZZ'],
                          reps=2, entanglement='full', data_map_func=data_map_11)
print(qfm_11)
qfm_12 = PauliFeatureMap(feature_dimension=2,
                          paulis = ['ZI','IZ','ZZ'],
                          reps=2, entanglement='full', data_map_func=data_map_12)
print(qfm_12)

print(qfm_8.draw())

# For quantum access, the following lines must be adapted
if quantum_backend is not None:
    # Compute code with online quantum simulators or quantum hardware from the cloud
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
    # Define service

```

```

        service = QiskitRuntimeService(channel = 'ibm_quantum', token = ibm_account,
instance = 'ibm-q/open/main')
        # Get backend
        backend = service.backend(quantum_backend) # Use a simulator or hardware from the cloud
        # Define Sampler
        #We use the reference implementation of the Sampler primitive and the
ComputeUncompute fidelity that computes overlaps between states. These are the
default values and if you don't pass a Sampler or Fidelity instance, the same objects
will be created automatically for you.
        # Run Quasi-Probability calculation
        # optimization_level=3 adds dynamical decoupling
        # resilience_level=1 adds readout error mitigation
        from qiskit_ibm_runtime import Options
        options = Options()
        options.resilience_level = 1
        options.execution.shots = 1024
        options.optimization_level = 3
        sampler = Sampler(session=backend, options = options)
else:
    # Compute code with local simulator (Aer simulators)
    from qiskit.primitives import Sampler
    sampler = Sampler()

from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit_machine_learning.kernels import FidelityQuantumKernel
fidelity = ComputeUncompute(sampler=sampler)
Q_Kernel_default = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_default)
Q_Kernel_8 = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_8)
Q_Kernel_9 = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_9)
Q_Kernel_10 = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_10)
Q_Kernel_11 = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_11)
Q_Kernel_12 = FidelityQuantumKernel(fidelity=fidelity, feature_map=qfm_12)

names = ["Q_Kernel_default", "Q_Kernel_8", "Q_Kernel_9",
         "Q_Kernel_10", "Q_Kernel_11", "Q_Kernel_12"]

classifiers = [
    PegasosQSVC(quantum_kernel=Q_Kernel_default, C=C, num_steps=tau),
    PegasosQSVC(quantum_kernel=Q_Kernel_8, C=C, num_steps=tau),
    PegasosQSVC(quantum_kernel=Q_Kernel_9, C=C, num_steps=tau),
    PegasosQSVC(quantum_kernel=Q_Kernel_10, C=C, num_steps=tau),
    PegasosQSVC(quantum_kernel=Q_Kernel_11, C=C, num_steps=tau),
    PegasosQSVC(quantum_kernel=Q_Kernel_12, C=C, num_steps=tau),
]

# Import dataset
data = '../data/datasets/neurons_test.csv'
df = pd.read_csv(data, delimiter=';')

# Drop row having at least 1 missing value
df = df.dropna()

```

```

# Creating an instance of Labelencoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[["Target"]] = df[["Target"]].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]]
y = df['Target']

# Splitting the data : training and test (20%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

# Scaling the data
Normalize = preprocessing.StandardScaler()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_train = pd.DataFrame(X_train, columns = X.columns)
X_test = Normalize.fit_transform(X_test)
X_test = pd.DataFrame(X_test, columns = X.columns)

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
# transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)
# Define a new DataFrame with two column (the principal components)
component_columns = []
for x in (n+1 for n in range(2)):
    component_columns = component_columns + ['PCA_%i'%x]
X_train = pd.DataFrame(data = X_train, columns = component_columns)
X_test = pd.DataFrame(data = X_test, columns = component_columns)

print(X_train)
print(X_test)

# iterate over classifiers
for name, clf in zip(names, classifiers):
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    print(f'Callable kernel classification test score for {name}: {score}')

from sklearn import metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report

# Provide metrics over classifiers
for name, clf in zip(names, classifiers):
    print("\n")
    print(name)

```

```

print("\n")
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("\n")
print("Print predicted data coming from X_test as new input data")
print(y_pred)
print("\n")
print("Print real values\n")
print(y_test)
print("\n")

print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
print("Precision:", metrics.precision_score(y_test, y_pred, average='micro'))
print("Recall:", metrics.recall_score(y_test, y_pred, average='micro'))
print("f1 Score:", metrics.f1_score(y_test, y_pred, average='micro'))
print("Cross Validation Mean:", cross_val_score(clf, X_train, y_train, cv=5).
mean())
print("Cross Validation Std:", cross_val_score(clf, X_train, y_train, cv=5).
std())

results = [metrics.accuracy_score(y_test, y_pred), metrics.precision_score
(y_test, y_pred, average='micro'), metrics.recall_score(y_test, y_pred,
average='micro'), metrics.f1_score(y_test, y_pred, average='micro'), cross_val_score
(clf, X_train, y_train, cv=5).mean(), cross_val_score(clf, X_train, y_train, cv=5).
std()]

metrics_dataframe = pd.DataFrame(results, index=["Accuracy", "Precision",
"Recall", "F1 Score", "Cross-validation mean", "Cross-validation std"], columns=
[name])

print('Classification Report: \n')
print(classification_report(y_test,y_pred))

print('Metrics:')
print(metrics_dataframe)

```

5.5 Quantum Neural Networks

Within the realm of quantum computing, machine learning models such as QNNs have emerged as a subclass of variational quantum algorithms. Feature maps, which find applications in QNN architectures, aim to harness quantum principles such as superposition, entanglement, and interference to enhance accuracy, expedite training, and accelerate model processing. While the debate between classical and quantum methods continues, some research papers have demonstrated theoretical advantages of future quantum systems.

In the context of Qiskit-Machine Learning, the `NeuralNetworkClassifier` and `NeuralNetworkRegressor` modules are employed. These modules accept a (Quantum) `NeuralNetwork` as input and utilize it within specific contexts. For convenience, pre-configured variants, namely the `QVC` (`VQC`) and `Variational Quantum Regressor` (`VQR`), are also available.

For classification tasks, the following options are available:

- Classification with an EstimatorQNN
- Classification with a SamplerQNN
- VQC

For regression tasks, the following options are available:

- Regression with an EstimatorQNN
- VQR

In the case of an EstimatorQNN used for classification within a NeuralNetworkClassifier, the EstimatorQNN is expected to produce one-dimensional output within the range of $[-1, +1]$. This setup is suitable for binary classification, where the two classes are assigned the values of -1 and $+1$, respectively.

Alternatively, a SamplerQNN employed for classification within a NeuralNetworkClassifier expects a d -dimensional probability vector as output, where d represents the number of classes. The underlying Sampler primitive generates quasi-distributions of bit strings, and a mapping from the measured bit strings to the different classes needs to be defined. For binary classification, a parity mapping is utilized.

The VQC serves as a specialized variant of the NeuralNetworkClassifier, employing a SamplerQNN. It applies a parity mapping (or extensions for multiple classes) to convert the bit string to the corresponding classification, resulting in a probability vector interpreted as a one-hot encoded result. By default, the VQC utilizes the CrossEntropyLoss function, which expects labels in a one-hot encoded format and returns predictions in the same format.

5.5.1 Binary Classification with EstimatorQNN

Let us start with a binary classification using EstimatorQNN within a NeuralNetworkClassifier. In the example below, we will import a .csv file with the dataset we want to classify (neuron morphologies to be classified as interneurons or principal neurons) and preprocess the data as usual:

- Drop rows having at least one missing value.
- Encode the labels to be $\{-1, +1\}$.
- Divide the data, with y the variable to predict (target) and X the features.
- Split the data into training (X_{train} , y_{train}) and test (30%).
- Normalize the data.
- Perform dimension reduction with PCA (with two principal components).

We then create a quantum instance and build our QNN, create the neural network classifier, fit the classifier to our data, and give the score and prediction using our testing dataset.

In addition, we create a callback function that will be called for each iteration of the optimizer. It needs two parameters, which are the current weights and the value of the objective function at those weights.

Input:

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

# Import dataset
data = '../data/datasets/neurons_binary.csv'
neuron = pd.read_csv(data, delimiter=';')

# Drop row having at least 1 missing value
neuron = neuron.dropna()
```

```

# We select some data from the dataset to compute faster
df = neuron.head(22).copy() # Principal neurons
df = pd.concat([df, neuron.iloc[17034:17054]]) # Interneurons

# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[["Target"]] = df[["Target"]].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]].to_numpy() # We remove labels and convert pandas DataFrame
into numpy array
y = df['Target'].replace(0, -1).to_numpy() # We replace our labels by 1 and -1 and
convert pandas DataFrame into numpy array

# Splitting the data : training and test (30%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Scaling the data
from sklearn import preprocessing
from sklearn.preprocessing import Normalizer
Normalize = preprocessing.Normalizer()

# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)

# Transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)

# Variable definition
feature_dimension = 2 # Number of qubits
quantum_backend = None # We use local simulator
reps = 2 # Number of repetitions

if quantum_backend is not None:
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Estimator
    # Define service
    service = QiskitRuntimeService(channel = 'ibm_quantum', token = ibm_account,
instance = 'ibm-q/open/main')

```

```

# Get backend
backend = service.backend(quantum_backend) # Use a simulator or hardware from the
cloud
# We use the reference implementation of the Estimator primitive.
from qiskit_ibm_runtime import Options
options = Options()
options.resilience_level = 1
options.execution.shots = 1024
options.optimization_level = 3
estimator = Estimator(session=backend, options = options)
else:
    from qiskit.primitives import Estimator
    estimator = Estimator()

# construct feature map
from qiskit.circuit.library import ZZFeatureMap
feature_map = ZZFeatureMap(feature_dimension)

# construct ansatz
from qiskit.circuit.library import RealAmplitudes
ansatz = RealAmplitudes(feature_dimension, reps=reps)

# construct quantum circuit
from qiskit import QuantumCircuit
qc = QuantumCircuit(feature_dimension)
qc.append(feature_map, range(feature_dimension))
qc.append(ansatz, range(feature_dimension))
qc.decompose().draw()

# Build QNN
from qiskit_machine_learning.neural_networks import EstimatorQNN
estimator_qnn = EstimatorQNN(
    circuit=qc,
    input_params=feature_map.parameters,
    weight_params=ansatz.parameters,
    estimator = estimator
)

# create empty array for callback to store evaluations of the objective function
objective_func_vals = []

# callback function that draws a live plot when the .fit() method is called
from IPython.display import clear_output
def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()

```

```

# Create the neural network classifier
from qiskit_machine_learning.algorithms.classifiers import NeuralNetworkClassifier
from qiskit.algorithms.optimizers import COBYLA
estimator_classifier = NeuralNetworkClassifier(neural_network=estimator_qnn,
optimizer=COBYLA(), callback=callback_graph)

estimator_classifier = NeuralNetworkClassifier(estimator_qnn, optimizer=COBYLA(),
callback=callback_graph)

# fit classifier to data
estimator_classifier.fit(X_train, y_train)

# score classifier
estimator_classifier.score(X_train, y_train)

# Predict data points from X_test
y_predict = estimator_classifier.predict(X_test)

# fit classifier to data
estimator_classifier.fit(X_train, y_train)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score classifier
estimator_classifier.score(X_test, y_test)

# Predict data points from X_test dataset
y_predict = estimator_classifier.predict(X_test)

# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X_train, y_train, y_predict):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if y_target != y_p:
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()

print(estimator_classifier.score(X_test, y_test))

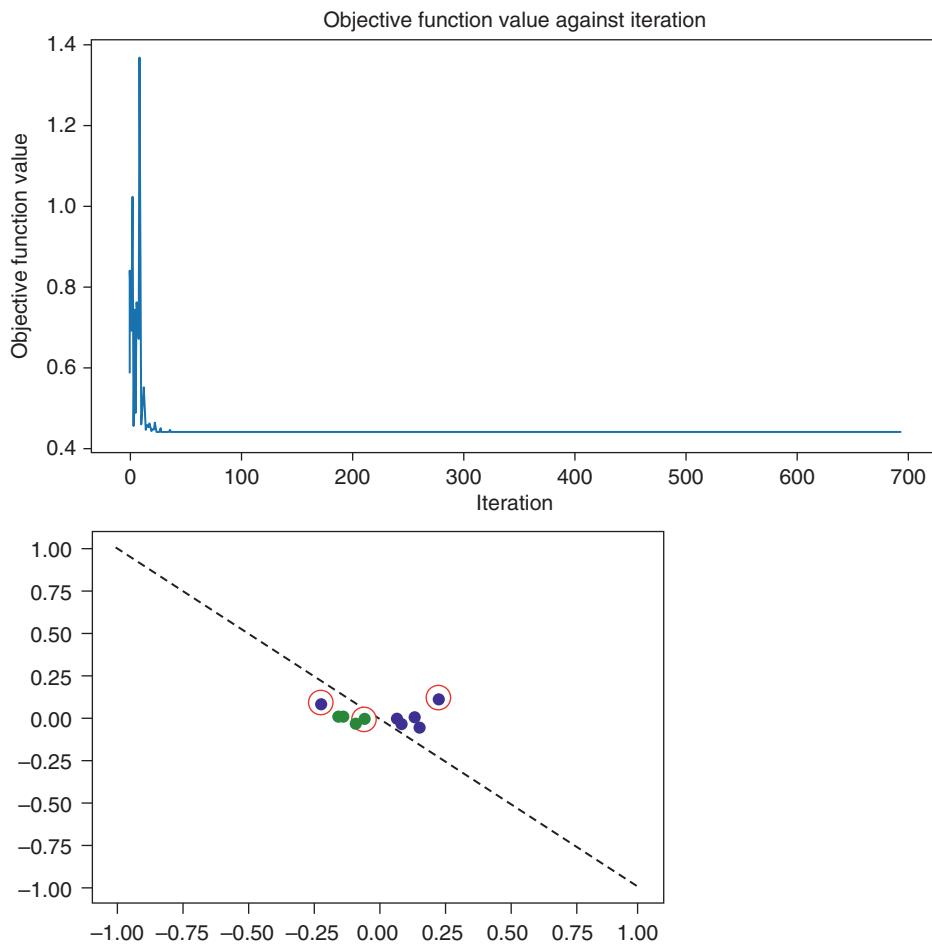
# Print predicted values and real values of the X_test dataset
print("\n")
print("Print predicted data coming from X_test as new input data")
print(y_predict)
print("\n")
print("Print real values\n")
print(y_test)
print("\n")

```

```
# Print accuracy metrics of the model
from sklearn import metrics
results = [metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_predict),
metrics.accuracy_score(y_test, y_predict), metrics.precision_score(y_test, y_predict,
average='micro'), metrics.recall_score(y_test, y_predict, average='micro'), metrics.
f1_score(y_test, y_predict, average='micro')]
metrics_dataframe = pd.DataFrame(results, index=["Balanced Accuracy", "Accuracy",
"Precision", "Recall", "F1 Score"], columns=['estimator_classifier'])
print('Classification Report: \n')
print(classification_report(y_test,y_predict))

metrics_dataframe
```

Output:



```
Print predicted data coming from X_test as new input data
[[-1.]
 [ 1.]
[-1.]
[-1.]
 [ 1.]
```

```
[ 1.]
[-1.]
[-1.]
[-1.]
[ 1.]
[-1.]
[ 1.]
[ 1.]]
```

Print real values

```
[-1  1   1 -1   1 -1   1 -1 -1   1 -1   1   1]
```

Classification Report:

	precision	recall	f1-score	support
-1	0.71	0.83	0.77	6
1	0.83	0.71	0.77	7
accuracy			0.77	13
macro avg	0.77	0.77	0.77	13
weighted avg	0.78	0.77	0.77	13

opflow_classifier	
Balanced accuracy	0.773810
Accuracy	0.769231
Precision	0.769231
Recall	0.769231
F1 score	0.769231

5.5.2 Classification with a SamplerQNN

In the next example, we will use a SamplerQNN to classify our neurons within a NeuralNetworkClassifier.

Input:

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

# Import dataset
data = '../data/datasets/neurons_binary.csv'
neuron = pd.read_csv(data, delimiter=';')

# Drop row having at least 1 missing value
neuron = neuron.dropna()
```

```

# We select some data from the dataset to compute faster
df = neuron.head(22).copy() # Principal neurons
df = pd.concat([df, neuron.iloc[17033:17053]]) # Interneurons

# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[["Target"]] = df[["Target"]].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]].to_numpy() # We remove labels and convert pandas DataFrame
into numpy array
y = df['Target'].to_numpy()

# Splitting the data : training and test (30%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Scaling the data
from sklearn.preprocessing import Normalizer
Normalize = preprocessing.Normalizer()
# Transform data
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
# transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)

# Variable definition
number_classes = 2 # Number of classes
feature_dimension = 2 # Number of qubits
quantum_backend = None # We use local simulator
reps = 2

if quantum_backend is not None:
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
    # Define service
    service = QiskitRuntimeService(channel = 'ibm_quantum', token = ibm_account,
instance = 'ibm-q/open/main')
    # Get backend
    backend = service.backend(quantum_backend) # Use a simulator or hardware from the cloud
    # Define Sampler
    # Run Quasi-Probability calculation

```

```

# optimization_level=3 adds dynamical decoupling
# resilience_level=1 adds readout error mitigation
from qiskit_ibm_runtime import Options
options = Options()
options.resilience_level = 1
options.execution.shots = 1024
options.optimization_level = 3
sampler = Sampler(session=backend, options = options)
else:
    from qiskit.primitives import Sampler
    sampler = Sampler()

# construct feature map
from qiskit.circuit.library import ZZFeatureMap
feature_map = ZZFeatureMap(feature_dimension)

# construct ansatz
from qiskit.circuit.library import RealAmplitudes
ansatz = RealAmplitudes(feature_dimension, reps=reps)

# construct quantum circuit
from qiskit import QuantumCircuit
qc = QuantumCircuit(feature_dimension)
qc.append(feature_map, range(feature_dimension))
qc.append(ansatz, range(feature_dimension))
qc.decompose().draw()

# Build QNN
from qiskit_machine_learning.neural_networks import SamplerQNN
sampler_qnn = SamplerQNN(
    circuit=qc,
    input_params=feature_map.parameters,
    weight_params=ansatz.parameters,
    output_shape=number_classes,
    sampler=sampler
)

# construct classifier
from qiskit_machine_learning.algorithms.classifiers import NeuralNetworkClassifier
from qiskit.algorithms.optimizers import COBYLA
sampler_classifier = NeuralNetworkClassifier(
    neural_network=sampler_qnn, optimizer=COBYLA(maxiter=30), callback=callback_graph
)

# create empty array for callback to store evaluations of the objective function
objective_func_vals = []

# fit classifier to data
sampler_classifier.fit(X_train, y_train)

# score classifier
sampler_classifier.score(X_train, y_train)

```

```

# evaluate data points
y_predict = sampler_classifier.predict(X_test)

# callback function that draws a live plot when the .fit() method is called
def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()

# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X_test, y_test, y_predict):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if y_target != y_p:
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()

# Print predicted values and real values of the X_test dataset
print("\n")
print("Print predicted data coming from X_test as new input data")
print(y_predict)
print("\n")
print("Print real values\n")
print(y_test)
print("\n")

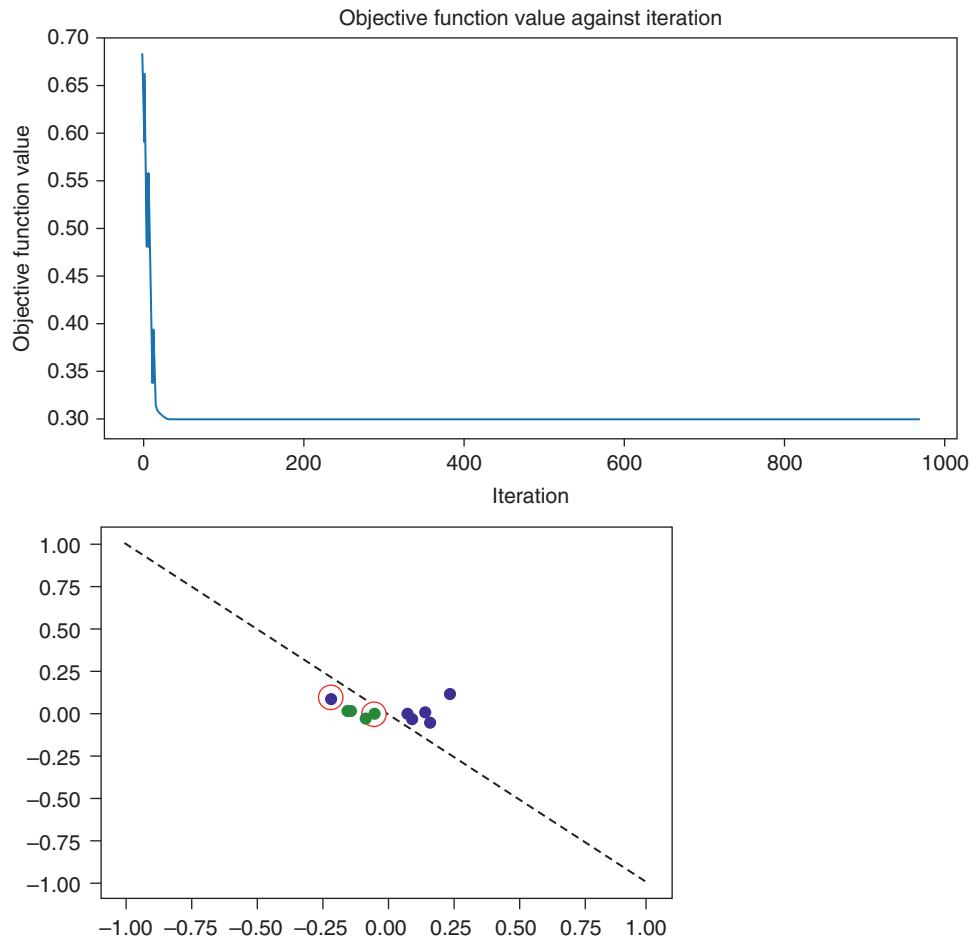
# Print accuracy metrics of the model
from sklearn import metrics
from sklearn.metrics import classification_report

results = [metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_predict),
metrics.accuracy_score(y_test, y_predict), metrics.precision_score(y_test, y_predict,
average='micro'), metrics.recall_score(y_test, y_predict, average='micro'), metrics.
f1_score(y_test, y_predict, average='micro')]
metrics_dataframe = pd.DataFrame(results, index=["Balanced Accuracy", "Accuracy",
"Precision", "Recall", "F1 Score"], columns=['sampler_classifier'])
print('Classification Report: \n')
print(classification_report(y_test, y_predict))

metrics_dataframe

```

Output:



```
Print predicted data coming from X_test as new input data
[0 1 0 0 1 1 1 0 0 1 0 1 1]
```

```
Print real values
[0 1 1 0 1 0 1 0 0 1 0 1 1]
```

Classification Report:

	precision	recall	f1-score	support
0	0.83	0.83	0.83	6
1	0.86	0.86	0.86	7
accuracy			0.85	13
macro avg	0.85	0.85	0.85	13
weighted avg	0.85	0.85	0.85	13

circuit_classifier	
Balanced accuracy	0.845238
Accuracy	0.846154
Precision	0.846154
Recall	0.846154
F1 score	0.846154

Points with red circles have been incorrectly classified.

5.5.3 Classification with Variational Quantum Classifier

A special variant of the NeuralNetworkClassifier with a SamplerQNN is the VQC. The cross-entropy loss function is applied by default, expecting labels that are one-hot encoded.

Input:

```

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

# Import dataset
data = '../data/datasets/neurons_binary.csv'
#data = '../data/datasets/neurons.csv'
neuron = pd.read_csv(data, delimiter=';')

# Drop row having at least 1 missing value
neuron = neuron.dropna()

# We select some data from the dataset to compute faster
df = neuron.head(22).copy() # Principal neurons
df = pd.concat([df, neuron.iloc[17033:17053]]) # Interneurons

# Creating an instance of Labelencoder
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
# Assigning numerical value and storing it
df[['Target']] = df[['Target']].apply(enc.fit_transform)

# Divide the data, y the variable to predict (Target) and X the features
X = df[df.columns[1:]].to_numpy() # We remove labels and convert pandas DataFrame
into numpy array
y = df['Target'].to_numpy()

# Splitting the data : training and test (30%)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Scaling the data
from sklearn.preprocessing import Normalizer
Normalize = preprocessing.Normalizer()
# Transform data

```

```
X_train = Normalize.fit_transform(X_train)
X_test = Normalize.fit_transform(X_test)

# Dimension Reduction with PCA (with two principal components)
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
# transform data
X_train = pca.fit_transform(X_train)
X_test = pca.fit_transform(X_test)

# One-hot encode target column
# A column will be created for each output category.
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Variable definition
number_classes = 2 # Number of classes
# Variable definition
feature_dimension = 2 # Number of qubits
quantum_backend = None # We use local simulator
reps = 2

# Create feature map, ansatz, and optimizer
from qiskit.circuit.library import ZZFeatureMap
feature_map = ZZFeatureMap(feature_dimension)
from qiskit.circuit.library import RealAmplitudes
ansatz = RealAmplitudes(feature_dimension, reps=reps)

if quantum_backend is not None:
    # Import QiskitRuntimeService and Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService, Sampler
    # Define service
    service = QiskitRuntimeService(channel = 'ibm_quantum', token = ibm_account,
instance = 'ibm-q-internal/deployed/default')
    # Get backend
    backend = service.backend(quantum_backend) # Use a simulator or hardware from the
cloud
    # Define Sampler
    # Run Quasi-Probability calculation
    # optimization_level=3 adds dynamical decoupling
    # resilience_level=1 adds readout error mitigation
    from qiskit_ibm_runtime import Options
    options = Options()
    options.resilience_level = 1
    options.execution.shots = 1024
    options.optimization_level = 3
    sampler = Sampler(session=backend, options = options)
else:
    from qiskit.primitives import Sampler
    sampler = Sampler()
```

```

from qiskit_machine_learning.algorithms.classifiers import VQC
vqc = VQC(
    feature_map=feature_map,
    ansatz=ansatz,
    loss="cross_entropy",
    optimizer=COBYLA(),
    sampler=sampler,
    callback=callback_graph,
)

# Create empty array for callback to store evaluations of the objective function
objective_func_vals = []

# fit classifier to data and calculate elapsed time
import time
start = time.time()
vqc.fit(X_train, y_train)
elapsed = time.time() - start

# score classifier
vqc.score(X_train, y_train)

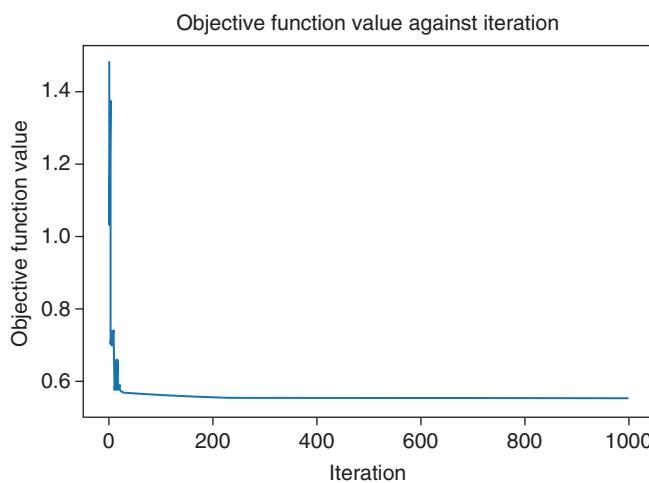
# Predict data points from X_test
y_predict = vqc.predict(X_test)

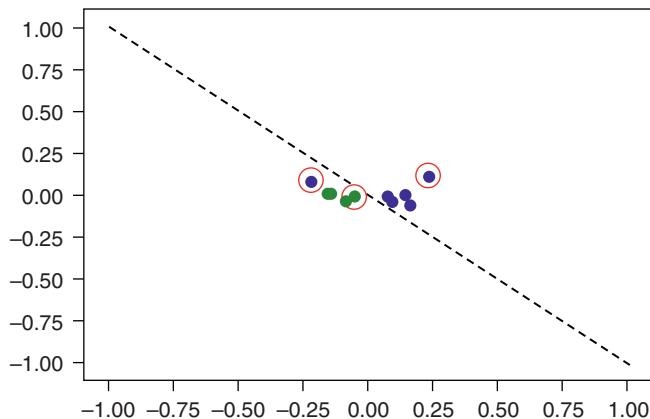
# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X_test, y_test, y_predict):
    if y_target[0] == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if not np.all(y_target == y_p):
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()

print(vqc.score(X_train, y_train))

```

Output:





0.9655172413793104

5.5.4 Regression

Regression can also be performed with similar methodology to VQC, EstimatorQNN, or SamplerQNN.

For example, we can use VQR instead, which is a special variant of the NeuralNetworkRegressor with an EstimatorQNN. It will consider the l2 loss function to minimize the mean squared error between targets and predictions.

In regression, we will use lines of code such as the following:

```
# construct QNN
regression_estimator_qnn = EstimatorQNN(
    circuit=qc, input_params=feature_map.parameters, weight_params=ansatz.parameters
)

# construct the regressor from the neural network
regressor = NeuralNetworkRegressor(
    neural_network=regression_estimator_qnn,
    loss="squared_error",
    optimizer=L_BFGS_B(maxiter=5),
    callback=callback_graph,
)

# fit to data
regressor.fit(X, y)

# score the result
regressor.score(X, y)
```

The following code can be used for VQR:

```
vqr = VQR(
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=L_BFGS_B(maxiter=5),
    callback=callback_graph,
)
```

5.6 Quantum Generative Adversarial Network

To produce new information from a given dataset, one possibility is to employ a quantum generative adversarial network (qGAN) combined with a hybrid quantum-classical algorithm specifically created for generative modeling tasks, as presented by Zoufal et al. (2019). This method takes advantage of the interplay between a quantum generator G_θ (also known as an ansatz) and a classical discriminator D_ϕ (a type of neural network) to grasp the fundamental probability distribution of the supplied training data. The training process for the generator and discriminator involves a series of alternating optimization steps. The generator's aim is to generate samples that the discriminator will identify as authentic training data samples (i.e., samples originating from the actual training distribution), while the discriminator endeavors to differentiate between genuine training data samples and those produced by the generator (essentially, distinguishing real and generated distributions). The ultimate objective is for the quantum generator to capture a representation of the basic probability distribution of the training data. As a result, the trained quantum generator can be employed to load a quantum state, which acts as an approximate model of the target distribution. The qGAN is a method used to learn the fundamental probability distribution of a collection of k -dimensional data samples and load it directly into a quantum state:

$$|g_\theta\rangle = \sum_{j=0}^{2^n - 1} \sqrt{p_\theta^j} |j\rangle$$

The qGAN training produces a state $|g_\theta\rangle$, which is composed of the occurrence probabilities of the basis states $|j\rangle$ defined by p_θ^j . The goal is to generate a probability distribution that closely resembles the distribution underpinning the training data $X = \{x_0, \dots, x_{k-1}\}$, with $j \in \{0, \dots, 2^n - 1\}$. To execute this algorithm, several steps must be followed in order to generate data using qGANs. To simplify the representation, the samples are converted into discrete values. The number of discrete values that can be represented depends on the quantity of qubits employed for mapping. As a result, the data's resolution is determined by the number of qubits utilized. For example, if 3 qubits are used to represent one feature, $2^3 = 8$ discrete values can be generated. In the example provided below, the training data was discretized using an array [5, 5], which indicates the number of qubits used to represent each data dimension, converted to PyTorch modeling, starting with the transformation of data arrays into tensors, and then creating a data loader from the training data. Following this, a backend (quantum hardware or simulator) is selected to operate the quantum generator. A quantum instance is then established for evaluation purposes, choosing 10,000 shots to obtain more comprehensive insights and launching the parameterized quantum circuit $G(\theta)$, where $\theta = \theta_1, \dots, \theta_k$, which will be used in the quantum generator. To implement the quantum generator, a depth-2 ansatz is chosen that employs RY rotations and CX gates and accepts a uniform distribution as input state. It is crucial to recognize that for $k > 1$, the generator's parameters must be selected carefully. For instance, the circuit depth should be more than 1 to enable the representation of more complex structures. Next, a function is created that generates the quantum generator using a specified parameterized quantum circuit. This function takes a quantum instance for data sampling as its parameters. The TorchConnector is utilized to encapsulate the created quantum neural network, facilitating PyTorch-based training. A classical neural network, representing the classical discriminator, is then constructed using PyTorch. The underlying gradients can be computed automatically through PyTorch's built-in features. Both the generator and the discriminator are trained using binary cross-entropy as the loss function:

$$L(\theta) = \sum_j p_j(\theta) \left[y_j \log(x_j) + (1 - y_j) \log(1 - x_j) \right]$$

where y_j are the labels of the data sample x_j .

To assess the custom gradients for the quantum generator, we combine the quantum gradients $\frac{\partial p_j(\theta)}{\partial \theta_l}$ – computed using Qiskit's gradient framework – with the binary cross-entropy in the following way:

$$\frac{\partial L(\theta)}{\partial \theta_l} = \sum_j \frac{\partial p_j(\theta)}{\partial \theta_l} \left[y_j \log(x_j) + (1 - y_j) \log(1 - x_j) \right]$$

We determine the relative entropy between the target and trained distributions, which serves as a measure of distance between probability distributions, to evaluate the closeness or divergence between the trained and target distributions. The ADAM optimizer can be employed to train both the generator and the discriminator.

It is time to start programming. We will be working with the following dataset available on GitHub at this link: <https://github.com/xaviervasques/hephaistos/blob/main/data/datasets/e-type.csv>.

The Jupyter Notebook of the code below can be found at this link: https://github.com/xaviervasques/hephaistos/blob/main/Notebooks/q_GAN_Pipeline.ipynb.

Input:

```
# Import the algorithm_globals from qiskit.utils for global settings management in
Qiskit algorithms.
from qiskit.utils import algorithm_globals

# Import the LabelEncoder class from sklearn.preprocessing for converting categorical
data into numerical form.
from sklearn.preprocessing import LabelEncoder

# Fixing seeds in the random number generators

# Set the random seed for PyTorch to ensure reproducibility in the generated random
numbers.
torch.manual_seed(42)

# Set the random seed for Qiskit's algorithm_globals to ensure reproducibility in the
generated random numbers used in Qiskit algorithms.
algorithm_globals.random_seed = 42

# Define the file path for the dataset as a string variable named 'mtype'.
mtype = '../data/datasets/e-type.csv'

# Load the dataset using pandas 'read_csv' function, specifying the delimiter as ';',
and store it in a DataFrame named 'df'.
df = pd.read_csv(mtype, delimiter=';')

# Rename the "e-type" column to "Target" in the 'df' DataFrame.
df = df.rename(columns={"e-type": "Target"})

# Display the first 5 rows of the DataFrame 'df' using the 'head' function.
df.head()
```

Output:

	Target	adaptation	avg_isi	electrode_0_pa	f_i_curve_slope	fast_trough_t_long_square	fast_trough_t_ramp	fast_trough_t_short_square	fast_trough_v_long_s
0	Exc_1		NaN	43.575	-21.118126	0.073476	1.078705	6.053648	1.025497
1	Exc_1	0.307737	86.730		6.264375	0.117905	1.072875	3.603470	1.028330
2	Exc_1		NaN	46.435	4.067499	0.068452	1.080250	6.163475	1.025503
3	Exc_1	0.189721	52.050		9.760625	0.073864	1.086980	4.466840	1.026288
4	Exc_1		NaN	41.485	16.541874	0.055147	1.073525	6.798208	1.025251

5 rows × 49 columns

Input:

```
# Creating instance of labelencoder

# Instantiate a LabelEncoder object and store it in the variable 'labelencoder'.
labelencoder = LabelEncoder()

# Assigning numerical values and storing in another column

# Apply the 'fit_transform' method of the 'labelencoder' object to the 'Target' column
# of the DataFrame 'df',
# converting the categorical values to numerical values, and store the results back in
# the 'Target' column.
df['Target'] = labelencoder.fit_transform(df['Target'])

# Display the first 5 rows of the DataFrame 'df' using the 'head' function to show the
# updated 'Target' column.
df.head()
```

Output:

Target	adaptation	avg_isi	electrode_0_pa	f_i_curve_slope	fast_trough_t_long_square	fast_trough_t_ramp	fast_trough_t_short_square	fast_trough_v_long_s
0	0	NaN	43.575	-21.118126	0.073476	1.078705	6.053648	1.025497
1	0	0.307737	86.730	6.264375	0.117905	1.072875	3.603470	1.028330
2	0	NaN	46.435	4.067499	0.068452	1.080250	6.163475	1.025503
3	0	0.189721	52.050	9.760625	0.073864	1.086980	4.466840	1.026288
4	0	NaN	41.485	16.541874	0.055147	1.073525	6.798208	1.025251

5 rows × 9 columns

Input:

```
# Handling missing values using KNN

# Import the KNNImputer class from the sklearn.impute module for handling missing
# values using k-Nearest Neighbors.
from sklearn.impute import KNNImputer

# Instantiate a KNNImputer object with 5 nearest neighbors and store it in the
# variable 'KNN_imputer'.
KNN_imputer = KNNImputer(n_neighbors=5)

# Apply the 'fit_transform' method of the 'KNN_imputer' object to the entire DataFrame
# 'df',
# imputing missing values based on the 5 nearest neighbors, and store the results back
# in 'df'.
df.iloc[:, :] = KNN_imputer.fit_transform(df)

# Display the first 5 rows of the DataFrame 'df' using the 'head' function to show the
# imputed values.
df.head()

# Select only one class
# Set the type_number variable to the class label you want to filter for
type_number = 0

# Filter the df_train DataFrame to include only instances where the 'Target' column
# is equal to the specified type_number
df = df[df['Target'] == type_number]

# Number of classes (starts from 0)
```

```

# Determine the number of unique classes in the 'Target' column by using the 'groupby'
method followed by 'size' and 'index',
# and store the result in the variable 'number_classes'.
number_classes = len(df.groupby('Target').size().index)

# Splitting data into training and testing dataset.

# Divide the data, y the variable to predict (Target) and X the features
# Assign all columns except the 'Target' column from the DataFrame 'df' to the
variable 'X'.
X = df[df.columns[1:]]

# Assign the 'Target' column from the DataFrame 'df' to the variable 'y'.
y = df['Target']

# Import the 'train_test_split' function from the sklearn.model_selection module.
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets using 'train_test_split' with a test
size of 0.5 (50%),
# stratifying the split based on 'y_all' to ensure a balanced distribution of classes,
and set the random state to 42 for reproducibility.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, stratify=y,
random_state=42)
X_train

```

Output:

	adaptation	avg_isi	electrode_0_pa	f_i_curve_slope	fast_trough_t_long_square	fast_trough_t_ramp	fast_trough_t_short_square	fast_trough_v_long_squ
46	0.067315	81.025458	12.654999	6.170382e-02	1.125140	5.722320	1.025400	-40.343
13	0.051659	56.770000	36.682501	2.766571e-02	1.110005	7.496912	1.025258	-44.187
24	0.153268	51.340000	9.057500	2.231405e-02	1.091960	7.882660	1.025660	-49.000
29	0.071684	353.060000	12.292499	8.287195e-02	1.059380	12.011627	1.025460	-45.156
25	0.046149	43.741958	3.225000	4.962122e-02	1.067040	10.278500	1.024992	-47.843
32	-0.001545	68.222828	0.390000	3.181818e-02	1.078520	8.199280	1.025616	-48.156
37	0.088268	119.593008	-146.452489	-6.639485e-19	1.077300	8.798380	1.026088	-44.906
8	0.097713	87.650929	-19.316250	1.587429e-02	1.082460	7.792698	1.025134	-47.843
17	0.071077	30.050000	64.213752	2.772021e-02	1.094465	4.749478	1.024862	-47.031
15	0.060014	52.677361	17.478751	4.488449e-02	1.086560	5.307645	1.025161	-46.781
9	0.126352	103.819271	-50.917499	4.080051e-02	1.054105	4.010890	1.024726	-49.375
43	0.052010	168.022975	44.069998	6.407850e-02	1.044700	9.970560	1.025624	-50.843
27	-0.007823	57.806557	22.672499	1.553030e-02	1.053760	14.329147	1.025384	-47.906
26	0.168379	84.563588	13.892499	2.372276e-02	1.099940	5.601700	1.025244	-46.125
19	0.077910	100.895750	-73.295627	3.787313e-02	1.097060	7.448642	1.025102	-43.593
6	0.084453	47.555000	18.617498	4.050633e-02	1.080785	4.441173	1.024957	-47.656
3	0.189721	52.050000	9.760625	7.386363e-02	1.086980	4.466840	1.026288	-49.687
12	0.208855	116.698827	-11.355626	2.126437e-02	1.063520	12.455952	1.024981	-38.937
33	0.163061	212.499708	-1.092500	6.962027e-03	1.071380	15.262633	1.024836	-47.843
40	0.046284	70.705813	29.337501	7.366073e-03	1.086700	17.317110	1.025633	-46.562
16	0.035890	81.615000	-36.987500	2.807364e-02	1.076205	5.598615	1.025151	-46.312
4	0.152914	41.485000	16.541874	5.514706e-02	1.073525	6.798208	1.025251	-48.031
45	0.074782	52.149331	76.700001	2.415254e-02	1.110000	8.125130	1.025240	-47.375
39	0.106168	50.260000	-9.787500	1.442112e-02	1.077040	9.301447	1.025920	-42.250

24 rows x 48 columns

Input:

```
# Import necessary libraries
from sklearn import preprocessing
from sklearn.neighbors import NeighborhoodComponentsAnalysis

# Data rescaling with RobustScaler
# Instantiate a RobustScaler object which is robust to outliers
scaler = preprocessing.RobustScaler()

# Fit the scaler to the feature data (X) and transform it
# The transformed data will have the same number of features as the input,
# but with a different scale
X_train = scaler.fit_transform(X_train)

# Load and apply Neighborhood Components Analysis (NCA)
# Instantiate a NeighborhoodComponentsAnalysis object with n_components=2,
# using PCA for initialization and a fixed random state for reproducibility
extraction = NeighborhoodComponentsAnalysis(n_components=2, init="pca",
random_state=42)

# Fit the NCA model to the feature data (X) and the target variable (y)
# and transform the data using the learned model
# The transformed data (training_data) will have 2 components, as specified
# by n_components
training_data = extraction.fit(X_train, y_train).transform(X_train)

# Display the transformed training_data
print(training_data)
```

Output:

```
[[ 1.18632791 -3.36745199]
 [ 0.68146915 -0.7093939 ]
 [ 0.24404409 -0.78579509]
 [-1.42294384  5.49353715]
 [-1.28454583  1.40429752]
 [-0.57685353 -0.34843588]
 [-0.40164212  0.10773078]
 [13.65363451  1.93219055]
 [ 6.57909835 -3.09613086]
 [ 0.45864905 -2.99663493]
 [14.43916508  1.37774594]
 [-1.65068257  2.86286648]
 [-2.10741473  3.93706039]
 [ 0.72275431 -1.66833441]
 [ 1.07689893 -0.31745944]
 [ 0.37896589 -2.47945056]
 [-0.17086296 -2.33438325]
 [-2.18486917  3.46464634]
 [-0.06455965  4.93661457]
 [-1.0877558  3.32550378]
 [ 0.05539514 -2.20910866]
 [17.13972368  2.15327072]
 [ 0.27363331 -1.69303335]
 [-0.66780033  1.23111316]]
```

Input:

```
# Import necessary libraries
import numpy as np
from qiskit_machine_learning.datasets.dataset_helper import discretize_and_truncate

# Assuming training_data is already defined as a 2D array or DataFrame
# resulting from the dimensionality reduction process

# Define minimal and maximal values for the training data
# Calculate the 5th percentile of the training data along each dimension (axis=0)
bounds_min = np.percentile(training_data, 5, axis=0)
# Calculate the 95th percentile of the training data along each dimension (axis=0)
bounds_max = np.percentile(training_data, 95, axis=0)

# Create a list of bounds for each dimension of the training data
# Each element of the list is a pair of the form [min_value, max_value]
bounds = []
for i, _ in enumerate(bounds_min):
    bounds.append([bounds_min[i], bounds_max[i]])

# Determine data resolution for each dimension of the training data in terms
# of the number of qubits used to represent each data dimension
data_dim = [5, 5]

# Pre-processing, i.e., discretization of the data (gridding)
# Call the discretize_and_truncate function to discretize the training_data
# according to the specified bounds and data_dim
(training_data, grid_data, grid_elements, prob_data) = discretize_and_truncate(
    training_data,
    np.asarray(bounds),
    data_dim,
    return_data_grid_elements=True,
    return_prob=True,
    prob_non_zero=True,
)

# Display the bounds
print(bounds)
```

Output:

```
[[ -2.0389049096518312, 14.321335491603316], [-3.081206467148244, 4.786681446847597]]
```

Input:

```
# Print training_data
training_data
```

Output:

```
array ([[ 0.59984354, -0.79698094],
       [ 0.07209385, -0.79698094],
       [-1.51115522,  1.48724458],
       [-0.45565584, -0.28937527],
       [-0.45565584,  0.2182304 ],
       [13.7935858,   1.99485025],
       [ 6.40509014, -3.08120647],
       [ 0.59984354, -3.08120647],
       [14.32133549,  1.48724458],
       [-1.51115522,  2.75625876],
       [-2.03890491,  4.02527294],
       [ 0.59984354, -1.55838945],
       [ 1.12759323, -0.28937527],
       [ 0.59984354, -2.5736008 ],
       [ 0.07209385, -2.31979796],
       [-2.03890491,  3.51766727],
       [-0.98340553,  3.26386443],
       [ 0.07209385, -2.31979796],
       [ 0.07209385, -1.81219229],
       [-0.45565584,  1.23344174]])
```

Input:

```
# Import necessary libraries
import torch
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from matplotlib import cm

# Assuming training_data and grid_elements are already defined as 2D arrays or DataFrames
# resulting from the data pre-processing process

# Convert training_data and grid_elements to PyTorch tensors with a float data type
training_data = torch.tensor(training_data, dtype=torch.float)
grid_elements = torch.tensor(grid_elements, dtype=torch.float)

# Define the training batch size
batch_size = 10

# Create a DataLoader object to handle batching of the training_data
# Shuffle the data before each epoch and drop the last incomplete batch
dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True,
drop_last=True)

# Create a histogram of the first and second variables of the training_data
# using matplotlib

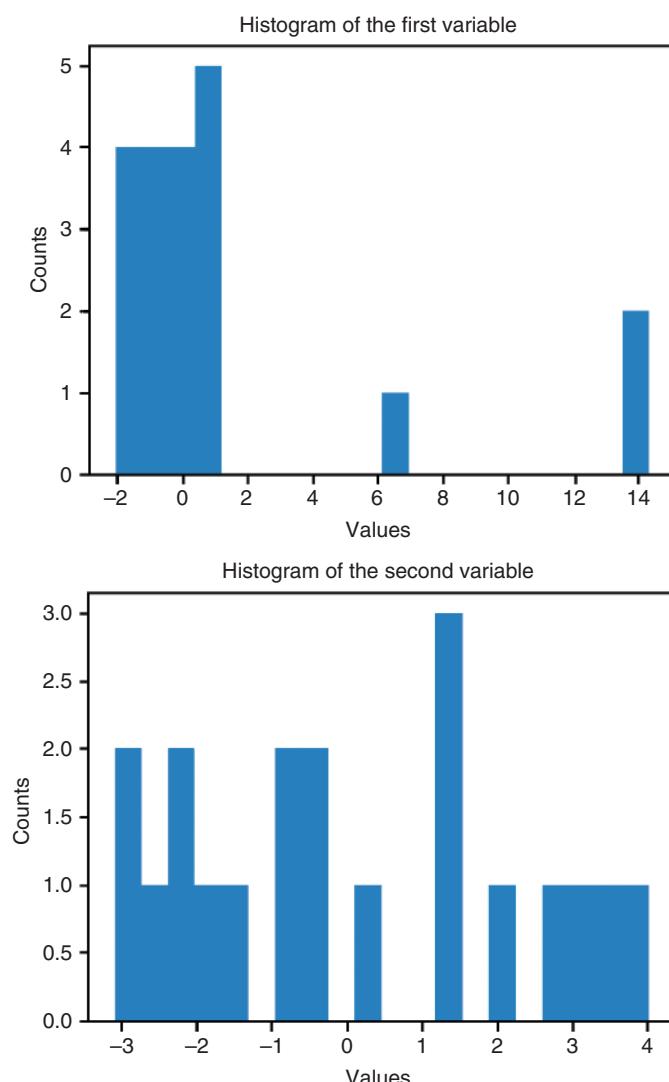
# Create a figure with two subplots (ax1 and ax2)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

# Plot a histogram of the first variable (column 0) of the training_data
# using 20 bins in the ax1 subplot
ax1.hist(training_data[:, 0], bins=20)
ax1.set_title("Histogram of the first variable")
ax1.set_xlabel("Values")
ax1.set_ylabel("Counts")
```

```
# Plot a histogram of the second variable (column 1) of the training_data
# using 20 bins in the ax2 subplot
ax2.hist(training_data[:, 1], bins=20)
ax2.set_title("Histogram of the second variable")
ax2.set_xlabel("Values")
ax2.set_ylabel("Counts")

# Display the figure
fig.show()
```

Output:



Input:

```
# Import necessary libraries
from qiskit import IBMQ
from qiskit.utils import QuantumInstance

# Save and load the IBMQ account with the provided API key
IBMQ.save_account('YOUR API', overwrite=True)
IBMQ.load_account()
```

```

# Get the provider and the backend for the quantum simulations
provider = IBMQ.get_provider(hub='YOUR PROVIDER')
backend = provider.get_backend('YOUR BACKEND')

# Create QuantumInstance objects for training and sampling, setting the number of shots
qi_training = QuantumInstance(backend, shots=batch_size)
qi_sampling = QuantumInstance(backend, shots=10000)

# Import necessary libraries for building the quantum circuit
from qiskit import Aer, QuantumCircuit
from qiskit.circuit.library import TwoLocal

# Assuming data_dim is already defined as a list of data resolutions for each dimension

# sum(data_dim) corresponds to the total number of qubits in our quantum circuit (qc)
qc = QuantumCircuit(sum(data_dim))

# Apply Hadamard (H) gates to all qubits in the quantum circuit
qc.h(qc.qubits)

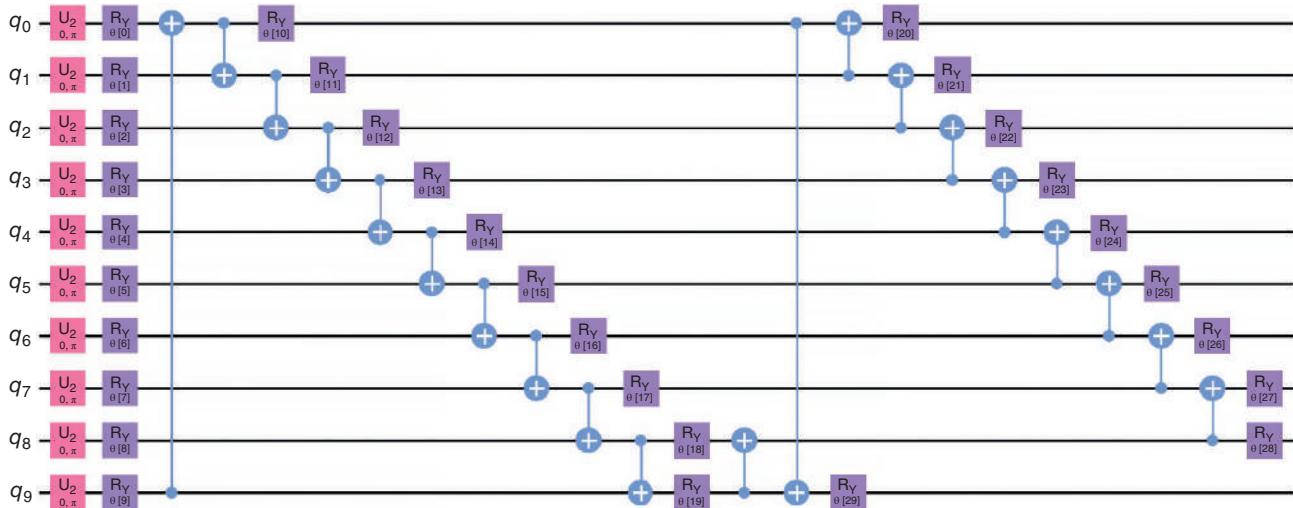
# Choose a hardware-efficient ansatz
# Create a TwoLocal object with rotation gates (RY), CNOT (CX) gates, 2 repetitions,
# and sparse_cyclic (sca) entanglement
twolocal = TwoLocal(sum(data_dim), "ry", "cx", reps=2, entanglement="sca")

# Compose the quantum circuit (qc) with the TwoLocal object (twolocal)
qc.compose(twolocal, inplace=True)

# Draw the decomposed quantum circuit using the matplotlib (mpl) backend
qc.decompose().draw("mpl")

```

Output:



Input:

```
# Import necessary libraries
from qiskit_machine_learning.connectors import TorchConnector
from qiskit_machine_learning.neural_networks import CircuitQNN
from torch.optim import Adam
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from scipy.stats import entropy

# Function to create a generator using a TorchConnector and a CircuitQNN
def create_generator(quantum_instance) -> TorchConnector:
    circuit_qnn = CircuitQNN(
        qc,
        input_params=[],
        weight_params=qc.parameters,
        quantum_instance=quantum_instance,
        sampling=True,
        sparse=False,
        interpret=lambda x: grid_elements[x],
    )

    return TorchConnector(circuit_qnn)

# Define a Discriminator class as a subclass of nn.Module
class Discriminator(nn.Module):
    def __init__(self, input_size):
        super(Discriminator, self).__init__()

        self.linear_input = nn.Linear(input_size, 20)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.linear20 = nn.Linear(20, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, input: torch.Tensor) -> torch.Tensor:
        x = self.linear_input(input)
        x = self.leaky_relu(x)
        x = self.linear20(x)
        x = self.sigmoid(x)
        return x

# Define generator and discriminator loss functions
gen_loss_fun = nn.BCELoss()
disc_loss_fun = nn.BCELoss()

# Import necessary libraries for gradient calculation
from qiskit.opflow import Gradient, StateFn

# Define the generator gradient
generator_grad = Gradient().gradient_wrapper(
    StateFn(qc), two-local.ordered_parameters, backend=qi_training
)
```

```

# Define the generator loss gradient function
def generator_loss_grad(parameter_values, discriminator):
    # Evaluate gradient
    grads = generator_grad(parameter_values).tolist()

    loss_grad = ()
    for j, grad in enumerate(grads):
        cx = grad[0].tocoo()
        input = torch.zeros(len(cx.col), len(data_dim))
        target = torch.ones(len(cx.col), 1)
        weight = torch.zeros(len(cx.col), 1)

        for i, (index, prob_grad) in enumerate(zip(cx.col, cx.data)):
            input[i, :] = grid_elements[index]
            weight[i, :] = prob_grad
        bce_loss_grad = F.binary_cross_entropy(discriminator(input), target, weight)
        loss_grad += (bce_loss_grad,)
    loss_grad = torch.stack(loss_grad)
    return loss_grad

# Define a function to calculate the relative entropy between generated data and true
# data
def get_relative_entropy(gen_data) -> float:
    prob_gen = np.zeros(len(grid_elements))
    for j, item in enumerate(grid_elements):
        for gen_item in gen_data.detach().numpy():
            if np.allclose(np.round(gen_item, 6), np.round(item, 6), rtol=1e-5):
                prob_gen[j] += 1
    prob_gen = prob_gen / len(gen_data)
    prob_gen = [1e-8 if x == 0 else x for x in prob_gen]
    return entropy(prob_gen, prob_data)

# Initialize generator and discriminator
generator = create_generator(qi_training)
discriminator = Discriminator(len(data_dim))

lr = 0.01 # learning rate
b1 = 0.9 # first momentum parameter
b2 = 0.999 # second momentum parameter
num_epochs = 100 # number of training epochs

# Optimizer for the generator
optimizer_gen = Adam(generator.parameters(), lr=lr, betas=(b1, b2))
# Optimizer for the discriminator
optimizer_disc = Adam(discriminator.parameters(), lr=lr, betas=(b1, b2))

# Import necessary libraries
from IPython.display import clear_output
from scipy.stats import entropy
import matplotlib.pyplot as plt
import torch

```

```

# Function to plot training progress
def plot_training_progress():
    # We don't plot if we don't have enough data
    if len(generator_loss_values) < 2:
        return

    clear_output(wait=True)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

    # Loss
    ax1.set_title("Loss")
    ax1.plot(generator_loss_values, label="generator loss", color="royalblue")
    ax1.plot(discriminator_loss_values, label="discriminator loss", color="magenta")
    ax1.legend(loc="best")
    ax1.set_xlabel("Iteration")
    ax1.set_ylabel("Loss")
    ax1.grid()

    # Relative Entropy
    ax2.set_title("Relative entropy")
    ax2.plot(relative_entropy_values)
    ax2.set_xlabel("Iteration")
    ax2.set_ylabel("Relative entropy")
    ax2.grid()

    plt_name = 'Loss_Relative_Entropy_%i.png'%type_number
    fig

```

Initialize lists for storing relative entropy, generator loss, and discriminator loss values

```

relative_entropy_values = []
generator_loss_values = []
discriminator_loss_values = []

# Training loop
for epoch in range(num_epochs):
    relative_entropy_epoch = []
    generator_loss_epoch = []
    discriminator_loss_epoch = []
    for i, data in enumerate(dataloader):
        # Adversarial ground truths
        valid = torch.ones(data.size(0), 1)
        fake = torch.zeros(data.size(0), 1)

        # Generate a batch of data points
        gen_data = generator()

        # Evaluate Relative Entropy
        relative_entropy_epoch.append(get_relative_entropy(gen_data))

        # Train Discriminator
        optimizer_disc.zero_grad()

```

```

# Loss measures discriminator's ability to distinguish real from generated samples
disc_data = discriminator(data)
real_loss = disc_loss_fun(disc_data, valid)
fake_loss = disc_loss_fun(discriminator(gen_data), fake)
discriminator_loss = (real_loss + fake_loss) / 2

discriminator_loss.backward(retain_graph=True)
optimizer_disc.step()

# Train Generator
optimizer_gen.zero_grad()

# Loss measures generator's ability to prepare good data samples
generator_loss = gen_loss_fun(discriminator(gen_data), valid)
generator_loss,retain_grad = True
g_loss_grad=generator_loss.grad(generator.weight.data.numpy(), discriminator)

# generator_loss.backward(retain_graph=True)
for j, param in enumerate(generator.parameters()):
    param.grad = g_loss_grad
optimizer_gen.step()

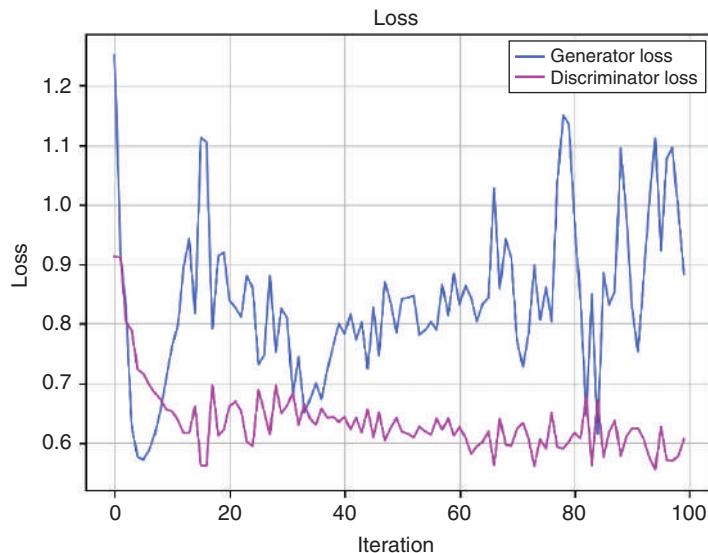
generator_loss_epoch.append(generator_loss.item())
discriminator_loss_epoch.append(discriminator_loss.item())

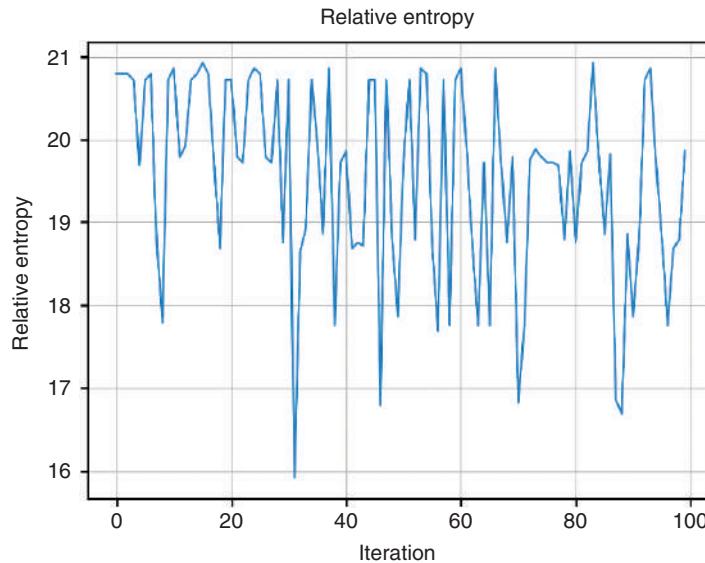
relative_entropy_values.append(np.mean(relative_entropy_epoch))
generator_loss_values.append(np.mean(generator_loss_epoch))
discriminator_loss_values.append(np.mean(discriminator_loss_epoch))

# Plot the training progress
plot_training_progress()

```

Output:





Input:

```
# Create a generator for sampling
generator_sampling = create_generator(qi_sampling)
generator_sampling.weight.data = generator.weight.data

# Generate data using the created generator
gen_data = generator_sampling().detach().numpy()
prob_gen = np.zeros(len(grid_elements))

# Calculate the probability distribution for generated data
for j, item in enumerate(grid_elements):
    for gen_item in gen_data:
        if np.allclose(np.round(gen_item, 6), np.round(item, 6), rtol=1e-5):
            prob_gen[j] += 1
prob_gen = prob_gen / len(gen_data)

# Replace zero probabilities with a very small value (1e-8)
prob_gen = [1e-8 if x == 0 else x for x in prob_gen]

# Plot the cumulative distribution function for generated and training data
fig = plt.figure(figsize=(12, 12))
ax1 = fig.add_subplot(111, projection="3d")
ax1.set_title("Cumulative Distribution Function")

ax1.bar3d(
    np.transpose(grid_elements) [1],
    np.transpose(grid_elements) [0],
    np.zeros(len(prob_gen)),
    0.05,
    0.05,
    np.cumsum(prob_gen),
    label="generated data",
    color="blue",
    alpha=1,
)
```

```

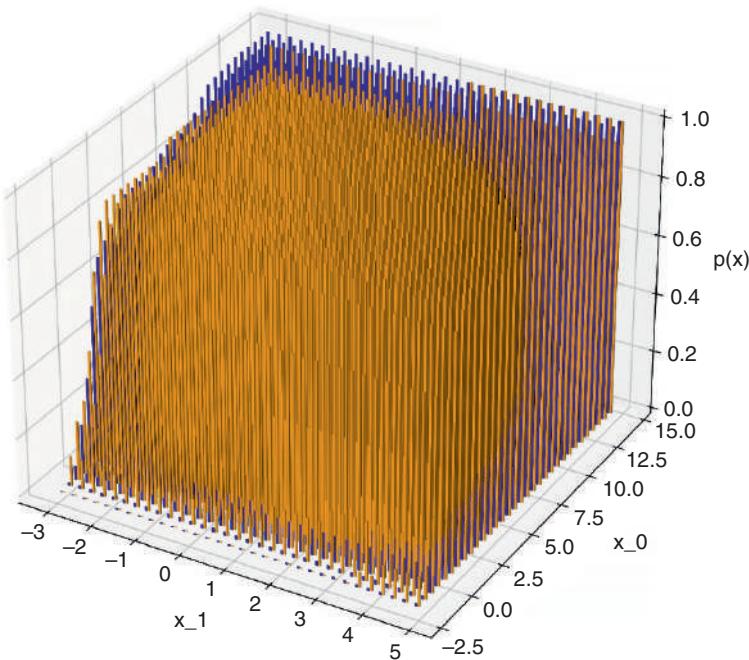
ax1.bar3d(
    np.transpose(grid_elements) [1] + 0.05,
    np.transpose(grid_elements) [0] + 0.05,
    np.zeros(len(prob_data)),
    0.05,
    0.05,
    np.cumsum(prob_data),
    label="training data",
    color="orange",
    alpha=1,
)
ax1.set_xlabel("x_1")
ax1.set_ylabel("x_0")
ax1.set_zlabel("p(x)")

plt_2_name = 'Cumulative_Distribution_Function_%i.png'%type_number
# plt.savefig(plt_2_name)
fig

```

Output:

Cumulative distribution function



Input:

```

# Apply the inverse PCA transformation to the generated data
X_orig = np.dot(gen_data, extraction.components_)

# Inverse scale the data to its original range
X_orig_backscaled = scaler.inverse_transform(X_orig)
X_orig_backscaled

```

```
# Define the column names for the DataFrame
column_names = list(df.columns)

# Create a DataFrame with the back-scaled data and the column names
data = pd.DataFrame(X_orig_backscaled, columns=[
    'adaptation',
    'avg_isi',
    'electrode_0_pa',
    'f_i_curve_slope',
    'fast_trough_t_long_square',
    'fast_trough_t_ramp',
    'fast_trough_t_short_square',
    'fast_trough_v_long_square',
    'fast_trough_v_ramp',
    'fast_trough_v_short_square',
    'input_resistance_mohm',
    'latency',
    'peak_t_long_square',
    'peak_t_ramp',
    'peak_t_short_square',
    'peak_v_long_square',
    'peak_v_ramp',
    'peak_v_short_square',
    'ri',
    'sag',
    'seal_gohm',
    'slow_trough_t_long_square',
    'slow_trough_t_ramp',
    'slow_trough_t_short_square',
    'slow_trough_v_long_square',
    'slow_trough_v_ramp',
    'slow_trough_v_short_square',
    'tau',
    'threshold_i_long_square',
    'threshold_i_ramp',
    'threshold_i_short_square',
    'threshold_t_long_square',
    'threshold_t_ramp',
    'threshold_t_short_square',
    'threshold_v_long_square',
    'threshold_v_ramp',
    'threshold_v_short_square',
    'trough_t_long_square',
    'trough_t_ramp',
    'trough_t_short_square',
    'trough_v_long_square',
    'trough_v_ramp',
    'trough_v_short_square',
    'upstroke_downstroke_ratio_long_square',
    'upstroke_downstroke_ratio_ramp',
    'upstroke_downstroke_ratio_short_square',
    'vm_for_sag',
    'vrest'
])
```

```
# Add a 'Target' column to the DataFrame
data['Target'] = j

# Save the DataFrame to a CSV file
output_file = 'data_gan_%i.csv'%type_number
data.to_csv(output_file, index=False)
```

A CSV file called data_gan_0.csv will be created, containing the freshly generated data.

5.7 Quantum Algorithms with HephAlistos

To execute the following example, create a Python file within the hephAlistos directory and proceed to run it. Quantum machine learning algorithms and processes such as those described above and including q_kernel_default, q_kernel_8, q_kernel_9, q_kernel_10, q_kernel_11, q_kernel_12, q_kernel_zz, q_kernel_training, q_kernel_8_pegasos, q_kernel_9_pegasos, q_kernel_10_pegasos, q_kernel_11_pegasos, q_kernel_12_pegasos, q_circuitqnn, q_twolayerqnn, and q_vqc can be easily handled using hephAlistos. For q_kernel_training, we need to adapt several parameters that are in the code such as the setup of the optimizer:

```
spsa_opt = SPSA(maxiter=10, callback=cb_qkt.callback, learning_rate=0.05,
perturbation=0.05)
```

The rotational layer for training and the number of circuits must also be specified:

```
# Create a rotational layer to train. We will rotate each qubit the same amount.
training_params = ParameterVector("θ", 1)
fm0 = QuantumCircuit(feature_dimension)
for qubit in range(feature_dimension):
    fm0.ry(training_params[0], qubit)
```

Let us view some examples. We will run the exact procedure we have run above with the q_kernel_zz algorithm, which includes data rescaling with the quantile uniform technique, label encoding of the “Target” column, repetition (the number of times the feature map circuit is repeated) set to 2, the use of the statevector simulator, the selection of five features using embedded decision tree, the use of five qubits, and a fivefold cross-validation to measure the quality of our model.

Input:

```
#!/usr/bin/python3
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

from data.datasets import neurons_maha_soma
neuron = neurons_maha_soma()

dfb5 = neuron.head(22).copy() # Ganglion
dfb5 = pd.concat([dfb5, neuron.iloc[320:340]]) # Granule
dfb5 = pd.concat([dfb5, neuron.iloc[1493:1513]]) # Medium Spiny
dfb5 = pd.concat([dfb5, neuron.iloc[1171:1191]]) # Parachromaffin
dfb5 = pd.concat([dfb5, neuron.iloc[10031:10051]]) # Pyramidal
```

```

dfb5 = pd.concat([dfb5, neuron.iloc[2705:2725]]) # Basket
dfb5 = pd.concat([dfb5, neuron.iloc[22589:22609]]) # Bitufted
dfb5 = pd.concat([dfb5, neuron.iloc[3175:3195]]) # Chandelier
dfb5 = pd.concat([dfb5, neuron.iloc[22644:22664]]) # Double bouquet
dfb5 = pd.concat([dfb5, neuron.iloc[3199:3219]]) # Martinotti
dfb5 = pd.concat([dfb5, neuron.iloc[8260:8280]]) # Nitrergic

dfb5 = pd.concat([dfb5, neuron.iloc[2255:2275]]) # Astrocytes
dfb5 = pd.concat([dfb5, neuron.iloc[3306:3326]]) # Microglia

scale = ['quantile_uniform']

selection = ['embedded_decision_tree_classifier']

kernel = ['q_kernel_zz']

for rescale in scale:
    for select in selection:
        for k in kernel:
            try:
                ml_pipeline_function(dfb5, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = rescale, feature_selection = select, k_features=5, cv = 5,
quantum_algorithms = [k], reps = 2, ibm_account = 'YOUR API', quantum_backend =
'statevector_simulator')
            except Exception as e: print(e)

```

The above code will produce the same results as previously obtained. We could also add the multiclass option (`multi-class = 'OneVsRestClassifier' or 'OneVsOneClassifier' or 'SVC'`) if we want to pass our quantum kernel to SVC from scikit-learn or “None” if we want to use QSVC from Qiskit.

For pegasos algorithms, we need to add the following options:

- `n_steps` = the number of steps performed during the training procedure.
- `C` = the regularization parameter.

If we want to create a pipeline to benchmark different quantum algorithms with a combination of feature rescaling and feature selection techniques, we can use the following code:

```

#!/usr/bin/python3
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Dataset
from data.datasets import neurons_maha_soma
neuron = neurons_maha_soma()

dfb5 = neuron.head(22).copy() # Ganglion
dfb5 = pd.concat([dfb5, neuron.iloc[320:340]]) # Granule
dfb5 = pd.concat([dfb5, neuron.iloc[1493:1513]]) # Medium Spiny
dfb5 = pd.concat([dfb5, neuron.iloc[1171:1191]]) # Parachromaffin
dfb5 = pd.concat([dfb5, neuron.iloc[10031:10051]]) # Pyramidal

```

```

dfb5 = pd.concat([dfb5, neuron.iloc[2705:2725]]) # Basket
dfb5 = pd.concat([dfb5, neuron.iloc[22589:22609]]) # Bitufted
dfb5 = pd.concat([dfb5, neuron.iloc[3175:3195]]) # Chandelier
dfb5 = pd.concat([dfb5, neuron.iloc[22644:22664]]) # Double bouquet
dfb5 = pd.concat([dfb5, neuron.iloc[3199:3219]]) # Martinotti
dfb5 = pd.concat([dfb5, neuron.iloc[8260:8280]]) # Nitrergic

dfb5 = pd.concat([dfb5, neuron.iloc[2255:2275]]) # Astrocytes
dfb5 = pd.concat([dfb5, neuron.iloc[3306:3326]]) # Microglia

scale = [
    'standard_scaler',
    'minmax_scaler',
    'maxabs_scaler',
    'robust_scaler',
    'normalizer',
    'log_transformation',
    'square_root_transformation',
    'reciprocal_transformation',
    'box_cox',
    'yeo_johnson',
    'quantile_gaussian',
    'quantile_uniform',
]
selection = [
    'variance_threshold',
    'chi_square',
    'anova_f_c',
    'pearson',
    'forward_stepwise',
    'backward_elimination',
    'exhaustive',
    'lasso',
    'feat_reg_ml',
    'embedded_linear_regression',
    'embedded_logistic_regression',
    'embedded_random_forest_classifier',
    'embedded_decision_tree_classifier',
    'embedded_xgboost_classification',
]
kernel = ['q_kernel_default', 'q_kernel_8', 'q_kernel_9', 'q_kernel_10',
          'q_kernel_11', 'q_kernel_12', 'q_kernel_zz', 'q_kernel_training', 'q_twolayerqnn',
          'q_circuitqnn', 'q_vqc']

for rescale in scale:
    for select in selection:
        for k in kernel:
            try:
                ml_pipeline_function(dfb5, output_folder = './Outputs/', missing_method =
'row_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =

```

```
['Target'], rescaling = rescale, feature_selection = select, k_features=5, cv = 5,
quantum_algorithms = [k], reps = 2, ibm_account = 'YOUR API', quantum_backend =
'statevector_simulator')
    except Exception as e: print(e)
```

If we want to create a pipeline to benchmark different quantum algorithms with a combination of feature rescaling and feature extraction techniques, we can use the following code:

```
#!/usr/bin/python3
from ml_pipeline_function import ml_pipeline_function
import pandas as pd

# Dataset
from data.datasets import neurons_maha_soma
neuron = neurons_maha_soma()

dfb5 = neuron.head(22).copy() # Ganglion
dfb5 = pd.concat([dfb5, neuron.iloc[320:340]]) # Granule
dfb5 = pd.concat([dfb5, neuron.iloc[1493:1513]]) # Medium Spiny
dfb5 = pd.concat([dfb5, neuron.iloc[1171:1191]]) # Parachromaffin
dfb5 = pd.concat([dfb5, neuron.iloc[10031:10051]]) # Pyramidal

dfb5 = pd.concat([dfb5, neuron.iloc[2705:2725]]) # Basket
dfb5 = pd.concat([dfb5, neuron.iloc[22589:22609]]) # Bitufted
dfb5 = pd.concat([dfb5, neuron.iloc[3175:3195]]) # Chandelier
dfb5 = pd.concat([dfb5, neuron.iloc[22644:22664]]) # Double bouquet
dfb5 = pd.concat([dfb5, neuron.iloc[3199:3219]]) # Martinotti
dfb5 = pd.concat([dfb5, neuron.iloc[8260:8280]]) # Nitrergic

dfb5 = pd.concat([dfb5, neuron.iloc[2255:2275]]) # Astrocytes
dfb5 = pd.concat([dfb5, neuron.iloc[3306:3326]]) # Microglia

scale = [
    'standard_scaler',
    'minmax_scaler',
    'maxabs_scaler',
    'robust_scaler',
    'normalizer',
    'log_transformation',
    'square_root_transformation',
    'reciprocal_transformation',
    'box_cox',
    'yeo_johnson',
    'quantile_gaussian',
    'quantile_uniform',
]

extraction = [
    'pca',
    'ica',
```

```

'icawithpca',
'lda_extraction',
'random_projection',
'truncatedSVD',
'isomap',
'standard_lle',
'modified_lle',
'hessian_lle',
'ltsa_lle',
'mds',
'spectral',
'tsne',
'nca'
]

kernel = ['q_kernel_default', 'q_kernel_8', 'q_kernel_9', 'q_kernel_10',
'q_kernel_11', 'q_kernel_12', 'q_kernel_zz', 'q_kernel_training', 'q_twolayerqnn',
'q_circuitqnn', 'q_vqc']

for rescale in scale:
    for select in extraction:
        for k in kernel:
            try:
                ml_pipeline_function(df5, output_folder = './Outputs/', missing_method =
'reow_removal', test_size = 0.2, categorical = ['label_encoding'], features_label =
['Target'], rescaling = rescale, feature_extraction = extraction, number_components = 2,
cv = 5, quantum_algorithms = [k], reps = 2, ibm_account = 'YOUR API', quantum_backend =
'statevector_simulator')
            except Exception as e: print(e)

```

References

- Blacoe, W., Kashefi, E., and Lapata, M. (2013). A quantum-theoretic approach to distributional semantics. In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 847–857. Atlanta, Georgia: Association for Computational Linguistics.
- Ezhov, A. and Ventura, D. (2000). Quantum neural networks. In: *Future Directions for Intelligent Systems and Information Sciences* (ed. N. Kasabov), pp. 213–235.
- J.R. Glick, T.P. Gujarati, A.D. Corcoles et al. (2021). Covariant quantum Kernels for data with group structure. <https://arxiv.org/pdf/2105.03406.pdf>.
- Grover, L.K (1996). A fast quantum mechanical algorithm for database search. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pp. 212–219.
- Havlíček, V., Córcoles, A.D., Temme, K. et al. (2019). Supervised learning with quantum-enhanced feature spaces. *Nature* 567 (7747): 209–212.
- Liu, Z., Liu, Y., Yang, Y. et al. (2017). Subthalamic nuclei stimulation in patients with pantothenate kinase-associated neurodegeneration (PKAN). *Neuromodulation* 20 (5): 484–491.
- Lloyd, S., Mohseni, M., and Rebentrost, P. (2013). Quantum algorithms for supervised and unsupervised machine learning. <https://arxiv.org/abs/1307.0411>.
- Rebentrost, P., Mohseni, M., and Lloyd, S. (2013). Quantum support vector machine for big data classification. *Physical Review Letters, American Physical Society* 113 (13): (revised in 2014). <https://doi.org/10.1103/physrevlett.113.130503>.
- Ricks, B. and Ventura, D. (2003). Training a quantum neural network. *Neural Information Processing Systems* 1019–1026.
- Shor, P.W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In: *Proceeding of the 35th Annual Symposium on Foundations of Computer Science IEEE*, pp. 124–134.

Further Reading

- Biamonte, J., Wittek, P., Pancotti, N. et al. (2018). Quantum machine learning. *Nature* 549 (7671): 195–202.
- Bishwas, A.K., Mani, A., and Palade, V. (2018). An all-pair quantum SVM approach for big data multiclass classification. *Quantum Information Processing* 17: 282.
- Coles, P.J. (2021). Seeking quantum advantage for neural networks. *Nature Computational Science* 1: 389–390.
- Farhi, E. and Neven, H. (2018). Classification with quantum neural networks on near term processors. <https://arxiv.org/abs/1802.06002>.
- Schuld, M. (2021). Supervised quantum machine learning models are Kernel methods. <https://arXiv:2101.11020>.
- Suzuki, Y., Yano, H., Gao, Q. et al. (2020). Analysis and synthesis of feature map for Kernel-based quantum classifier. <https://arXiv:1906.10467>.
- Schuld, M. and Killoran, N. (2018). Quantum machine learning in feature Hilbert spaces. *Physical Review Letters* 122: 040504.
- Vapnik, V.N. (2000). *The Nature of Statistical Learning Theory*. Book series: Information Science and Statistics (ISS). Springer.
- Zoufal, C., Lucchi, A., and Woerner, S. (2019). Quantum generative adversarial networks for learning and loading random distributions. *NPJ Quantum Information* 5 (1): 103.

6

Machine Learning in Production



Photo by Annamária Borsos

How many artificial intelligence (AI) models that have been created have been put into production? With investment in data science teams and technologies, the number of AI projects has increased significantly and with it a number of missed opportunities to put them into production and assess their true business value. The goal of this chapter is to provide a brief introduction to several technologies that can help bring our AI models to life.

6.1 Why Use Docker Containers for Machine Learning?

Even though there are different containerization technologies, we will choose Docker to explain why containerization of machine learning applications is important. Docker is an open-source technology that allows packaging of applications into containers.

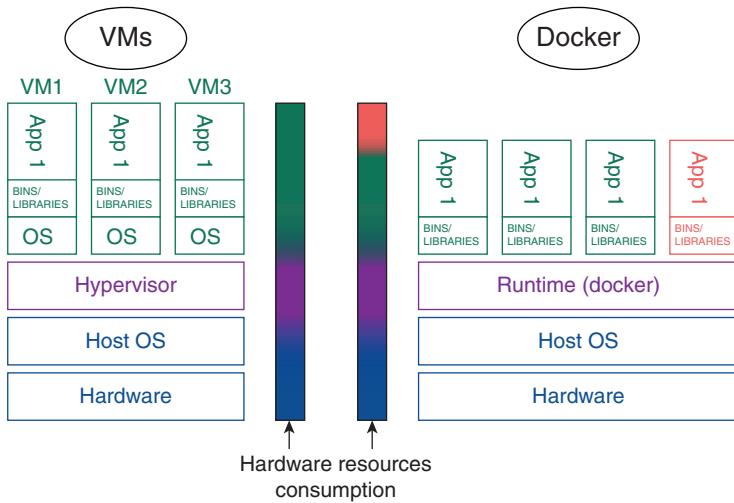
6.1.1 First Things First: The Microservices

The first thing to understand before talking about containerization is the concept of microservices. If a large application is broken down into smaller services, each of those services or small processes can be termed microservices, and they communicate with each other over a network. The microservice approach is the opposite of the monolithic approach, which can be difficult to scale. If one particular feature has some issues or crashes, all other features will experience the same. Another example is when the demand for a particular feature seriously increases and we are forced to increase resources such as the hardware not only for this particular feature but also for the entire application, generating unnecessary additional cost. This cost can be minimized if a microservice approach is taken by breaking down the application into a group of smaller services.

Each service or feature of the application is isolated in a way that we can scale or update without impacting other application features. To put machine learning into production, let us consider that the application needs to be broken down into smaller microservices such as ingestion, preparation, combination, separation, training, evaluation, inference, postprocessing, and monitoring.

6.1.2 Containerization

Microservice architecture also has its drawbacks. When developing a machine learning application in one server, we will require the same number of virtual machines (VMs) as microservices containing dependencies. Each VM will need an operating system (OS), libraries, and binaries and will consume more hardware resources such as processor, memory, and disk space even if the microservice is not running. This is where Docker comes in. If a container is not running, the remaining resources become shared resources and accessible to other containers. We do not need to add an OS in a container. Let us consider an entire solution composed of applications 1 and 2 (APP 1 and APP 2, respectively). If we want to scale out the APP 1 or add other applications as shown in the scheme below, we can be limited using VMs instead of containers by the resources available. If we decide to scale out, only APP 1 and not APP 2 (keeping only a single one), APP 2 becomes “share” of all container processes.



6.1.3 Docker and Machine Learning: Resolving the “It Works in My Machine” Problem

Creating a machine learning model that works on our computer is not really complicated. But when we work, for example, with a customer who wants to use a model that can scale and function in all types of servers across the globe, it is more challenging. After developing our model, it might run perfectly well on our laptop or server but not really on other systems such as when we move the model to the production stage or another server. Many problems can occur, including performance issues, application crashes, or poor optimization. The other challenging situation is that our machine learning model can certainly be written with one single programming language such as Python, but the application will also certainly need to interact with other applications written in other programming languages for data ingestion, data preparation, front-end, and so on. Docker allows better management of all these interactions as each microservice can be written in a different language, allowing scalability and the straightforward addition or deletion of independent services. Docker provides **reproducibility, portability, easy deployment, granular updates, lightness, and simplicity**.

When a model is ready, the data scientist’s worry is that the model will not reproduce the results of real life. Sometimes, it is not because of the model but rather the need to reproduce the whole stack. Docker allows easy reproduction of the working environment that can be used to train and run the machine learning model anywhere. Docker allows packaging of code and dependencies into containers that can be ported to different servers even with different hardware or OSs. A training model can be developed on a local machine and easily ported to external clusters with additional resources such as graphics processing units (GPUs), more memory, or powerful central processing units (CPUs). It is easy to deploy and make a model available to the globe by wrapping it into an application programming interface (API) in a container and deploying the container using technology such as OpenShift, a Kubernetes distribution. This simplicity is also a good

argument in favor of the containerization of machine learning applications, as we can automatically create containers with templates and have access to an open-source registry containing existing user-contributed containers. Docker allows developers to track the different versions of a container image, check who built a version with what platform, and roll back to previous versions. Finally, another argument is that a machine learning application can continue running even if one of its services is updating, repairing, or down. For example, to update an output message that is embedded in the entire solution, there is no need to update the entire application and to interfere with other services.

6.1.4 Quick Install and First Use of Docker

6.1.4.1 Install Docker

For the installation, we can find all the necessary documentation on Docker's website (<https://docs.docker.com/engine>).

When we create Docker containers, we need to use some tools and terminologies such as **Dockerfile**, **Docker Images**, and **Docker Hub**. Docker containers run instances of Docker images. Docker images contain all the tools, libraries, dependencies, and executable application source code necessary to run the application as a container. We can build the Docker image from common repositories or from scratch using a Dockerfile, which is a text file containing instructions on how to build Docker container images. It is a list of commands that **Docker Engine** will run. Docker Hub is the public repository of Docker images. We can think of it as a GitHub for Docker images. There is a massive number of images that have been published by Docker, Inc., individual developers, commercial software vendors, and open-source projects.

To install Docker on a MacOS machine, the simplest and fastest way is to download and install the Docker Desktop package at: <https://docs.docker.com/docker-for-mac/install>.

The Docker Desktop installation includes Docker Engine, Docker CLI client, Docker Compose, Notary, and Kubernetes. If we want to install Docker through a terminal, we need to install Homebrew (<https://brew.sh>). We can install the docker dependency with Homebrew after ensuring that we have the latest version of Homebrew dependencies:

```
brew update
brew install docker
```

Then, we need to install the docker-machine and VirtualBox dependencies because Docker uses a Linux environment natively:

```
brew install docker-machine
brew cask install virtualbox
```

We can add Docker Compose if needed later:

```
brew install docker-compose
```

Now, Docker is installed!

For Ubuntu, we can install Docker using default repositories. It is recommended to uninstall any previous Docker version before installing a new one:

```
sudo apt-get update
sudo apt-get remove docker docker-engine docker.io
sudo apt install docker.io
```

We type the following commands to run the Docker service at startup:

```
sudo systemctl start docker
sudo systemctl enable docker
```

We can check the Docker version:

```
docker --version
```

For CentOS or Red Hat, we go to <https://download.docker.com/linux/centos> and choose our version of CentOS. Then, we browse to `x86_64/stable/Packages/` and download the `.rpm` file for the Docker version we wish to install:

```
sudo yum install docker-ce-19.03.13-3.el8.x86_64.rpm
```

The `docker-ce-19.03.13-3.el8.x86_64.rpm` package can be replaced by the package we have selected.

We start Docker:

```
sudo systemctl start docker
```

We verify that Docker is correctly installed by running the **hello-world** image:

```
sudo docker run hello-world
```

The last command downloads a test image and runs it in a container.

6.1.4.2 Using Docker from the Command Line

A quick use of Docker is when we need to start, for example, a Python interpreter. We can run a Docker container directly in the command line:

```
xaviervasques$ docker run --rm -ti python:3.6 python
Python 3.6.13 (default, Apr  2 2021, 23:08:33)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We do not need to have Python installed, as Docker will automatically download the images from the Docker hub registry at <https://hub.docker.com>.

Of course, we can run any version of Python, such as 2.7:

```
docker run --rm -ti python:2.7 python
```

The command tells Docker to run a new container from the **python:2.7** image. The **-rm** flag tells Docker to remove the container once we stop the process (`ctrl+d`). The **-ti** flag means that we can interact with the container from our terminal. To see what Docker images are running on our computer, we can run the following from the command line:

```
docker images
```

To execute bash in the containers, we simply add the **bash** command:

```
docker run --rm --ti python:3.6 bash
```

Python alone may not be sufficient to create an environment to perform more complex tasks. For example, if we need an interactive environment such as Jupyter Notebooks, we can use an available public image. The command to run Jupyter Notebooks is the following:

```
docker run --rm -p 8888:8888 jupyter/scipy-notebook
```

We obtain the following output:

```

WARN: Jupyter Notebook deprecation notice https://github.com/jupyter/docker-
stacks#jupyter-notebook-deprecation-notice.
Executing the command: jupyter notebook
[I 16:45:11.911 NotebookApp] Writing notebook server cookie secret to /home/jovyan/.
local/share/jupyter/runtime/notebook_cookie_secret
/opt/conda/lib/python3.8/site-packages/jupyter_server/transutils.py:13:
FutureWarning: The alias `__()` will be deprecated. Use `__i18n__()` instead.
    warnings.warn(warn_msg, FutureWarning)
[W 2021-04-12 16:45:13.840 LabApp] 'ip' has moved from NotebookApp to ServerApp. This
config will be passed to ServerApp. Be sure to update your config before our next release.
[W 2021-04-12 16:45:13.840 LabApp] 'port' has moved from NotebookApp to ServerApp.
This config will be passed to ServerApp. Be sure to update your config before our next release.
[W 2021-04-12 16:45:13.840 LabApp] 'port' has moved from NotebookApp to ServerApp. This
config will be passed to ServerApp. Be sure to update your config before our next release.
[W 2021-04-12 16:45:13.840 LabApp] 'port' has moved from NotebookApp to ServerApp.
This config will be passed to ServerApp. Be sure to update your config before our next
release.
[I 2021-04-12 16:45:13.848 LabApp] JupyterLab extension loaded from /opt/conda/lib/
python3.8/site-packages/jupyterlab
[I 2021-04-12 16:45:13.848 LabApp] JupyterLab application directory is /opt/conda/
share/jupyter/lab
[I 16:45:13.856 NotebookApp] Serving notebooks from local directory: /home/jovyan
[I 16:45:13.856 NotebookApp] Jupyter Notebook 6.2.0 is running at:
[I 16:45:13.856 NotebookApp] http://766eac9863ee:8888/?token=6afc35687b7768490947e82d4e3876ae2fa2dc435c436cec
[I 16:45:13.856 NotebookApp] or
http://127.0.0.1:8888/?token=6afc35687b7768490947e82d4e3876ae2fa2dc435c436cec
[I 16:45:13.856 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
[C 16:45:13.863 NotebookApp]

To access the notebook, open this file in a browser:
file:///home/jovyan/.local/share/jupyter/runtime/nbserver-8-open.html
Or copy and paste one of these URLs:
http://766eac9863ee:8888/?token=6afc35687b7768490947e82d4e3876ae2fa2dc435
c436cec
or http://127.0.0.1:8888/?token=6afc35687b7768490947e82d4e3876ae2fa2dc435c436cec

```

We can access the Jupyter Notebook by copying and pasting one of the output URLs (the token from the command line). The **-p** flag tells Docker to open a port from the container to a port on the host machine. Port **8888** on the left side is the port number on the host machine, and port **8888** on the right side is the port in the container.

6.1.5 Dockerfile

All of the above is useful, but running bash in a container and installing what we need would be painful to replicate. Therefore, let us get our code into a container and create our process by building a custom Docker image with a **Dockerfile**, which tells Docker what we need in our application to run. **Dockerfile** is a text-based script of instructions.

To create a **Dockerfile**, we open a file named **Dockerfile** in our working environment. Once created, we use a simple syntax such as the following:

Command	What it does
FROM	Specifies the base image. We can browse base images on Docker Hub
WORKDIR	Changes and create the directory within the image to the specified path
RUN	Runs a command in a terminal inside the image
ADD	Specifies the files to add from our directory to a directory (creates if it does not exist) in the image
EXPOSE	This command opens a specific port number
CMD	Takes the argument for running an application
ENV	Specifies environment variables specific to services
COPY	Adds files

Let us create a simple **Dockerfile**:

```
FROM jupyter/scipy-notebook

# install build utilities
RUN pip install joblib

# check our python environment
RUN python3 -version
RUN pip3 -version

# set the working directory for containers
WORKDIR /usr/src/my-app-name

# Copy all the necessary files to the image (COPY <src> ... <dest>)
COPY train.py ./train.py
COPY inference.py ./inference.py

# Run python scripts
RUN python3 train.py
RUN python3 inference.py
```

Here, we start with the **jupyter/scipy-notebook** image, which is a Jupyter Notebook scientific Python stack including popular packages from the scientific Python ecosystem such as popular Python deep learning libraries. We run **pip install joblib** to install **joblib**. We then check our Python environment and set the working directory for containers. We copy the **train.py** and **inference.py** scripts into the image. Then, we run the scripts. We could also run the scripts as follows:

```
# Run python scripts
CMD ["python3","train.py"]
CMD ["python3","inference.py"]
```

Let us see another example:

```
FROM python:3.7

# install build utilities
RUN pip install joblib pandas scikit-learn

COPY train.py ./train.py

RUN python3 train.py
```

To build the image, we need to run the following command:

```
docker build -t docker-ml-model -f Dockerfile .
```

To run a Custom Docker image, we run the following:

```
docker run docker-ml-model
```

6.1.6 Build and Run a Docker Container for Your Machine Learning Model

The idea of this section is to perform a rapid and easy build of a Docker container with a simple machine learning model and then run it. To start building a Docker container for a machine learning model, let us consider three files:

- **Dockerfile**
- **train.py**
- **inference.py**

All files can be found on GitHub (<https://github.com/xaviervasques/EEG-letters>). The file **train.py** is a Python script that ingests and normalizes EEG data in a .csv file (train.csv) and trains two models to classify the data (using scikit-learn). The script saves two models: linear discriminant analysis (LDA) (clf_lda) and neural network multilayer perceptron (clf_NN).

```
#!/usr/bin/python3
# train.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import dump
from sklearn import preprocessing
```

```

def train():

    # Load, read and normalize training data
    training = "./train.csv"
    data_train = pd.read_csv(training)

    y_train = data_train['# Letter'].values
    X_train = data_train.drop(data_train.loc[:, 'Line':'# Letter'].columns, axis = 1)

    print("Shape of the training data")
    print(X_train.shape)
    print(y_train.shape)

    # Data normalization (0,1)
    X_train = preprocessing.normalize(X_train, norm='l2')

    # Models training

    # Linear Discriminant Analysis (Default parameters)
    clf_lda = LinearDiscriminantAnalysis()
    clf_lda.fit(X_train, y_train)

    # Save model
    from joblib import dump
    dump(clf_lda, 'Inference_lda.joblib')

    # Neural Networks multi-layer perceptron (MLP) algorithm
    clf_NN = MLPClassifier(solver='adam', activation='relu', alpha=0.0001,
                           hidden_layer_sizes=(500,), random_state=0, max_iter=1000)
    clf_NN.fit(X_train, y_train)

    # Save model
    from joblib import dump, load
    dump(clf_NN, 'Inference_NN.joblib')

if __name__ == '__main__':
    train()

```

The **inference.py** script will be called to perform batch inference by loading the two models that have been created. The application will normalize new EEG data coming from a .csv file (test.csv), perform inference on the dataset, and print the classification accuracy and predictions:

```

#!/usr/bin/python3
# inference.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

```

```

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import dump
from sklearn import preprocessing

def inference():

    # Load, read and normalize training data
    testing = "/Users/LRENC/Desktop/Kubernetes/code/test.csv"
    data_test = pd.read_csv(testing)

    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)

    print("Shape of the test data")
    print(X_test.shape)
    print(y_test.shape)

    # Data normalization (0,1)
    X_test = preprocessing.normalize(X_test, norm='l2')

    # Models training

    # Run model
    clf_lda = load('Inference_lda.joblib')
    print("LDA score and classification:")
    print(clf_lda.score(X_test, y_test))
    print(clf_lda.predict(X_test))

    # Run model
    clf_nn = load('Inference_NN.joblib')
    print("NN score and classification:")
    print(clf_nn.score(X_test, y_test))
    print(clf_nn.predict(X_test))

if __name__ == '__main__':
    inference()

```

Let us create a simple **Dockerfile** with the **jupyter/scipy-notebook** image as our base image. We need to install **joblib** to allow serialization and deserialization of our trained model. We copy the train.csv, test.csv, train.py, and inference.py files into the image. Then, we run **train.py**, which will fit and serialize the machine learning models as part of our image build process and provide several advantages such as the ability to debug at the beginning of the process, use Docker Image ID for keeping track, or use different versions.

```

FROM jupyter/scipy-notebook

RUN pip install joblib

```

```
COPY train.csv ./train.csv
COPY test.csv ./test.csv

COPY train.py ./train.py
COPY inference.py ./inference.py

RUN python3 train.py
```

To build the image, we run the following command in a terminal:

```
docker build -t docker-ml-model -f Dockerfile .
```

It is now time to perform the inference on new data (test.csv):

```
docker run docker-ml-model python3 inference.py
```

The output is the following:

```
xaviervasques@LRENCS:~/code$ docker run docker-ml-model python3 inference.py
Linux-5.10.25-linuxkit-x86_64-with-glibc2.10
Python 3.8.8 | packaged by conda-forge | (default, Feb 20 2021, 16:22:27)
[GCC 9.3.0]
NumPy 1.20.1
SciPy 1.6.1
Shape of the test data
(1300, 160)
(1300,)
LDA score and classification:
0.6915384615384615
[ 0  0  0 ... 25 25 25]
NN score and classification:
0.6615384615384615
[ 0  0  0 ... 25 25 24]
```

We can do some other things to improve our containerization experience. For example, we can bind a host directory in the container using WORKDIR in the Dockerfile:

```
FROM jupyter/scipy-notebook

WORKDIR /mydata

RUN pip install joblib

COPY train.csv ./train.csv
COPY test.csv ./test.csv

COPY train.py ./train.py
COPY inference.py ./inference.py

RUN python3 train.py
```

In **inference.py**, we can decide for example to save an **output.csv** file with the **X_test** data in it:

```
#!/usr/bin/python3
# inference.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
```

```
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import load
from sklearn import preprocessing

def inference():

    dirpath = os.getcwd()
    print("dirpath = ", dirpath, "\n")

    output_path = os.path.join(dirpath, 'output.csv')
    print(output_path, "\n")

    # Load, read and normalize training data
    testing = "test.csv"
    data_test = pd.read_csv(testing)

    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)

    print("Shape of the test data")
    print(X_test.shape)
    print(y_test.shape)

    # Data normalization (0,1)
    X_test = preprocessing.normalize(X_test, norm='l2')

    # Models training

    # Run model
    clf_lda = load('Inference_lda.joblib')
    print("LDA score and classification:")
    print(clf_lda.score(X_test, y_test))
    print(clf_lda.predict(X_test))

    # Run model
    clf_nn = load('Inference_NN.joblib')
    print("NN score and classification:")
    print(clf_nn.score(X_test, y_test))
    print(clf_nn.predict(X_test))

    #X_test.to_csv(output_path)
    print(output_path)
    pd.DataFrame(X_test).to_csv(output_path)

if __name__ == '__main__':
    inference()
```

When we build and run the code above, we should be able to see the **output.csv** file in **/mydata**:

```
xavierasques:code LRENC$ docker run -v /Users/LRENC/Desktop/data docker-ml-model python3 inference.py
Linux-5.10.25-linuxkit-x86_64-with-glibc2.10
Python 3.8.8 | packaged by conda-forge | (default, Feb 20 2021, 16:22:27)
[GCC 9.3.0]
NumPy 1.20.1
SciPy 1.6.1
dirpath =  /mydata
/mydata/output.csv

Shape of the test data
(1300, 160)
(1300,)
LDA score and classification:
0.6915384615384615
[ 0  0  0 ... 25 25 25]
NN score and classification:
0.6615384615384615
[ 0  0  0 ... 25 25 24]
/mydata/output.csv
```

We can also add the **VOLUME** instruction in the Dockerfile, resulting in an image that will create a new mount point:

```
FROM jupyter/scipy-notebook

VOLUME /Users/Xavi/Desktop/code/data

RUN pip install joblib

COPY train.csv ./train.csv
COPY test.csv ./test.csv

COPY train.py ./train.py
COPY inference.py ./inference.py

RUN python3 train.py
```

With the name that we specify, the **VOLUME** instruction creates a mount point that is tagged as holding an externally mounted volume from the native host or other containers that hold the data we want to process.

For future development, it could be necessary to set environment variables from the beginning, only once at the build time, for persisting the trained model and perhaps adding additional data or metadata to a specific location. The advantage of setting environment variables is to avoid the hard code of the necessary paths all over our code and to better share our work with others on an agreed-upon directory structure.

Let us take another example, with a new Dockerfile:

```
FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE_LDA=clf_lda.joblib
ENV MODEL_FILE_NN=clf_nn.joblib

RUN pip install joblib

COPY train.csv ./train.csv
COPY test.csv ./test.csv
```

```
COPY train.py ./train.py
COPY inference.py ./inference.py
```

RUN python3 train.py

We need to add the environment variables to the **train.py** and **inference.py** files:

```
#!/usr/bin/python3
# train.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import dump
from sklearn import preprocessing

def train():

    # Load directory paths for persisting model

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

    # Load, read and normalize training data
    training = "./train.csv"
    data_train = pd.read_csv(training)

    y_train = data_train['# Letter'].values
    X_train = data_train.drop(data_train.loc[:, 'Line':'# Letter'].columns, axis = 1)

    print("Shape of the training data")
    print(X_train.shape)
    print(y_train.shape)

    # Data normalization (0,1)
    X_train = preprocessing.normalize(X_train, norm='l2')

    # Models training

    # Linear Discriminant Analysis (Default parameters)
```

```

clf_lda = LinearDiscriminantAnalysis()
clf_lda.fit(X_train, y_train)

# Save model
from joblib import dump
dump(clf_lda, MODEL_PATH_LDA)

# Neural Networks multi-layer perceptron (MLP) algorithm
clf_NN = MLPClassifier(solver='adam', activation='relu', alpha=0.0001,
hidden_layer_sizes=(500,), random_state=0, max_iter=1000)
clf_NN.fit(X_train, y_train)

# Record model
from joblib import dump, load
dump(clf_NN, MODEL_PATH_NN)

if __name__ == '__main__':
    train()

```

We also need to add the environment variables to **inference.py**:

```

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import load
from sklearn import preprocessing

def inference():

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

    # Load, read and normalize training data
    testing = "test.csv"
    data_test = pd.read_csv(testing)

    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)

```

```

print("Shape of the test data")
print(X_test.shape)
print(y_test.shape)

# Data normalization (0,1)
X_test = preprocessing.normalize(X_test, norm='l2')

# Models training

# Run model
print(MODEL_PATH_LDA)
clf_lda = load(MODEL_PATH_LDA)
print("LDA score and classification:")
print(clf_lda.score(X_test, y_test))
print(clf_lda.predict(X_test))

# Run model
clf_nn = load(MODEL_PATH_NN)
print("NN score and classification:")
print(clf_nn.score(X_test, y_test))
print(clf_nn.predict(X_test))

if __name__ == '__main__':
    inference()

```

The goal is to produce fast and easy steps to build a Docker container with a simple machine learning model. Building is as simple as executing **docker build -t my-docker-image**.

From this step, we can begin the deployment of our models, which will be much simpler and remove the fear of publishing and scaling the machine learning model. The next step is to produce a workflow with a continuous integration/continuous delivery (CI/CD) tool such as Jenkins. With this approach, it will be possible to build and serve a docker container anywhere and expose a REST API so that external stakeholders can use it. If we are training a deep learning model that needs high computational needs, we can move the containers to high-performance computing servers or any platform of choice such as on-premises or private or public cloud. The idea is that we can not only scale our model but also create resilient deployment, as we can scale the container across regions or availability zones.

I hope that the great simplicity and flexibility that containers provide are clear. By containerizing a machine or deep learning application, we can make it visible to the world. The next step is to deploy it in the cloud and expose it. At certain times, we might need to orchestrate, monitor, and scale the containers to serve millions of users with the help of technologies such as Red Hat OpenShift, a Kubernetes distribution.

6.2 Machine Learning Prediction in Real Time Using Docker and Python REST APIs with Flask

The idea of this section is to perform a rapid and easy build of a Docker container to perform online inference with trained machine learning models using Python APIs with Flask.

Batch inference is excellent when we have time to compute our predictions. Let us imagine we need real-time predictions. In this case, batch inference is not suitable; we need online inference. Many applications would not work or would not be very useful without online predictions such as autonomous vehicles, fraud detection, high-frequency trading, applications based on localization data, object recognition and tracking, or brain-computer interfaces. Sometimes, the prediction needs to be provided in milliseconds.

To understand this concept, we will implement online inferences (LDA and multilayer perceptron neural network models) with Docker and Flask-RESTful.

To start, let us consider the following files:

- **Dockerfile**
- **train.py**
- **api.py**
- **requirements.txt**
- **train.csv**
- **test.json**

The file **train.py** is a Python script that ingests and normalizes electroencephalography (EEG) data and trains two models to classify the data. The **Dockerfile** will be used to build our Docker image, requirements.txt (flask, flask-restful, joblib) is for the Python dependencies, and api.py is the script that will be called to perform the online inference using REST APIs. The file **train.csv** contains the data used to train our models, and **test.json** is a file containing new EEG data that will be used with our inference models. All files can be found on GitHub.

6.2.1 Flask-RESTful APIs

The first step in building APIs is to consider the data we desire to process, how we want to process it, and what output we desire with our APIs. In our example, we will use the **test.json** file, which contains 1300 rows of EEG data with 160 features each (columns). We want our APIs to perform the following:

- **API 1:** We will give a row number to the API, which will extract for us the data from the selected row and print it.
- **API 2:** We will give a row number to the API, which will extract the selected row, inject the new data into the models, and retrieve the classification prediction (letter variable in the data).
- **API 3:** We will ask the API to take all the data in the **test.json** file and instantly print the classification score of the models.

In the end, we want to access these processes by making an HTTP request.

Let us view the **api.py** file:

```
# We now need the json library so we can load and export json data
import json
import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import load
from sklearn import preprocessing

from flask import Flask

# Set environment variables
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

# Loading LDA model
print("Loading model from: {}".format(MODEL_PATH_LDA))
inference_lda = load(MODEL_PATH_LDA)

# loading Neural Network model
print("Loading model from: {}".format(MODEL_PATH_NN))
inference_NN = load(MODEL_PATH_NN)
```

```

# Creation of the Flask app
app = Flask(__name__)

# Flask route so that we can serve HTTP traffic on that route
@app.route('/line/<Line>')
# Get data from json and return the requested row defined by the variable Line
def line(Line):
    with open('./test.json', 'r') as jsonfile:
        file_data = json.loads(jsonfile.read())
    # We can then find the data for the requested row and send it back as json
    return json.dumps(file_data[Line])

# Flask route so that we can serve HTTP traffic on that route
@app.route('/prediction/<int:Line>', methods=['POST', 'GET'])
# Return prediction for both Neural Network and LDA inference model with the requested
row as input
def prediction(Line):
    data = pd.read_json('./test.json')
    data_test = data.transpose()
    X = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)
    X_test = X.iloc[Line,:].values.reshape(1, -1)

    clf_lda = load(MODEL_PATH_LDA)
    prediction_lda = clf_lda.predict(X_test)

    clf_nn = load(MODEL_PATH_NN)
    prediction_nn = clf_nn.predict(X_test)

    return {'prediction LDA': int(prediction_lda), 'prediction Neural Network': int(prediction_nn)}

# Flask route so that we can serve HTTP traffic on that route
@app.route('/score', methods=['POST', 'GET'])
# Return classification score for both Neural Network and LDA inference model from the
all dataset provided
def score():

    data = pd.read_json('./test.json')
    data_test = data.transpose()
    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)

    clf_lda = load(MODEL_PATH_LDA)
    score_lda = clf_lda.score(X_test, y_test)

    clf_nn = load(MODEL_PATH_NN)
    score_nn = clf_nn.score(X_test, y_test)

    return {'Score LDA': score_lda, 'Score Neural Network': score_nn}

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0')

```

The first step, after importing dependencies including the open-source web microframework Flask, is to set the environment variables that are written in the Dockerfile. We also need to load our linear discriminant analysis and multilayer perceptron neural network serialized models. We create our Flask application by writing `app = Flask(__name__)`. Then, we create our three Flask routes so that we can serve HTTP traffic on that route:

- `http://0.0.0.0:5000/line/250`: Obtain data from `test.json` and return the requested row defined by the variable `Line` (in this example, we want to extract the data of row number 250).
- `http://0.0.0.0:5000/prediction/51`: Return classification predictions from models trained by both LDA and NN by injecting the requested data (in this example, we want to inject the data of row number 51).
- `http://0.0.0.0:5000/score`: Return classification score for both the neural network and LDA inference models on all the available data (`test.json`).

The Flask routes allow us to request what we need from the API by adding the name of our procedure (`/line/<Line>`, `/prediction/<int:Line>`, `/score`) to the URL (`http://0.0.0.0:5000`). Whatever data we add, `api.py` will always return the output we request.

6.2.2 Machine Learning Models

The file `train.py` is a Python script that ingests and normalizes EEG data in a .csv file (`train.csv`) and trains two models to classify the data (using scikit-learn). The script saves two models: LDA (`clf_lda`) and neural network multilayer perceptron (`clf_NN`):

```
#!/usr/bin/python3
# tain.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import dump
from sklearn import preprocessing

def train():

    # Load directory paths for persisting model

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

    # Load, read and normalize training data
    training = "./train.csv"
    data_train = pd.read_csv(training)
```

```

y_train = data_train['# Letter'].values
X_train = data_train.drop(data_train.loc[:, 'Line':'# Letter'].columns, axis = 1)

print("Shape of the training data")
print(X_train.shape)
print(y_train.shape)

# Data normalization (0,1)
X_train = preprocessing.normalize(X_train, norm='l2')

# Models training

# Linear Discriminant Analysis (Default parameters)
clf_lda = LinearDiscriminantAnalysis()
clf_lda.fit(X_train, y_train)

# Serialize model
from joblib import dump
dump(clf_lda, MODEL_PATH_LDA)

# Neural Networks multi-layer perceptron (MLP) algorithm
clf_NN = MLPClassifier(solver='adam', activation='relu', alpha=0.0001,
hidden_layer_sizes=(500,), random_state=0, max_iter=1000)
clf_NN.fit(X_train, y_train)

# Serialize model
from joblib import dump, load
dump(clf_NN, MODEL_PATH_NN)

if __name__ == '__main__':
    train()

```

6.2.3 Docker Image for the Online Inference

We now have all we need to build our Docker image. To start, we need our **Dockerfile** with the **jupyter/scipy-notebook** image as our base image. We also need to set our environment variables and install joblib to allow serialization and deserialization of our trained models and Flask (**requirements.txt**). We copy the **train.csv**, **test.json**, **train.py**, and **api.py** files into the image. Then, we run **train.py**, which will fit and serialize the machine learning models as part of our image-building process.

Here is the code:

```

FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE_LDA=clf_lda.joblib
ENV MODEL_FILE_NN=clf_nn.joblib

COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY train.csv ./train.csv

```

```
COPY test.json ./test.json
COPY train.py ./train.py
COPY api.py ./api.py

RUN python3 train.py
```

To build this image, we run the following command:

```
docker build -t my-docker-api -f Dockerfile .
```

We obtain the following output:

```
LRENc@xaviervasques Online_Inference % docker build -t docker-api -f Dockerfile .
[+] Building 17.9s (14/14) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 554B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/jupyter/scipy-notebook:latest
=> [1/9] FROM docker.io/jupyter/scipy-notebook
=> [internal] load build context
=> => transferring context: 4.76kB
=> CACHED [2/9] RUN mkdir my-model
=> CACHED [3/9] COPY requirements.txt ./requirements.txt
=> CACHED [4/9] RUN pip install -r requirements.txt
=> CACHED [5/9] COPY train.csv ./train.csv
=> CACHED [6/9] COPY test.json ./test.json
=> CACHED [7/9] COPY train.py ./train.py
=> [8/9] COPY api.py ./api.py
=> [9/9] RUN python3 train.py
=> exporting to image
=> => exporting layers
=> => writing image sha256:44c268b60b1eb9dc966afc5bdc4eb713d711f8fcc863dbc4069f28a52439e616
=> => naming to docker.io/library/docker-api
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
LRENc@xaviervasques Online_Inference %
```

6.2.4 Running Docker Online Inference

The goal now is to run our online inference, meaning that each time a client issues a POST request to the /line/<Line>, /prediction/<Line>, /score endpoints, we will show the requested data (row), predict the class of the data we inject using our pre-trained models, and present the score of our pre-trained models using all the available data. To launch the web server, we will run a Docker container and run the **api.py** script:

```
docker run -it -p 5000:5000 my-docker-api python3 api.py
```

The **-p** flag exposes port 5000 in the container to port 5000 on our host machine. The **-it** flag allows us to see the logs from the container, and we run **python3 api.py** in the **my-api** image.

The output is the following:

```
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
[LRENc@xaviervasques Online_Inference % docker run -it -p 5000:5000 docker-api python3 api.py
Loading model from: /home/jovyan/my-model/clf_lda.joblib
Loading model from: /home/jovyan/my-model/clf_nn.joblib
 * Serving Flask app "api" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
Loading model from: /home/jovyan/my-model/clf_lda.joblib
Loading model from: /home/jovyan/my-model/clf_nn.joblib
 * Debugger is active!
 * Debugger PIN: 126-303-090
]
```

We are running on `http://0.0.0.0:5000/`, and we can now use our web browser or the `curl` command to issue a POST request to the IP address.

If we type

```
curl http://0.0.0.0:5000/line/232
```

we will get row number 232 extracted from our data (`test.json`):

```
[LRENC@xaviervasques Online_Inference % curl http://0.0.0.0:5000/line/232
{"Line": "232", "# Letter": "4", "Cz/theta": "0.249", "Cz/alpha": "0.246", "Cz/betaL": "0.094", "Cz/betaH": "0.08699999999999998", "Cz/gamma": "0.011", "Fz/theta": "0.71", "Fz/alpha": "0.181", "Fz/betaL": "0.038", "Fz/betaH": "0.037", "Fz/gamma": "0.02", "Fp1/theta": "0.89", "Fp1/alpha": "1.887", "Fp1/betaL": "0.273", "Fp1/betaH": "0.145", "Fp1/gamma": "0.044", "F7/theta": "1.331", "F7/alpha": "1.947", "F7/betaL": "0.35", "F7/betaH": "0.117", "F7/gamma": "0.044", "F3/theta": "0.403", "F3/alpha": "0.333", "F3/betaL": "0.05", "F3/betaH": "0.043", "F3/gamma": "0.017", "FC1/theta": "0.138", "FC1/alpha": "0.028", "FC1/betaL": "0.012", "FC1/betaH": "0.007", "FC1/gamma": "0.003", "C3/theta": "0.404", "C3/alpha": "0.312", "C3/betaL": "0.109", "C3/betaH": "0.118", "C3/gamma": "0.031", "FC5/theta": "0.288", "FC5/alpha": "0.638", "FC5/betaL": "0.152", "FC5/betaH": "0.032", "FC5/gamma": "0.022", "FT9/theta": "1.15", "FT9/alpha": "1.545", "FT9/betaL": "0.182", "FT9/betaH": "0.284", "FT9/gamma": "0.111", "TP9/theta": "1.77", "TP9/alpha": "3.312", "TP9/betaL": "0.422", "TP9/betaH": "0.324", "TP9/gamma": "0.079", "CP5/theta": "0.811", "CP5/alpha": "2.658", "CP5/betaL": "0.36", "CP5/betaH": "0.257", "CP5/gamma": "0.065", "CP1/theta": "0.526", "CP1/alpha": "2.13", "CP1/betaL": "0.179", "CP1/betaH": "0.155", "CP1/gamma": "0.05", "P3/theta": "11.99", "P3/alpha": "0.889", "P3/betaL": "0.445", "P3/betaH": "0.383", "P3/gamma": "0.076", "P7/theta": "1.985", "P7/alpha": "12.245", "P7/betaL": "0.781", "P7/betaH": "0.632", "P7/gamma": "0.125", "O1/theta": "2.034", "O1/alpha": "36.23", "O1/betaL": "1.542", "O1/betaH": "0.966", "O1/gamma": "0.193", "Pz/theta": "1.88", "Pz/alpha": "9.173", "Pz/betaL": "1.518", "Pz/betaH": "8.479", "Pz/gamma": "0.555", "Oz/theta": "1.119", "Oz/alpha": "22.548", "Oz/betaL": "0.749", "Oz/betaH": "0.423", "Oz/gamma": "0.119", "O2/alpha": "28.502", "O2/betaL": "1.007", "O2/betaH": "0.74", "O2/gamma": "0.171", "P8/theta": "1.401", "P8/alpha": "66.183", "P8/betaL": "0.965", "P8/betaH": "1.852", "P8/gamma": "0.269", "P4/theta": "1.501", "P4/alpha": "35.674", "P4/betaL": "1.257", "P4/betaH": "1.07", "P4/gamma": "0.158", "CP2/theta": "0.775", "CP2/alpha": "4.684", "CP2/betaL": "0.889", "CP2/betaH": "5.785", "CP2/gamma": "1.296", "CP6/theta": "1.961", "CP6/alpha": "25.088", "CP6/betaL": "0.592", "CP6/betaH": "0.605", "CP6/gamma": "0.217", "TP10/theta": "1.952", "TP10/alpha": "12.045", "TP10/betaL": "0.802", "TP10/betaH": "0.797", "TP10/gamma": "0.098", "T8/theta": "1.959", "T8/alpha": "6.875", "T8/betaL": "0.688", "T8/betaH": "0.379", "T8/gamma": "0.125", "FT10/theta": "2.542", "FT10/alpha": "2.529", "FT10/betaL": "0.533", "FT10/betaH": "0.129", "FT10/gamma": "0.087", "FC6/theta": "0.394", "FC6/alpha": "2.432", "FC6/betaL": "0.514", "FC6/betaH": "1.044", "FC6/gamma": "0.105", "C4/theta": "0.811", "C4/alpha": "3.986", "C4/betaL": "0.215", "C4/betaH": "0.167", "C4/gamma": "0.049", "F4/theta": "0.796", "F4/alpha": "2.529", "F4/betaL": "0.895", "F4/betaH": "1.845", "F4/gamma": "0.158", "F4/alpha": "0.7209999999999999", "F4/betaL": "0.276", "F4/betaH": "0.864", "F4/gamma": "0.035", "F8/theta": "0.798", "F8/alpha": "1.231", "F8/betaL": "0.3", "F8/betaH": "0.117", "F8/gamma": "0.064", "Fp2/theta": "1.533", "Fp2/alpha": "1.392", "Fp2/betaL": "0.27", "Fp2/betaH": "0.183", "Fp2/gamma": "0.101"}]
[LRENC@xaviervasques Online_Inference % ]
```

If we type the curl command

```
curl http://0.0.0.0:5000/prediction/232
```

we will see the following output:

```
[LRENC@xaviervasques Online_Inference % curl http://0.0.0.0:5000/prediction/232
{
  "prediction LDA": 21,
  "prediction Neural Network": 8
}
[LRENC@xaviervasques Online_Inference % ]
```

The above output means that the LDA model has classified the provided data (row 232) as letter 21 (U), while the multilayer perceptron neural network has classified the data as letter 8 (H). The two models do not agree.

If we type

```
curl http://0.0.0.0:5000/score
```

we will see the score of our models on the entire dataset:

```
[LRENC@xaviervasques Online_Inference % curl http://0.0.0.0:5000/score
{
  "Score LDA": 0.17846153846153845,
  "Score Neural Network": 0.5969230769230769
}
[LRENC@xaviervasques Online_Inference % ]
```

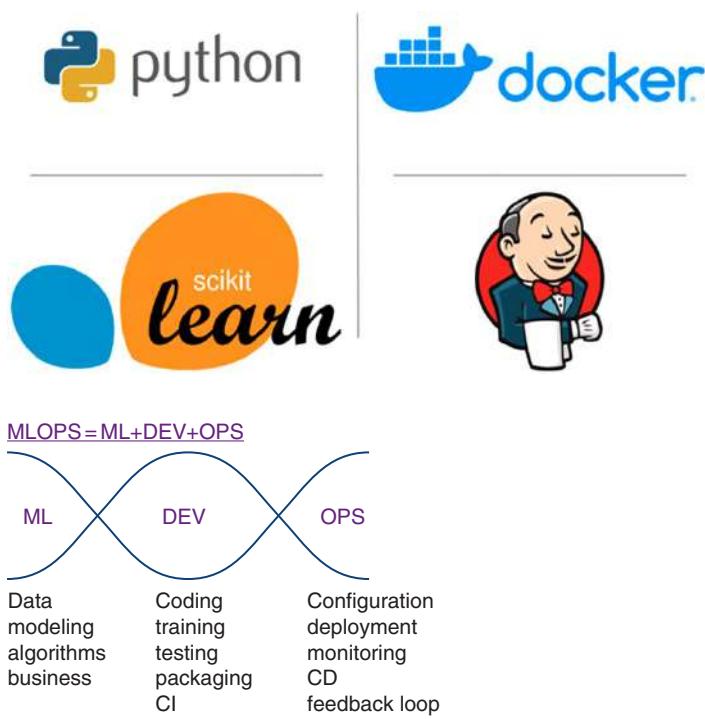
As we can see, we should trust the multilayer perceptron neural network more with its accuracy score of 0.59, even though the score is not so high. There is some work to do to improve the accuracy!

I hope the simplicity of containerizing machine learning and deep learning (ML/DL) applications using Docker and Flask to perform online inference is clear. This is an essential step when we want to put our models into production. Of course, this

example is a simple view, as we need to take into account many more aspects such as the network, security, monitoring, infrastructure, and orchestration or add a database to store the data instead of using a .json file.

6.3 From DevOps to MLOPS: Integrate Machine Learning Models Using Jenkins and Docker

How many AI models have been put into production in enterprises? With investment in data science teams and technologies, the number of AI projects has increased significantly and with it the number of missed opportunities to put them into production and assess their real business value. One of the solutions is MLOPS, which delivers the capabilities to bring data science and information technology (IT) operations together to deploy, monitor, and manage ML/DL models in production. Continuous integration (CI) and continuous delivery (CD), known as the CI/CD pipeline, embody a culture with agile operating principles and practices for DevOps teams that allows software development teams to change code more frequently and reliably or data scientists to continuously test the models for accuracy. CI/CD is a way to focus on business requirements such as improved model accuracy, automated deployment steps, or code quality. Continuous integration is a set of practices that drive development teams to continuously implement small changes and check in code to version-control repositories. Today, data scientists and IT operations have different platforms at their disposal (on-premises, private and public cloud, multi-cloud, and so on) and tools that need to be addressed by an automatic integration and validation mechanism to allow building, packaging, and testing of applications with agility. Continuous delivery steps in when continuous integration ends by automating the delivery of applications to selected platforms.



The objective of this section is to integrate machine learning models with DevOps using Jenkins and Docker. There are many advantages to using Jenkins and Docker for ML/DL. For example, when we train a machine learning model, it is necessary to continuously test the models for accuracy. This task can be fully automated using Jenkins. When we work on a data science project, we usually spend some time increasing model accuracy and then, when we are satisfied, we deploy the application to production, serving it as an API. Let us say our model accuracy is 85%. After a few days or weeks, we decide to tune some hyperparameters and add some more data in order to improve the model accuracy. Then, we plan to deploy it in production and to do this, we need to spend some effort to build, test, and deploy the model again, which can create

considerable work depending on the context and environments. This is where the open-source automation server, Jenkins, comes in. Jenkins provides a continuous integration and continuous delivery (CI/CD) system with hundreds of plug-ins to build, deploy, and automate software projects. There are several advantages of using Jenkins. It is easy to install and configure, it has an important community, it contains hundreds of plug-ins, and it can distribute work across different environments. Jenkins has one objective: to spend less time on deployment and more time on code quality. Jenkins allows us to create Jobs, which are the nucleus of the build process in Jenkins. For example, we can create Jobs to test our data science project with different tasks. Jenkins also offers a suite of plug-ins, Jenkins Pipeline, that supports CI/CD. They can be both declarative and scripted pipelines.

The declarative pipeline is a feature that supports the pipeline as a code concept and is a more recent feature. It provides richer syntactical features over scripted pipeline syntax and makes writing and reading pipeline code easier. Using the scripting way, the pipeline is written on the Jenkins user interface instance instead of writing it on a file.

In this section, we will see how to integrate a machine learning model (linear discriminant analysis and multilayer perceptron neural network) trained on EEG data using Jenkins and Docker.

To learn these concepts, let us consider the following files:

- **Dockerfile**
- **train-lda.py**
- **train-nn.py**
- **train-auto-nn.py**
- **requirements.txt**
- **train.csv**
- **test.csv**

The **train-lda.py** and **train-nn.py** files are Python scripts that ingest and normalize EEG data, train two models to classify the data, and test the model. The **Dockerfile** will be used to build our Docker image, and **requirements.txt** (*joblib*) is for the Python dependencies. The file **train-auto-nn.py** is a Python script that tweaks the neural network model with different parameters. The file **train.csv** contains the data used to train our models, and **test.csv** is a file containing new EEG data that will be used with our inference models.

All files can be found on GitHub at <https://github.com/xaviervasques/Jenkins>.

6.3.1 Jenkins Installation

With Jenkins, we will run a container image that has all the necessary requirements installed for training our models. For that, we will create a job chain using a build pipeline plug-in in Jenkins.

First, we need to install Jenkins. We will install the long-term support release.

On Red Hat or CentOS, we can type the following commands:

```
sudo wget -O /etc/yum.repos.d/jenkins.repo \
  https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key
sudo yum upgrade
sudo yum install jenkins java-1.8.0-openjdk-devel
```

On Ubuntu, we can type the following commands to install Jenkins:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > \
/etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

Jenkins requires Java. We can install the Open Java Development Kit (OpenJDK). We can see all the needed information to install Jenkins here: <https://www.jenkins.io/doc/book/installing>.

Jenkins can be installed on many distributions (Linux, macOS, Windows) and deployed on a private or public cloud such as IBM Cloud or others. We can use different commands to start some Jenkins services such as the following:

Register the Jenkins service:

```
sudo systemctl daemon-reload
```

Start the Jenkins service:

```
sudo systemctl start jenkins
```

Check the status of the Jenkins service:

```
sudo systemctl status jenkins
```

If everything has been set up correctly, we should see an output similar to this:

```
● jenkins.service - LSB: Start Jenkins at boot time
   Loaded: loaded (/etc/init.d/jenkins; generated)
   Active: active (exited) since Sun 2021-05-02 16:27:35 CEST; 2min 45s ago
     Docs: man:systemd-sysv-generator(8)
   Process: 1429 ExecStart=/etc/init.d/jenkins start (code=exited, status=0/SUCCESS)

mai 02 16:27:34 xavi systemd[1]: Starting LSB: Start Jenkins at boot time...
mai 02 16:27:34 xavi jenkins[1429]: Correct java version found
mai 02 16:27:34 xavi jenkins[1429]: * Starting Jenkins Automation Server jenkins
mai 02 16:27:34 xavi su[1503]: (to jenkins) root on none
mai 02 16:27:34 xavi su[1503]: pam_unix(su-l:session): session opened for user jenkins by (uid=0)
mai 02 16:27:34 xavi su[1503]: pam_unix(su-l:session): session closed for user jenkins
mai 02 16:27:35 xavi jenkins[1429]: ...done.
mai 02 16:27:35 xavi systemd[1]: Started LSB: Start Jenkins at boot time.
```

Jenkins uses port 8080. We can open it using ufw:

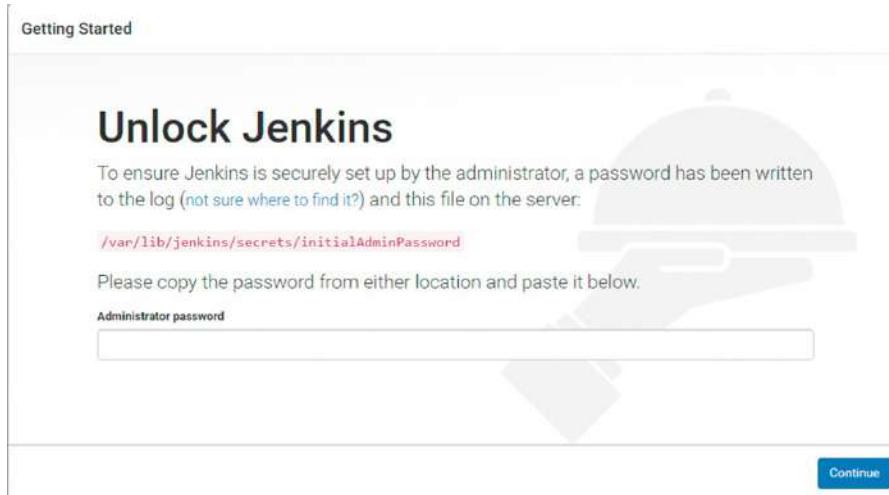
```
sudo ufw allow 8080
```

We can check the status to confirm the new rules:

```
sudo ufw status
```

To launch Jenkins, we obtain the IP address of our server by typing hostname -I and launch our browser by entering our IP and port: 192.168.1.107:8080.

We should see something similar to the following:



In a terminal, we can use the cat command to display the password:

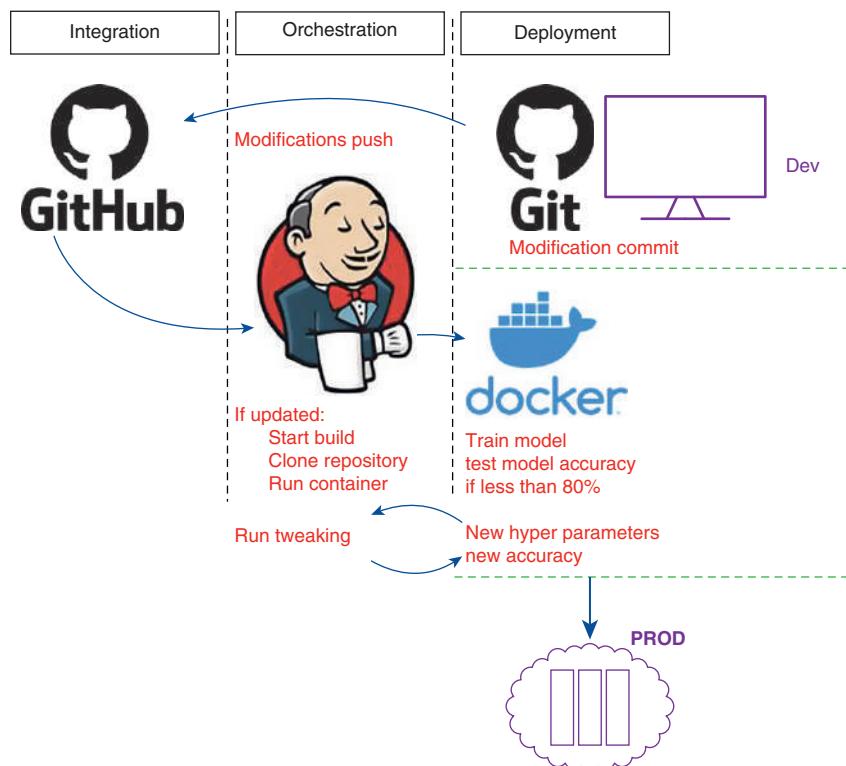
```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

We copy the password and paste it into *Administrator password* text box and click *continue*. Then, we follow some simple steps to configure the environment.

6.3.2 Scenario Implementation

Let us say we need to train our model regularly. In this case, it is recommended to wrap the process in Jenkins in order to avoid manual work and make the code much easier to maintain and improve. In this section, we will describe two scenarios:

- **Scenario 1:** We will clone a GitHub repository automatically when someone updates the machine learning code or provides additional data. Jenkins will then automatically start the training of a model and provide the classification accuracy, checking whether the accuracy is less than 80%.
- **Scenario 2:** We will perform the same task as in scenario 1 and add some additional tasks. We will automatically start the training of a multilayer perceptron neural network model, provide the classification accuracy score, and check whether it is less than 80%. If it is, we will run `train-auto-nn.py`, which will look for the best hyperparameters of our model and print the new accuracy and the best hyperparameters.



Scenario 1

We will first create a container image using Dockerfile. Previous sections and articles describe how to do this: Quick Install and First Use of Docker, and Build and Run a Docker Container for your Machine Learning Model and <https://towardsdatascience.com/machine-learning-prediction-in-real-time-using-docker-and-python-rest-apis-with-flask-4235aa2395eb>. Then, we will use the build pipeline in Jenkins to create a job chain. We will use a simple model, linear discriminant analysis coded with scikit-learn, which we will train with EEG data (`train.csv`). In our first scenario, we want to design a Jenkins process in which each job will perform different tasks:

- **Job #1:** Pull the GitHub repository automatically when we update our code in GitHub.

- **Job #2:** Automatically start the machine learning application, train the model, and provide the prediction accuracy. Check whether the model accuracy is less than 80%.

Jenkins uses Linux, with a user called *jenkins*. For our *jenkins* user to use the **sudo** command, we might want to tell the OS not to ask for a password while executing commands. To do that, we can type the following:

```
sudo visudo /etc/sudoers
```

This command will open *sudoers* file in edit mode, and you can add or modify the file as follows:

```
jenkins ALL=(ALL) NOPASSWD: ALL
```

An alternative, perhaps safer, way is to create a file within the */etc/sudoers.d* directory, as all files included in the directory will be automatically processed, avoiding the modification of the *sudoers* file and preventing any conflicts or errors during an upgrade. The only thing we need to do is to include this command at the bottom of the *sudoers* file:

```
#includedir /etc/sudoers.d
```

To create a new file in */etc/sudoers.d* with the correct permissions, we use the following command:

```
sudo visudo -f /etc/sudoers.d/filename
```

Now we only need to include the relevant line in our file:

```
jenkins ALL=(ALL) NOPASSWD: ALL
```

We now open Jenkins and click on *Freestyle project*:

The screenshot shows the Jenkins interface for creating a new project. The top bar says "Enter an item name" and has a "Required field" placeholder. Below it, the input field contains "example-pipeline". A list of project types is displayed:

- Freestyle project**: Described as the central feature of Jenkins, combining any SCM with any build system.
- Pipeline**: Described as orchestrating long-running activities across multiple build slaves.
- Multi-configuration project**: Suitable for projects with many configurations, like testing on multiple environments.
- Bitbucket Team/Project**: Scans a Bitbucket Cloud Team or Bitbucket Server Project.
- Folder**: Creates a container for grouping items together.
- GitHub Organization**: Scans a GitHub organization or user account.

At the bottom, there is an "OK" button and a note about creating a multibranch pipeline.

Job #1: Pull the GitHub repository automatically when we modify our ML code in GitHub.

We click on *Create a job* and name it *download* or whatever name we choose:

The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with links like 'New Item', 'People', 'Build History', 'Manage Jenkins', 'My Views', 'Lockable Resources', and 'New View'. Under 'Build Queue', it says 'No builds in the queue.' Below that is 'Build Executor Status' with two entries: '1 Idle' and '2 Idle'. In the center, it says 'Welcome to Jenkins!' with a sub-section 'Start building your software project' containing a 'Create a job' button. To the right, there's a section 'Set up a distributed build' with 'Set up an agent' and 'Configure a cloud' buttons, and a link 'Learn more about distributed builds'. At the bottom of the dashboard, the URL '192.168.1.107:8080/newJob' is shown, along with 'REST API' and 'Jenkins 2.277.3'.

In Source Code Management, we select Git and insert our repository URL and our credentials:

This screenshot shows the 'Source Code Management' configuration for a Jenkins job. The 'Source Code Management' tab is selected. Under 'Repositories', 'Git' is chosen, and the 'Repository URL' is set to 'https://github.com/xaviersvasques/Jenkins.git'. The 'Credentials' dropdown shows 'vasquesxavier@hotmail.com/*****'. There are buttons for 'Advanced...', 'Add Repository', and 'Add Branch'. In the 'Branches to build' section, the 'Branch Specifier (blank for 'any')' field contains '**'. There are buttons for 'Add Branch' and 'Advanced...'. At the bottom, there are sections for 'Repository browser' (set to '(Auto)') and 'Additional Behaviours' (with an 'Add +' button).

Next, we go to Build Triggers and select Poll SCM:

Trigger builds remotely (e.g., from scripts) ?

Build after other projects are built ?

Build periodically ?

GitHub hook trigger for GITScm polling ?

Poll SCM ?

Schedule
H/10****

Would last have run at Sunday, May 2, 2021 at 6:17:01 PM Central European Summer Time; would next run at Sunday, May 2, 2021 at 6:27:01 PM Central European Summer Time.

We can click on “?” to get help. As an example, H/10**** means download the code from GitHub every 10 minutes. This is not really useful for our example, so we can leave the Schedule box empty. If we leave it empty, it will only run due to SCM changes if triggered by a post-commit hook.

Then, we click on the “Add build step” drop-down and select “Execute shell.” We type the following command to copy the repository on GitHub to a specific path previously created:

```
sudo -S cp * /home/xavi/Public/Code/Kubernetes/Jenkins/code
```

General Source Code Management Build Triggers Build Environment Build Post-build Actions

Build Environment

- Delete workspace before build starts ?
- Use secret text(s) or file(s) ?
- Abort the build if it's stuck ?
- Add timestamps to the Console Output ?
- Inspect build log for published Gradle build scans ?
- With Ant ?

Build

Execute shell

Command

```
sudo -S cp * /home/xavi/Public/Code/Kubernetes/Jenkins/code
```

See the list of available environment variables

Advanced...

Add build step ▾

Post-build Actions

Add post-build action ▾

Save Apply

We now click *save*.

We can click on “Build Now,” and we should see our code in the created repository. When we modify our files in our GitHub repository (git add, git commit, git push), the files will automatically be updated in our created repository.

Job #2: Automatically start to train our machine learning model and provide the prediction accuracy.

Let us start by building our Docker image (we could put this step directly into Jenkins):

```
docker build -t my-docker -f Dockerfile .
```

Following the same procedure, we will create a new job. We need to go *Build Triggers*, click on *Build after other projects are built*, and type the name of **Job #1** (*download* in our case):

We also click *Trigger only if build is stable*.

Then, in *Build* we open *Execute shell* and type the following commands:

```

if sudo docker ps -l | grep my-docker-lda
then
    echo "my-docker-lda already built"
else
    echo "my-docker-lda not built"
fi

if sudo grep "clf_lda" /home/xavi/Public/Code/Kubernetes/Jenkins/code/train-lda.py
then
    sudo docker run -p 5000:5000 my-docker-lda python3 train-lda.py > result.txt
    sudo cat result.txt
fi

res=$(sudo cat result.txt)
test=$0

if [ $test -gt $res ]
then
    echo "yes"
else
    echo "no"
fi

```

Here, we check whether *my-docker-lda* is already built, and we then run our container and save the accuracy of our LDA model in a *result.txt* file. The next step is to check whether the accuracy of the model is less than 80% and provide the output “yes” if this is the case or “no” otherwise. We can also send an email to provide the information: <https://plugins.jenkins.io/email-ext>.

To see the outputs of the job, we simply go to *Dashboard*, *Last Success* column, select the job, and go to *Console Output*.

All	W	Name	Last Success	Last Failure	Last Duration
S	W	auto-launch	17 min · #100 ·	14 hr · #91 ·	1.7 sec
S	W	download	24 min ·	1 day 15 hr · #71 ·	0.66 sec

Scenario 2

Let us keep **Job #1** and **Job #2** and create a new job.

Job #3: Automatically start the neural network training, provide the prediction accuracy, and check whether accuracy is less than 80%. If so, run a docker container to perform autoML.

Source Code Management

- None
- Git

Build Triggers

- Trigger builds remotely (e.g., from scripts)
- Build after other projects are built

Projects to watch

```
auto-launch
```

- Trigger only if build is stable
- Trigger even if the build is unstable
- Trigger even if the build fails

- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM

Build

Execute shell

```
if sudo docker ps -l | grep my-docker-autoML
then
    echo 'my-docker-autoML already built'
else
    echo 'my-docker-autoML not built'
fi

sudo docker run -p 5000:5000 my-docker-nn python3 train-nn.py > nn_result.txt
res=$(sudo cat nn_result.txt)
test=$0

if [ "$test" -gt "$res" ]
then
    echo 'accuracy not superior to 80%'
    sudo cat nn_result.txt
else
    sudo docker run -p 5000:5000 my-docker-nn python3 train-auto-nn.py > auto_result.txt
    sudo cat auto_result.txt
fi
```

See the list of available environment variables

Add build step ▾ Advanced...

Jenkins

Dashboard

New Item

People

Build History

Project Relationship

Check File Fingerprint

Manage Jenkins

My Views

Lockable Resources

All	W	Name	Last Success	Last Failure	Last Duration
●	●	auto-launch	2 min 23 sec - #108	17 hr - #91	1.3 sec
●	●	autoML	2 min 13 sec - #8	18 min - #7	49 sec
●	●	download	2 min 28 sec - #116	1 day 18 hr - #71	0.33 sec

Icon: \$ ML

Legend: ● Atom feed for all ● Atom feed for failures ● Atom feed for just latest builds

We should see in the *Console Output* the selected parameters and new accuracy.

Jenkins mainly involves automation of data science code.

The next steps would be to consider using Ansible with Jenkins, as Ansible could play an important role in a CI/CD pipeline. Ansible will perform the deployment of the application, and we would not need to worry about either how to deploy the application or whether the environment has been properly established.

6.4 Machine Learning with Docker and Kubernetes: Install a Cluster from Scratch

Kubernetes, the open-source container orchestration platform, is certainly one of the most important tools for scaling ML/DL efforts. To understand the utility of Kubernetes for data scientists, we can consider all the applications we have developed and containerized. How will we coordinate and schedule all these containers? How can we upgrade our machine learning models without interruptions of service? How do we scale the models and make them available to users over the internet? What happens if our model is used by many more people than we had expected? If we had not thought about the architecture before, we would need to increase the computing resources and certainly manually create new instances and redeploy the application. Kubernetes schedules, automates, and manages tasks of container-based architectures. Kubernetes deploys containers, updates them, and provides service discovery, monitoring, storage provisioning, load balancing, and more.

If we search for “Kubernetes” on the internet, we will see articles comparing Docker and Kubernetes; this is like comparing an apple and an apple pie. The first thing to state is that Kubernetes is designed to run on a cluster, while Docker runs on a single node. Kubernetes and Docker are complementary in creating, deploying, and scaling containerized applications. There is also a comparison to be made between Kubernetes and Docker Swarm, which is a tool for clustering and scheduling Docker containers. Kubernetes has several options that provide important advantages such as high-availability policies, autoscaling capabilities, and the possibility to manage complex and hundreds of thousands of containers running on a public, hybrid, or multi-cloud, or on-premises environments.

All the files used in this chapter can be found on GitHub at <https://github.com/xaviervasques/kubernetes.git>.

6.4.1 Kubernetes Vocabulary

In Kubernetes, there are different concepts to learn. We have the Kubernetes master, which is a set of three processes running on one single node of our cluster that we call the master node: **kube-apiserver**, **kube-controller-manager**, and **kube-scheduler**. Each node (excluding the master node) in our cluster runs two processes: **kubelet**, which communicates with the Kubernetes master, and **kube-proxy**, which is a network proxy reflecting the Kubernetes network services on each node. The API server is used for communication between components, the controller manager checks the state against the desired state, and the scheduler decides which Pods should run. The **kubelet** is the primary “node agent” that runs on each node. The **kube-proxy** is the Kubernetes network proxy running on each node.

In Kubernetes, the **Control Plane** (the Master Node) is the orchestrator and the **nodes** are the machines (physical servers, virtual machines, etc.) that run containerized applications. The nodes are controlled by the master node (see Figure 6.1 for the typical Kubernetes architecture). We also have the **Pod**, which is the basic building block of Kubernetes. Pods are the smallest deployable units of computing that we can create and manage in Kubernetes. They are made of one or more containers. We can consider, for example, creating a Pod that trains our machine learning model (the code is in a single container). We also must deal with **Jobs**, which creates one or more Pods. The Jobs will make sure that the execution of the Pods is successful. We can create a Job to train our models, to perform batch inference, or to store information such as training metadata or predictions to external storage. Kubernetes can help put our ML/DL models into production by simplifying the process of exposing the models to others. We will see that we need to follow some steps such as creating a deployment specifying which container to run and how many replicas we want to create and exposing the deployment using a service to define the rules for exposing Pods to each other and to the internet. A very important feature of Kubernetes is load balancing to perform the work of balancing traffic among replicas and autoscaling resources to meet increased demands.

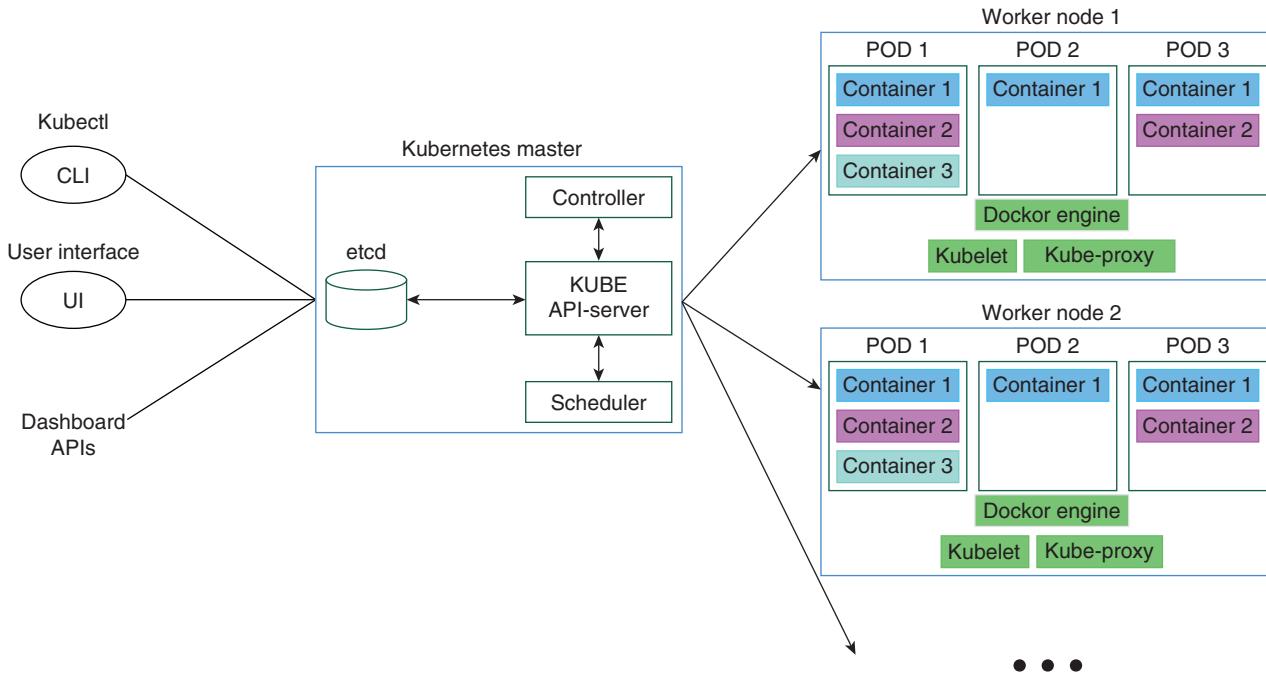


Figure 6.1 Typical Kubernetes architecture.

6.4.2 Kubernetes Quick Install

We can use different methods to install Kubernetes. All of them are well explained on the Kubernetes website at <https://kubernetes.io/fr/docs/tasks/tools/install-kubectl/>.

We can use Homebrew by typing the following in a terminal:

```
brew install kubectl
```

Alternatively, we can type the following:

Ubuntu/Debian:

```
sudo apt-get update
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/
sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubectl
```

Red Hat/CentOS:

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
EOF
```

```
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
yum install -y kubectl
```

We can check the installation with the following command:

```
kubectl version --client
```

Then, we can install all Kubernetes binaries (kubeadm, kubelet, kubectl):

```
sudo apt-get install -y kubelet kubeadm kubernetes-cni
systemctl enable kubelet
```

6.4.3 Install a Kubernetes Cluster

To become familiar with Kubernetes, we will use Vagrant, a tool to build and manage virtual machines. Vagrant creates and configures lightweight environments in a single workflow by using a configuration file named **Vagrantfile**. Let us install Vagrant.

Ubuntu/Debian:

```
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com
$(lsb_release -cs) main"
sudo apt-get update sudo apt-get install vagrant
```

Red Hat/CentOS:

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
sudo yum -y install vagrant
```

If we want to install it on other platforms, we can check <https://www.vagrantup.com/downloads>.

Vagrant relies on interactions with providers such as VirtualBox, VMware, or Hyper-V to provide Vagrant with resources to run development environments. Installation of one of them is required.

We can install **VirtualBox**:

```
sudo apt update
sudo apt install virtualbox
sudo apt install virtualbox-dkms
```

We can then create a “Vagrantfile”:

```
sudo vagrant init bento/ubuntu-20.04
```

For our project, we will create or edit a specific Vagrantfile to build our own environments as follows:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.define "kubmaster" do |kub|
    kub.vm.box = "bento/ubuntu-20.04"
    kub.vm.hostname = 'kubmaster'
    kub.vm.provision "docker"
    config.vm.box_url = "bento/ubuntu-20.04"

    kub.vm.network :private_network, ip: "192.168.56.101"

    kub.vm.provider :virtualbox do |v|
      v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
      v.customize ["modifyvm", :id, "--memory", 2048]
      v.customize ["modifyvm", :id, "--name", "master"]
      v.customize ["modifyvm", :id, "--cpus", "2"]
    end
  end

  config.vm.define "kubnode1" do |kubnode|
    kubnode.vm.box = "bento/ubuntu-20.04"
    kubnode.vm.hostname = 'kubnode1'
    kubnode.vm.provision "docker"
    config.vm.box_url = "bento/ubuntu-20.04"

    kubnode.vm.network :private_network, ip: "192.168.56.102"

    kubnode.vm.provider :virtualbox do |v|
      v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
      v.customize ["modifyvm", :id, "--memory", 2048]
      v.customize ["modifyvm", :id, "--name", "kubnode1"]
      v.customize ["modifyvm", :id, "--cpus", "2"]
    end
  end

  config.vm.define "kubnode2" do |kubnode|
    kubnode.vm.box = "bento/ubuntu-20.04"
    kubnode.vm.hostname = 'kubnode2'
    kubnode.vm.provision "docker"
    config.vm.box_url = "bento/ubuntu-20.04"

    kubnode.vm.network :private_network, ip: "192.168.56.103"

    kubnode.vm.provider :virtualbox do |v|
      v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
      v.customize ["modifyvm", :id, "--memory", 2048]
      v.customize ["modifyvm", :id, "--name", "kubnode2"]
      v.customize ["modifyvm", :id, "--cpus", "2"]
    end
  end
end
```

As can be read in the Vagrantfile, we are creating a master node that we name “kubmaster” with Ubuntu version 20.04, two GB of memory and two CPUs, one IP address (192.168.56.101), and Docker. Then we create two nodes (kubnode1 and kubnode2) with the same configuration as the master node.

Once edited, we change to root and type the following in the terminal (where the Vagrantfile is):

```
vagrant up
```

This command will create and configure guest machines according to our edited Vagrantfile. When the process is finished, we can connect to the guest machines by using the following commands in a terminal:

```
vagrant ssh kubmaster
vagrant ssh kubnode1
vagrant ssh kubnode2
```

We must deactivate the swap on the master node (kubmaster) and each node (kubnode1 and kubnode2). We connect to each guest machine to perform the following commands:

```
swapoff -a
vim /etc/fstab
```

In the *fstab* file, we also need to comment out the swap line:

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
/dev/mapper/vgvgvagrant-root / ext4 errors=remount-ro 0 1
# /boot/efi was on /dev/sdal during installation
UUID=5A33-E8B5 /boot/efi vfat umask=0077 0 1
#/dev/mapper/vgvgvagrant-swap_1 none swap sw 0 0
#VAGRANT-BEGIN
# The contents below are automatically generated by Vagrant. Do not modify.
vagrant /vagrant vboxsf uid=1000,gid=1000,_netdev 0 0
#VAGRANT-END
~
```

Let us also install some packages on each machine (kubmaster, kubnode1, kubnode2) such as curl and apt-transport-https:

```
apt-get update && apt-get install -y apt-transport-https curl
```

We can now perform a *curl* to get the gpg key that will allow us to use the Kubernetes binaries kubectl, kubeadm, and kubelet:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

We add access to the Google repository (<http://apt.kubernetes.io>), which will allow us to download and install the binaries:

```
add-apt-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

To install the binaries, we enter the following:

```
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
systemctl enable kubelet
```

6.4.4 Kubernetes: Initialization and Internal Network

Now that we have installed the necessary packages in all our nodes, we will work on the initialization and network to join the different parts of the Kubernetes cluster.

To initiate the master node, we connect to the master node and type the following:

```
root@kubmaster:~# kubeadm init --apiserver-advertise-address=192.168.56.101
--node-name $HOSTNAME --pod-network-cidr=10.244.0.0/16
```

192.168.56.101 is the IP address of the master node we defined previously, and 10.244.0.0/16 is a mask for the Kubernetes internal network defining the range that Kubernetes will use to assign IP addresses within its network.

We will see the following output with a token that has been generated to join our different nodes:

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

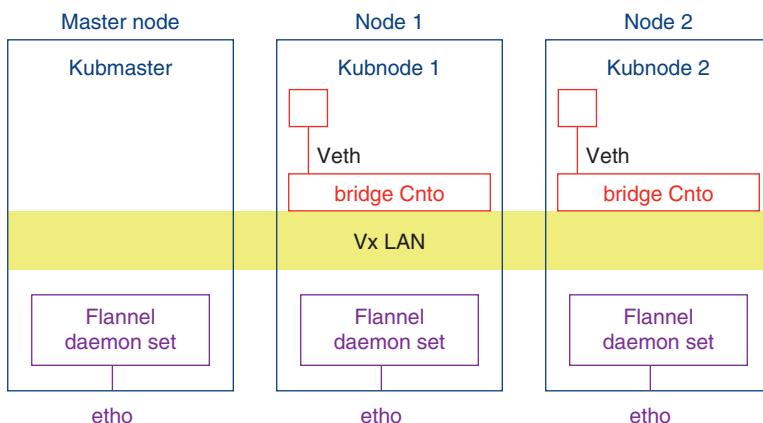
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.56.101:6443 --token 0dgzgxg.86lj9mirurile2f \
--discovery-token-ca-cert-hash sha256:59ca1ef21c1c4304ff1558210cf78528a6750bab3fa69241e925c2d5f7d90c6
```

As can be read in the output, to start using our cluster, we need to create the configuration file to work with kubectl:

```
root@kubmaster:~# mkdir -p $HOME/.kube
root@kubmaster:~# cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
root@kubmaster:~# chown $(id -u):$(id -g) $HOME/.kube/config
```

To establish the internal network, we need to provide a network between nodes in the cluster. We will use Flannel, which is a very simple way to configure a layer 3 network fabric designed for Kubernetes. Different solutions exist, such as WeaveNet, Contiv, Cilium, and others. Flannel runs a binary agent called *flanneld* on each host. Flannel is also responsible for allocating a subnet lease to each host out of a larger, preconfigured address space.



We add Pods to allow management of the internal network (command to be launched in all nodes):

```
sysctl net.bridge.bridge-nf-call-iptables=1
```

We then install our Flannel network using a configuration file (`kube-flannel.yml`, available online) by typing the following command in the master node:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
Documentation/kube-flannel.yml
```

We can check the status of the Pods in the master node (Flannel network, kube-scheduler, kube-apiserver, kube-controller-manager, kube-proxy, pods managing internal DNS, a Pod that stores configurations with etcd, etc.):

```
kubectl get pods --all-namespaces
```

```
[root@kubmaster:/home/vagrant# kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS    RESTARTS   AGE
kube-system   coredns-558bd4d5db-4tm72           1/1     Running   0          31m
kube-system   coredns-558bd4d5db-96gw9           1/1     Running   0          31m
kube-system   etcd-kubmaster                     1/1     Running   0          31m
kube-system   kube-apiserver-kubmaster          1/1     Running   0          31m
kube-system   kube-controller-manager-kubmaster  1/1     Running   0          31m
kube-system   kube-flannel-ds-dx8tv              1/1     Running   0          67s
kube-system   kube-proxy-8bmj4                  1/1     Running   0          31m
kube-system   kube-scheduler-kubmaster          1/1     Running   0          32m
[root@kubmaster:/home/vagrant#
[root@kubmaster:/home/vagrant#
[root@kubmaster:/home/vagrant# ]
```

It is sometimes necessary to modify our configuration of flannel by editing the network (from 10.244.0.0/16 to 10.10.0.0/16). We can enter the following for this purpose:

```
kubectl edit cm -n kube-system kube-flannel-cfg
# edit network 10.244.0.0/16 to 10.10.0.0/16
```

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  cni-conf.json: |
    {
      "name": "cbr0",
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "flannel",
          "delegate": {
            "hairpinMode": true,
            "isDefaultGateway": true
          }
        },
        {
          "type": "portmap",
          "capabilities": {
            "portMappings": true
          }
        }
      ]
    }
  net-conf.json: |
    {
      "Network": "10.10.0.0/16",
      "Backend": {
        "Type": "vxlan"
      }
    }
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"cni-conf.json":"\\n \\\"name\\\": \\\"cbr0\\\",\\n \\\"cniVersion\\\": \\\"0.3.1\\\",\\n \\\"plugins\\\": [\\n \\\"type\\\": \\\"flannel\\\",\\n \\\"delegate\\\": {\\n \\\"hairpinMode\\\": true,\\n \\\"isDefaultGateway\\\": true\\n \\} \\n },\\n \\\"type\\\": \\\"portmap\\\",\\n \\\"capabilities\\\": [\\n \\\"portMappings\\\": true\\n \\} \\n ] \\n ],\\n \\\"net-conf.json\\\": {\\n \\\"Network\\\": \\\"10.244.0.0/16\\\",\\n \\\"Backend\\\": {\\n \\\"Type\\\": \\\"vxlan\\\" \\n } \\n },\\n \\\"kind\\\": \\\"ConfigMap\\\",\\n \\\"metadata\\\": {\\n \\\"annotations\\\": {},\\n \\\"labels\\\": {\\n \\\"app\\\": \\\"flannel\\\",\\n \\\"tier\\\": \\\"node\\\",\\n \\\"name\\\": \\\"kube-flannel-cfg\\\",\\n \\\"namespace\\\": \\\"kube-system\\\",\\n \\\"resourceVersion\\\": \\\"2721\\\",\\n \\\"uid\\\": 9a53c35e-2588-4b49-a254-d73a1a400974\\n } \\n } \\n } \\n } \\n "
  creationTimestamp: "2021-05-13T09:56:28Z"
  labels:
    app: flannel
    tier: node
    name: kube-flannel-cfg
    namespace: kube-system
    resourceVersion: "2721"
    uid: 9a53c35e-2588-4b49-a254-d73a1a400974
  
```

If we type the command below in the master node, we can see that our master is ready:

```
kubectl get nodes
```

```
[root@kubmaster:/home/vagrant# kubectl get nodes
NAME      STATUS    ROLES          AGE     VERSION
kubmaster  Ready     control-plane,master  34m    v1.21.1]
```

It is now time to join the nodes to the master. For this, we copy the previously generated token and type the following command in the two nodes (kubnode1 and kubnode2):

```
kubeadm join 192.168.56.101:6443 --token 0dgzxg.86lj9mirurlele2f \
--discovery-token-ca-cert-hash
sha256:59ca1ef21c1c4304ff1558210cf78528a6750babc3fa69241e925c2d5f7d90c6
```

We will see the following output:

```
[root@kubnode1:/home/vagrant# kubeadm join 192.168.56.101:6443 --token 0dgzxg.86lj9mirurlele2f --discovery-token-ca-cert-hash sha256:59ca1ef21c1c4304ff1558210cf78528a6750babc3fa69241e925c2d5f7d90c6
[preflight] Running pre-flight checks
[WARNING IsDockerSystemdCheck]: detected "cgrouprps" as the Docker cgroup driver. The recommended driver is "systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
root@kubnode1:/home/vagrant# ]
```

We can return to the master node and type the following commands to check the status:

```
kubectl get pods --all-namespaces
```

```
[root@kubmaster:/home/vagrant# kubectl get pods --all-namespaces
NAMESPACE   NAME           READY   STATUS    RESTARTS   AGE
kube-system  coredns-558bd4d5db-4tm72   1/1     Running   0          47m
kube-system  coredns-558bd4d5db-96gw9   1/1     Running   0          47m
kube-system  etcd-kubmaster            1/1     Running   0          47m
kube-system  kube-apiserver-kubmaster  1/1     Running   0          47m
kube-system  kube-controller-manager-kubmaster  1/1     Running   0          47m
kube-system  kube-flannel-ds-56f9h     1/1     Running   0          3m6s
kube-system  kube-flannel-ds-dx8tv     1/1     Running   0          17m
kube-system  kube-flannel-ds-kgxvt    1/1     Running   0          2m25s
kube-system  kube-proxy-8bmj4        1/1     Running   0          47m
kube-system  kube-proxy-clfl7        1/1     Running   0          3m6s
kube-system  kube-proxy-png7g        1/1     Running   0          2m25s
kube-system  kube-scheduler-kubmaster 1/1     Running   0          48m
root@kubmaster:/home/vagrant# ]
```

```
kubectl get nodes
```

```
[root@kubmaster:/home/vagrant# kubectl get nodes
NAME      STATUS    ROLES          AGE     VERSION
kubmaster  Ready     control-plane,master  48m    v1.21.1
kubnode1   Ready     <none>        3m23s   v1.21.1
kubnode2   Ready     <none>        2m42s   v1.21.1
root@kubmaster:/home/vagrant# ]
```

We should see the nodes with the status “Ready.” We can also execute `docker ps` to see all our launched containers (coredns, flannel, etc.) in both the master node and the others.

One last comment pertains to cluster access. If we type in the master node:

```
kubectl get nodes
```

We can see that our kubmaster and nodes have the same internal IP:

```
kubectl get nodes -o wide
```

```
[root@kubmaster:/home/vagrant# kubectl get nodes -o wide
NAME      STATUS    ROLES          AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE           KERNEL-VERSION   CONTAINER-RUNTIME
kubmaster  Ready     control-plane,master  92m    v1.21.1   10.0.2.15    <none>       Ubuntu 20.04.1 LTS  5.4.0-58-generic  docker://20.10.6
kubnode1   Ready     <none>        47m    v1.21.1   10.0.2.15    <none>       Ubuntu 20.04.1 LTS  5.4.0-58-generic  docker://20.10.6
kubnode2   Ready     <none>        47m    v1.21.1   10.0.2.15    <none>       Ubuntu 20.04.1 LTS  5.4.0-58-generic  docker://20.10.6
```

We need to edit `/etc/hosts` and add our master node IP:

```
vim /etc/hosts
```

```
192.168.56.101  kubmaster      kubmaster
127.0.0.1        localhost
127.0.1.1        vagrant.vm    vagrant

# The following lines are desirable for IPv6 capable hosts
::1      localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
127.0.2.1 kubmaster kubmaster
```

The `/etc/hosts` file also needs to be modified in each node (kubnode1 and kubenode2):

```
192.168.56.102  kubnode1      kubnode1
127.0.0.1        localhost
127.0.1.1        vagrant.vm    vagrant

# The following lines are desirable for IPv6 capable hosts
::1      localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
127.0.2.1 kubnode1 kubnode1
```

Then, we can delete each flannel by typing the following in the master node:

```
kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-558bd4d5db-4tm72	1/1	Running	0	109m
coredns-558bd4d5db-96gw9	1/1	Running	0	109m
etcd-kubmaster	1/1	Running	0	110m
kube-apiserver-kubmaster	1/1	Running	0	110m
kube-controller-manager-kubmaster	1/1	Running	0	110m
kube-flannel-ds-56f9h	1/1	Running	0	65m
kube-flannel-ds-dx8tv	1/1	Running	0	79m
kube-flannel-ds-kgxvt	1/1	Running	0	64m
kube-proxy-8bmj4	1/1	Running	0	109m
kube-proxy-clfl7	1/1	Running	0	65m
kube-proxy-png7g	1/1	Running	0	64m
kube-scheduler-kubmaster	1/1	Running	0	110m

Then, we input the following lines:

```
kubectl delete pods kube-flannel-ds-56f9h -n kube-system
kubectl delete pods kube-flannel-ds-dx8tv -n kube-system
kubectl delete pods kube-flannel-ds-kgxvt -n kube-system
```

The magic of Kubernetes is that it can reconfigure again without losing the service and have new flannels:

NAME	READY	STATUS	RESTARTS	AGE
coredns-558bd4d5db-4tm72	1/1	Running	0	119m
coredns-558bd4d5db-96gw9	1/1	Running	0	119m
etcd-kubmaster	1/1	Running	0	120m
kube-apiserver-kubmaster	1/1	Running	0	120m
kube-controller-manager-kubmaster	1/1	Running	0	120m
kube-flannel-ds-kms29	1/1	Running	0	2m18s
kube-flannel-ds-l96q6	1/1	Running	0	106s
kube-flannel-ds-q8595	1/1	Running	0	117s
kube-proxy-8bmj4	1/1	Running	0	119m
kube-proxy-clfl7	1/1	Running	0	75m
kube-proxy-png7g	1/1	Running	0	74m
kube-scheduler-kubmaster	1/1	Running	0	120m

We will see that we have the correct IP addresses:

```
kubectl get nodes -o wide
```

root@kubmaster:/home/vagrant# kubectl get nodes -o wide										
NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	COUNTAINER-RUNTIME	
kubmaster	Ready	control-plane,master	121m	v1.21.1	192.168.56.101	<none>	Ubuntu 20.04.1 LTS	5.4.0-58-generic	docker://20.10.6	
kubnode1	Ready	<none>	76m	v1.21.1	192.168.56.102	<none>	Ubuntu 20.04.1 LTS	5.4.0-58-generic	docker://20.10.6	
kubnode2	Ready	<none>	76m	v1.21.1	192.168.56.103	<none>	Ubuntu 20.04.1 LTS	5.4.0-58-generic	docker://20.10.6	

To be more comfortable, we can also add autocompletion by typing these lines:

```
apt-get install bash-completion
echo "source <(kubectl completion bash)" >> ~/.bashrc
source ~/.bashrc
```

Now that we know how to install a Kubernetes cluster, and we can create Kubernetes Jobs that will create Pods (and hence containers) allowing us, for instance, to train our machine learning models, to serialize them, to load models into memory, and to perform inferences.

6.5 Machine Learning with Docker and Kubernetes: Training Models

6.5.1 Kubernetes Jobs: Model Training and Batch Inference

In this chapter, we will work on Kubernetes Jobs and see how we can use these Jobs to train a machine learning model. A Job creates one or more Pods. It is a Kubernetes controller making sure that the Pods successfully terminate their workload. The Job is considered complete when a specified number of Pods have terminated. If a Pod fails, the Job will create a new Pod to replace it.

Our goal will be to create a Kubernetes Job, called “training Job,” that loads training data stored in GitHub, trains a machine learning model, serializes the model, and stores it outside the cluster. To store the data outside the cluster, we could use different locations such as a public cloud, private cloud, or on-premises storage. In our example, we will use SSH (Secure Shell Protocol) between our Kubernetes cluster and an external server. The reason we are using external storage is that Jobs will create Pods running containers that are independent from one to another. We need to persist our model object with the help of external storage.

All the files used in this chapter can be found on GitHub at <https://github.com/xaviervasques/kubernetes.git>.

6.5.2 Create and Prepare the Virtual Machines

To install our Kubernetes Cluster, let us create virtual machines or use bare-metal servers. All instructions in this article have been tested using virtual machines created with Ubuntu version 20.04:

- kubmaster: 2 vCPUs, 4096 MB of RAM, 20 GB of drive space
- kubenode1: 2 vCPUs, 4096 MB of RAM, 20 GB of drive space

We enable traffic between the VM and the host machine. We change to root and make sure to turn off the swap and comment out the reference swap in /etc/fstab:

```
swapoff -a
vim /etc/fstab
```

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
# / was on /dev/sda5 during installation
UUID=10d50295-cad2-4748-b6ec-38962a551e45 / ext4 errors=remount-ro 0 1
# /boot/efi was on /dev/sda1 during installation
UUID=7506-7FD9 /boot/efi vfat umask=0077 0 1
# swapfile none swap sw 0 0
```

6.5.3 Kubeadm Installation

First, we will install a Docker engine in each virtual machine (in our case, kubmaster and kubenode1) using our preferred methodology (<https://docs.docker.com/engine/install>). We will use the repository. The different steps below will update the apt package index, install the packages to allow apt to use a repository over an https connection, add Docker’s official GPG key, set up the stable repository, and install the latest version of Docker Engine and containers.

```
sudo apt-get update
```

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

```
echo \
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

After changing to root (sudo -s), we perform a *curl* to obtain the gpg key that will allow us to use the Kubernetes binaries kubectl, kubeadm, and kubectl:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

We add access to the Google repository (<http://apt.kubernetes.io>) that will allow us to download and install the binaries:

```
add-apt-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

To install the binaries, we enter the following:

```
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
systemctl enable kubelet
```

All these steps must be performed in all nodes of our cluster (master and nodes).

6.5.4 Create a Kubernetes Cluster

Now that we have installed the necessary packages in all our nodes, and we will work on the initialization and network to join the different parts of the Kubernetes cluster.

To initiate the master node, we connect to the master node and type the following:

```
root@kubmaster:~# kubeadm init --apiserver-advertise-address=192.168.1.55
--node-name $HOSTNAME --pod-network-cidr=10.244.0.0/16
```

192.168.1.55 is the IP address of the master node (kubmaster) we defined previously, and 10.244.0.0/16 is a mask for the Kubernetes internal network defining the range that Kubernetes will use to assign IP addresses within its network.

We receive the following output:

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.1.55:6443 --token 2xt6km.20ro31607vnflleq \
    --discovery-token-ca-cert-hash sha256:f51c6d28ffad6729b7f974e602036473774a26da92c38f47263625f7dd14c7ca
root@kubmaster:/home/xavi/Public#
```

As can be read in the output, to start using our cluster, we need to create the configuration file to work with kubectl (as a regular user):

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

To put the internal network in place, we need to provide a network between nodes in the cluster. For this, we will use Flannel, which is a very simple way to configure a layer 3 network fabric designed for Kubernetes. We need to provide the possibility for managing the internal network (a command to be launched in all nodes):

```
sysctl net.bridge.bridge-nf-call-iptables=1
```

We then install our Flannel network using a configuration file (kube-flannel.yml), available online, by typing the following command in the master node:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
Documentation/kube-flannel.yml
```

We can check the status of the Pods in the master node (Flannel network, kube-scheduler, kube-apiserver, kube-controller-manager, kube-proxy, Pods managing internal DNS, a Pod that stores configurations with etcd, etc.):

```
kubectl get pods --all-namespaces
```

If everything is running, it is time to join the nodes to the master. We copy the previously generated token and type the following command in the nodes (kubenode1):

```
kubeadm join 192.168.1.55:6443 --token 08hcql.zbbieuknlh96f1px \
    --discovery-token-ca-cert-hash
sha256:851d02642c9b2177dd89c8e7cf7178c36185d61799eaaec4fec99b172809373f
```

We return to the master node and type the following command to check the status:

```
kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-558bd4d5db-b2tq8	1/1	Running	0	11m
kube-system	coredns-558bd4d5db-q9v2t	1/1	Running	0	11m
kube-system	etcd-kubmaster	1/1	Running	0	11m
kube-system	kube-apiserver-kubmaster	1/1	Running	0	11m
kube-system	kube-controller-manager-kubmaster	1/1	Running	0	11m
kube-system	kube-proxy-2b52z	1/1	Running	0	7m31s
kube-system	kube-proxy-lv4tf	1/1	Running	0	11m
kube-system	kube-scheduler-kubmaster	1/1	Running	0	11m

From typing the command below in the master node, we can see that our master and kubnode1 are ready:

```
kubectl get nodes
```

root@kubmaster:/home/xavi# kubectl get nodes				
NAME	STATUS	ROLES	AGE	VERSION
kubnode1	Ready	<none>	5s	v1.21.1
kubmaster	Ready	control-plane,master	4m13s	v1.21.1

6.5.5 Containerize our Python Application that Trains Models

As we have seen above, we need to create our **Dockerfile** to indicate to Docker a base image, the settings we need, and the commands to execute. We will use the **jupyter/scipy-notebook** image as our base image:

```
FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE_LDA=clf_lda.joblib
ENV MODEL_FILE_NN=clf_nn.joblib
ENV METADATA_FILE=metadata.json

COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY id_rsa ./id_rsa
COPY train.py ./train.py
```

As can be seen, we have set our environment variables and have installed **joblib**, which allows serialization and deserialization of our trained models, and **paramiko**, which is a Python implementation of the SSHv2 protocol (**requirements.txt**).

We have set environment variables from the beginning to persist the trained model and add data and metadata. We have copied the **train.py** file into the image. We have also copied the *id_rsa* file, generated using *ssh-keygen*, to be able to connect to a remote server through SSH.

We must set up an RSA key authentication to establish the connection between the cluster and the external server. We need to generate a public (*id_rsa.pub*) and a private key (*id_rsa*) that we can use to authenticate:

```
ssh-keygen -t rsa
```

```
Your identification has been saved in /home/xavi/.ssh/id_rsa
Your public key has been saved in /home/xavi/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:JX91d5MN+8xsECLSAkGV7Cpr7Qbyb0Dls0t7ABxcINY xavi@xavi
The key's randomart image is:
+---[RSA 3072]---+
| oo.o..o+o.. o |
| . E . o+. . +o|
| . . =.+. +o=|
| o o ++. . B+|
| o oS... *|
| . * . . . |
| + B |
| = = |
| ..Bo |
+---[SHA256]---+
```

We copy and paste the content of `id_rsa.pub` into `~/.ssh/authorized_keys` (on the target system):

```
vim /home/xavi/.ssh/id_rsa.pub
```

If we wish, we can copy the `id_rsa` file into our current directory and modify the permissions with `chmod`:

```
sudo cp /home/xavi/.ssh/id_rsa .
sudo chmod a+r id_rsa
```

We can then remove the `id_rsa` file with the `docker run` command.

For this step, we could use various methodologies such as writing a Dockerfile similar to the following by ensuring we remove the `id_rsa` file at the end of the build process:

```
ARG SSH_PRIVATE_KEY
RUN mkdir /root/.ssh/
RUN echo "${SSH_PRIVATE_KEY}" > /root/.ssh/id_rsa
# [...]
RUN rm /root/.ssh/id_rsa
```

We can build the Docker image with code similar to the following:

```
sudo docker build -t kubernetes-models -f Dockerfile --build-arg SSH_PRIVATE_KEY=
"$(cat ~/.ssh/id_rsa)" .
```

It is not the topic of this chapter, but we need to be careful regarding not leaving traces inside our Docker image. Even if we are deleting a file, it still can be viewed in one of the layers of the image we will push. We can use the `-squash` parameter to reduce multiple layers between the origin and the latest stage to one. In addition, the `-squash` parameter can be used to reduce the size of an image by removing files that are no longer present. We can also work with multi-stage builds in a single Dockerfile in which we build multiple Docker images. Only the last one will persist and leave traces:

```
# This is intermediate
FROM ubuntu as intermediate

# [...]

# Final image
FROM ubuntu

# [...]
```

For security reasons, we can also use Kubernetes secret objects to store and manage sensitive information such as passwords, OAuth tokens, and SSH keys. Putting this information in a secret object is safer and more flexible than putting it in the definition of a Pod or in an image container: <https://kubernetes.io/docs/concepts/configuration/secret/>.

Now that we have our Dockerfile, let us look at other files. In **train.py**, we import the necessary libraries, read the environment variables set in our Docker image for persisting models, load training data (`train.csv`), which is stored on GitHub (<https://raw.githubusercontent.com/xaviervasques/kubernetes/main/train.csv>), train two models (linear discriminant analysis and a multilayer perceptron neural network), serialize them, perform a cross-validation, and upload the trained models and cross-validation results in a remote server (192.168.1.11) and specified directory (`/home/xavi/output/`). We could also define the URL, specified directory, and the IP address as environment variables:

```
#!/usr/bin/python3
# train.py
# Xavier Vasques 16/05/2021

import os
import json
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
import pandas as pd
from joblib import dump, load
from sklearn import preprocessing
import paramiko

def train():

    # Load directory paths for persisting model

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    METADATA_FILE = os.environ["METADATA_FILE"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)
    METADATA_PATH = os.path.join(MODEL_DIR, METADATA_FILE)

    # Load training data
    url="https://raw.githubusercontent.com/xaviervasques/Jenkins/main/train.csv"
    data_train = pd.read_csv(url)

    y_train = data_train['# Letter'].values
    X_train = data_train.drop(data_train.loc[:, 'Line':'# Letter'].columns, axis = 1)

    # Print the shape of the training data
    print("Shape of the training data")
    print(X_train.shape)
    print(y_train.shape)

    # Data normalization (0,1)
    X_train = preprocessing.normalize(X_train, norm='l2')
```

```

# Models training

# Linear Discriminant Analysis (Default parameters)
clf_lda = LinearDiscriminantAnalysis()
clf_lda.fit(X_train, y_train)

# Serialize model
from joblib import dump
dump(clf_lda, MODEL_PATH_LDA)

# Neural Network: multi-layer perceptron (MLP)
clf_NN = MLPClassifier(solver='adam', activation='relu', alpha=0.0001,
hidden_layer_sizes=(500,), random_state=0, max_iter=1000)
clf_NN.fit(X_train, y_train)
# Serialize model
from joblib import dump, load
dump(clf_NN, MODEL_PATH_NN)

# Perform cross validation and store it in a json file
accuracy_lda = cross_val_score(clf_lda, X_train, y_train, cv=5)
accuracy_nn = cross_val_score(clf_NN, X_train, y_train, cv=5)

print(accuracy_lda)

metadata = {
    "train_accuracy_lda": accuracy_lda.mean(),
    "train_accuracy_nn": accuracy_nn.mean()
}

print("Serializing metadata to: {}".format(METADATA_PATH))
with open(METADATA_PATH, 'w') as outfile:
    json.dump(metadata, outfile)

print("Moving to 192.168.1.11")
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
k = paramiko.RSAKey.from_private_key_file('id_rsa')
client.connect("192.168.1.11", username="xavi", pkey=k)

# upload the file using SFTP
sftp = client.open_sftp()

sftp.put(MODEL_PATH_LDA, "/home/xavi/output/" + MODEL_FILE_LDA)
sftp.put(MODEL_PATH_NN, "/home/xavi/output/" + MODEL_FILE_NN)
sftp.put(METADATA_PATH, "/home/xavi/output/" + METADATA_FILE)

sftp.close()
client.close()

if __name__ == '__main__':
    train()

```

We can go to the next step by building the Docker image, running it to test our application locally, tagging it with the name of an image repository on the Docker Hub registry, and pushing it to the registry, to be ready to use the image in our Kubernetes cluster:

```
docker build -t kubernetes-models -f Dockerfile .
```

```
Sending build context to Docker daemon 2.854MB
Step 1/10 : FROM jupyter/scipy-notebook
--> 256408565b0e
Step 2/10 : RUN mkdir my-model
--> Using cache
--> 565d176445c2
Step 3/10 : ENV MODEL_DIR=/home/jovyan/my-model
--> Using cache
--> 3ab376af4f9c
Step 4/10 : ENV MODEL_FILE_LDA=clf_lda.joblib
--> Using cache
--> 52b82f88ae07
Step 5/10 : ENV MODEL_FILE_NN=clf_nn.joblib
--> Using cache
--> 8884701c240d
Step 6/10 : ENV METADATA_FILE=metadata.json
--> Using cache
--> 3296866d38df
Step 7/10 : COPY requirements.txt ./requirements.txt
--> Using cache
--> fbb694b9f8cf
Step 8/10 : RUN pip install -r requirements.txt
--> Using cache
--> 8392e1e5d6ec
Step 9/10 : COPY id_rsa ./id_rsa
--> Using cache
--> 3da71eedddbe
Step 10/10 : COPY train.py ./train.py
--> Using cache
--> 89fd0ed4ddef
Successfully built 89fd0ed4ddef
Successfully tagged kubernetes-models:latest
```

To test if our code is functional, we will run a container locally and test our built image:

```
docker run kubernetes-models python3 train.py rm ./id_rsa
```

If everything is working, we can push a new image to the repository using the CLI:

```
docker login
docker tag kubernetes-models:latest xaviervasques/kubernetes-models:latest
docker push xaviervasques/kubernetes-models:latest
```

6.5.6 Create Configuration Files for Kubernetes

Once the image has been successfully uploaded to the registry, we can go to our project directory (connect to kubmaster) and create a Job configuration that executes our Python code: **job.yaml**.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: train-models-job
```

```

spec:
  template:
    spec:
      containers:
        - name: train-container
          imagePullPolicy: Always
          image: xaviersvasques/kubernetes-models:latest
          command: ["python3", "train.py", "rm", "./id_rsa" ]
          restartPolicy: Never
  backoffLimit: 4

```

As explained in the Kubernetes documentation, as with all other Kubernetes configurations, a Job needs **apiVersion**, **kind**, and **metadatafields**. A Job also needs a **.spec** section. **apiVersion** specifies the version of the Kubernetes API to use, and **kind** is the type of Kubernetes resource; we can provide a label with **metadata**. The **.spec.template** is the only required field of **.spec** and represents a Pod template, as it has the same scheme except that it is nested with no **apiVersion** or **kind**. In **.spec.template.spec.containers**, we provide each container with a name, the image we desire to use, and the command we want to run in the container. Here, we want to run **train.py** and remove our **id_rsa** file.

The equivalent docker command would be the following:

```
docker run kubernetes-models python3 train.py rm ./id_rsa
```

Kubernetes will look for the image from the registry instead of using a cached image, thanks to **imagePullPolicy**. Finally, we set whether containers should be restarted if they fail (Never or OnFailure) and decide the number of retries before considering a Job as failed (the back-off limit is set by default to 6 minutes).

We are finally ready to get our application running on Kubernetes. We launch the following command:

```
kubectl create -f job.yaml
```

We check the Job objects:

```
kubectl get jobs
kubectl get pods --selector=job-name=train-models-job
```

NAME	READY	STATUS	RESTARTS	AGE
train-models-job-vd5rq	0/1	Completed	0	5m5s

We can view the logs of the process:

```
kubectl logs train-models-job-vd5rq
```

```
[root@kubmaster:/home/xavi/Public# kubectl logs train-models-job-vd5rq
Shape of the training data
(1300, 160)
(1300,
[0.87307692 0.91923077 0.90769231 0.81538462 0.90384615]
Serializing metadata to: /home/jovyan/my-model/metadata.json
Moving to 192.168.1.11
```

All files are stored on the remote server.

6.5.7 Commands to Delete the Cluster

There are several useful commands that we can find in the Kubernetes documentation, including the following:

Delete the job:

```
kubectl delete job train-models-job
```

Reset kubeadm:

```
sudo kubeadm reset
```

Delete the entire cluster:

```
kubeadm reset
sudo apt-get purge kubeadm kubectl kubelet kubernetes-cni kube*
sudo apt-get autoremove
sudo rm -rf ~/.kube
```

The next steps would be to perform batch inference and scoring by loading predictions from our trained models and also to perform real-time, online inference using REST APIs. In addition, it is essential to explore the way we set the hybrid or multi-cloud architecture (end to end) to run our models in production in an open and agile environment.

6.6 Machine Learning with Docker and Kubernetes: Batch Inference

In this section, we will implement a batch inference from trained models in the Kubernetes Cluster we developed and installed in the previous chapter.

All the files used in this chapter can be found on GitHub at <https://github.com/xaviervasques/kubernetes.git>.

To begin, we will need to modify our previous **Dockerfile** as follows:

```
FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE_LDA=clf_lda.joblib
ENV MODEL_FILE_NN=clf_nn.joblib
ENV METADATA_FILE=metadata.json

COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY id_rsa ./id_rsa
COPY inference.py ./inference.py
COPY train.py ./train.py
```

Compared to our previous **Dockerfile** (Part II), we have added **inference.py**:

```
#!/usr/bin/python3
# inference.py
# Xavier Vasques 13/04/2021
```

```

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import load
from sklearn import preprocessing
import paramiko

def inference():

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

    # Load, read and normalize testing data
    testing = "https://raw.githubusercontent.com/xaviervasques/kubernetes/main/test.csv"
    data_test = pd.read_csv(testing)

    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)
    print("Shape of the test data")
    print(X_test.shape)
    print(y_test.shape)

    # Data normalization (0,1)
    X_test = preprocessing.normalize(X_test, norm='l2')

    # Go to remote server, load trained models, output scores and predictions
    print("Moving to 192.168.1.11, load LDA and NN models and provide accuracy scores and predictions")
    ssh_client = paramiko.SSHClient()
    ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    k = paramiko.RSAKey.from_private_key_file('id_rsa')
    ssh_client.connect("192.168.1.11", username="LRENC", pkey=k)

    print(MODEL_PATH_LDA)
    sftp_client = ssh_client.open_sftp()
    remote_file = sftp_client.open("/Users/LRENC/Desktop/Kubernetes/Jenkins/" + MODEL_FILE_LDA)

    #clf_lda = load(MODEL_PATH_LDA)
    clf_lda = load(remote_file)
    print("LDA score and classification:")
    print(clf_lda.score(X_test, y_test))
    print(clf_lda.predict(X_test))

```

```

    remote_file.close()

    print(MODEL_PATH_NN)
    sftp_client = ssh_client.open_sftp()
    remote_file = sftp_client.open("/home/xavi/output/" + MODEL_FILE_NN)

    # Run model
    #clf_nn = load(MODEL_PATH_NN)
    clf_nn = load(remote_file)
    print("NN score and classification:")
    print(clf_nn.score(X_test, y_test))
    print(clf_nn.predict(X_test))

    remote_file.close()

if __name__ == '__main__':
    inference()

```

In `inference.py`, we import the necessary libraries, read the environment variables set in our Docker image, and read new data (`test.csv`) stored at GitHub (<https://raw.githubusercontent.com/xaviersques/kubernetes/main/test.csv>) to feed our serialized models to make predictions and provide an accuracy score. We will download our previously trained models (linear discriminant analysis and a multilayer perceptron neural network) stored in a specified directory (`/home/xavi/output`) from a remote server (192.168.1.11) using SSH.

We can then go to the next step by building the Docker image, running it to test our application locally, tagging it with the name of an image repository on the Docker Hub registry, and pushing it to the registry to be ready to use the image in our Kubernetes cluster:

```
docker build -t kubernetes-inference -f Dockerfile .
```

```

Sending build context to Docker daemon 4.092MB
Step 1/11 : FROM jupyter/scipy-notebook
--> 256408565b0e
Step 2/11 : RUN mkdir my-model
--> Using cache
--> 565d176445c2
Step 3/11 : ENV MODEL_DIR=/home/jovyan/my-model
--> Using cache
--> 3ab376af4f9c
Step 4/11 : ENV MODEL_FILE_LDA=clf_lda.joblib
--> Using cache
--> 52b82f88ae07
Step 5/11 : ENV MODEL_FILE_NN=clf_nn.joblib
--> Using cache
--> 8884701c240d
Step 6/11 : ENV METADATA_FILE=metadata.json
--> Using cache
--> 3296866d38df
Step 7/11 : COPY requirements.txt ./requirements.txt
--> Using cache
--> fbb694b9f8cf
Step 8/11 : RUN pip install -r requirements.txt
--> Using cache
--> 8392e1e5d6ec
Step 9/11 : COPY id_rsa ./id_rsa
--> Using cache
--> 8b82c631a7b5
Step 10/11 : COPY inference.py ./inference.py
--> Using cache
--> bd08a02d3eca
Step 11/11 : COPY train.py ./train.py
--> Using cache
--> 539caeae6273
Successfully built 539caeae6273
Successfully tagged kubernetes-models:latest

```

To test if our code is functional, we will run our container locally to test the image:

```
docker run kubernetes-inference python3 train.py rm ./id_rsa
```

If everything is working, we can push the new image to a repository using the CLI:

```
docker login
docker tag kubernetes-inference:latest xaviervasques/kubernetes-inference:latest
docker push xaviervasques/kubernetes-inference:latest
```

6.6.1 Create Configuration Files for Kubernetes

Once the image has been successfully uploaded to the registry, we go to our project directory (connect to kubmaster) and create a configuration file, **inference.yaml**, that executes our Python code:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: inference-job
spec:
  template:
    spec:
      containers:
        - name: inference-container
          imagePullPolicy: Always
          image: xaviervasques/kubernetes-inference:latest
          command: ["python3", "inference.py", "rm", "./id_rsa" ]
          restartPolicy: Never
  backoffLimit: 4
```

The equivalent docker command would be the following:

```
docker run kubernetes-inference python3 inference.py rm ./id_rsa
```

We are finally ready to get our application running on Kubernetes. We launch the following command:

```
kubectl create -f inference.yaml
```

We check the Job objects:

```
kubectl get jobs
kubectl get pods --selector=job-name=inference-job
```

```
[root@kubmaster:/home/xavi/Public# kubectl get jobs
NAME           COMPLETIONS  DURATION   AGE
inference-job  1/1         4m7s       5m34s
train-models-job 1/1         86s        17h
[root@kubmaster:/home/xavi/Public# kubectl get pods --selector=job-name=inference-job
NAME            READY   STATUS    RESTARTS   AGE
inference-job-stskf  0/1    Completed  0          5m36s
```

We can view the logs of the process (output of our code):

```
kubectl logs inference-job-stskf
```

```
Linux-5.8.0-53-generic-x86_64-with-glibc2.10
Python 3.8.8 | packaged by conda-forge | (default, Feb 20 2021, 16:22:27)
[GCC 9.3.0]
NumPy 1.20.2
SciPy 1.6.3
Shape of the test data
(1300, 160)
(1300,)
Moving to 192.168.1.11, load LDA and NN models and provide accuracy scores and predictions
/home/jovyan/my-model/clf_lda.joblib
LDA score and classification:
0.6915384615384615
[ 0  0  0 ... 25 25 25]
/home/jovyan/my-model/clf_nn.joblib
NN score and classification:
0.6615384615384615
[ 0  0  0 ... 25 25 24]
```

Kubernetes serves as an excellent framework with which to deploy models effectively. We can use Kubernetes to deploy each of our models as independent and lightweight microservices. These microservices can be used for other applications. The next steps would be to deploy online inferences using REST APIs. We can also work with Kubernetes to schedule our training and inference processes to run on a recurring schedule.

6.7 Machine Learning Prediction in Real Time Using Docker, Python Rest APIs with Flask, and Kubernetes: Online Inference

The idea of this section is to create a Docker container to perform online inference with trained machine learning models using Python APIs with Flask. As an example of this concept, we will implement online inferences (linear discriminant analysis and multilayer perceptron neural network models) with Docker and Flask-RESTful.

To start, let us consider the following files:

- **Dockerfile**
- **train.py**
- **api.py**
- **requirements.txt**
- **train.csv**
- **test.json**

All files can be found on GitHub at <https://github.com/xaviervasques/Kubernetes-ML-Online.git>.

The file **train.py** is a Python script that ingests and normalizes data and trains two models to classify the data. The **Dockerfile** will be used to build our Docker image, requirements.txt (flask, flask-restful, joblib) is for the Python dependencies, and **api.py** is the script that will be called to perform the online inference using REST APIs. The file **train.csv** contains the data used to train our models, and **test.json** is a file containing new data that will be used with our inference models.

All the files used in this chapter can be found on GitHub at <https://github.com/xaviervasques/kubernetes.git>.

6.7.1 Flask-RESTful APIs

The first step in building APIs is to consider the data we wish to process, how we want to process it, and what output we want with our APIs. In our example, we will use the **test.json** file in which we have 1300 rows of EEG data with 160 features each (columns). We want our APIs to do the following:

- **API 1:** We will give a row number to the API that will extract for us the data from the selected row and print it.
- **API 2:** We will give a row number to the API that will extract the selected row, inject the new data into the models, and retrieve the classification prediction (# Letter variable in the data).
- **API 3:** We will ask the API to take all the data in the **test.json** file and instantly print the classification score of the models.

In the end, we want to access these processes by making an HTTP request.

Let us look at the **api.py** file:

```
#!/usr/bin/python3
# api.py
# Xavier Vasques 13/04/2021

import json
import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import load
from sklearn import preprocessing

from flask import Flask

# Set environment variables
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

# Loading LDA model
print("Loading model from: {}".format(MODEL_PATH_LDA))
inference_lda = load(MODEL_PATH_LDA)

# loading Neural Network model
print("Loading model from: {}".format(MODEL_PATH_NN))
inference_NN = load(MODEL_PATH_NN)

# Creation of the Flask app
app = Flask(__name__)

# API 1
# Flask route so that we can serve HTTP traffic on that route
@app.route('/line/<Line>')
# Get data from json and return the requested row defined by the variable Line
def line(Line):
    with open('./test.json', 'r') as jsonfile:
        file_data = json.loads(jsonfile.read())
    # We can then find the data for the requested row and send it back as json
    return json.dumps(file_data[Line])
```

```

# API 2
# Flask route so that we can serve HTTP traffic on that route
@app.route('/prediction/<int:Line>',methods=['POST', 'GET'])
# Return prediction for both Neural Network and LDA inference model with the requested
row as input
def prediction(Line):
    data = pd.read_json('./test.json')
    data_test = data.transpose()
    X = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)
    X_test = X.iloc[Line,:].values.reshape(1, -1)

    clf_lda = load(MODEL_PATH_LDA)
    prediction_lda = clf_lda.predict(X_test)

    clf_nn = load(MODEL_PATH_NN)
    prediction_nn = clf_nn.predict(X_test)

    return {'prediction LDA': int(prediction_lda), 'prediction Neural Network': int(prediction_nn)}


# API 3
# Flask route so that we can serve HTTP traffic on that route
@app.route('/score',methods=['POST', 'GET'])
# Return classification score for both Neural Network and LDA inference model from the
all dataset provided
def score():

    data = pd.read_json('./test.json')
    data_test = data.transpose()
    y_test = data_test['# Letter'].values
    X_test = data_test.drop(data_test.loc[:, 'Line':'# Letter'].columns, axis = 1)
    clf_lda = load(MODEL_PATH_LDA)
    score_lda = clf_lda.score(X_test, y_test)

    clf_nn = load(MODEL_PATH_NN)
    score_nn = clf_nn.score(X_test, y_test)

    return {'Score LDA': score_lda, 'Score Neural Network': score_nn}

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0')

```

The first step, after importing dependencies including the open-source web microframework Flask, is to set the environment variables that are written in the Dockerfile. We also need to load our linear discriminant analysis and multilayer perceptron neural network serialized models. We create our Flask application by writing `app = Flask(__name__)`. Then, we create our three Flask routes so that we can serve HTTP traffic on those routes:

- `http://0.0.0.0:5000/line/250`: Get data from test.json and return the requested row defined by the variable Line (in this example, we want to extract the data of row number 250).

- `http://0.0.0.0:5000/prediction/51`: Return a classification prediction from both LDA and neural network-trained models by injecting the requested data (in this example, we want to inject the data of row number 51).
- `http://0.0.0.0:5000/score`: Return a classification score for both the neural network and the LDA inference models on all the available data (`test.json`).

The Flask routes allow us to request what we need from the API by adding the name of our procedure (`/line<Line>`, `/prediction<int:Line>`, `/score`) to the URL (`http://0.0.0.0:5000`). Whatever data we add, `api.py` will always return the output we request.

6.7.2 Machine Learning Models

The `train.py` file is a Python script that ingests and normalizes data from a csv file (`train.csv`) and trains two models to classify the data (using scikit-learn). The script saves two models: linear discriminant analysis (`clf_lda`) and neural network multilayer perceptron (`clf_NN`):

```
#!/usr/bin/python3
# tain.py
# Xavier Vasques 13/04/2021

import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)

import os
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
import pandas as pd
from joblib import dump
from sklearn import preprocessing

def train():

    # Load directory paths for persisting model
    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE_LDA = os.environ["MODEL_FILE_LDA"]
    MODEL_FILE_NN = os.environ["MODEL_FILE_NN"]
    MODEL_PATH_LDA = os.path.join(MODEL_DIR, MODEL_FILE_LDA)
    MODEL_PATH_NN = os.path.join(MODEL_DIR, MODEL_FILE_NN)

    # Load, read and normalize training data
    training = "./train.csv"
    data_train = pd.read_csv(training)

    y_train = data_train['# Letter'].values
    X_train = data_train.drop(data_train.loc[:, 'Line':'# Letter'].columns, axis = 1)

    print("Shape of the training data")
    print(X_train.shape)
    print(y_train.shape)
```

```

# Data normalization (0,1)
X_train = preprocessing.normalize(X_train, norm='l2')

# Models training

# Linear Discriminant Analysis (Default parameters)
clf_lda = LinearDiscriminantAnalysis()
clf_lda.fit(X_train, y_train)

# Serialize model
from joblib import dump
dump(clf_lda, MODEL_PATH_LDA)

# Neural Networks multi-layer perceptron (MLP) algorithm
clf_NN = MLPClassifier(solver='adam', activation='relu', alpha=0.0001,
hidden_layer_sizes=(500,), random_state=0, max_iter=1000)
clf_NN.fit(X_train, y_train)

# Serialize model
from joblib import dump, load
dump(clf_NN, MODEL_PATH_NN)

if __name__ == '__main__':
    train()

```

6.7.3 Docker Image for Online Inference

We are ready to build our Docker image. To start, we need our **Dockerfile** with the **jupyter/scipy-notebook** image as our base image. We also need to set our environment variables and install joblib, to allow serialization and deserialization of our trained models, and Flask (**requirements.txt**). We copy the **train.csv**, **test.json**, **train.py**, and **api.py** files into the image. We then run **train.py**, which will fit and serialize the machine learning models as part of our image build process.

Here is the code:

```

FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE_LDA=clf_lda.joblib
ENV MODEL_FILE_NN=clf_nn.joblib

COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY train.csv ./train.csv
COPY test.json ./test.json

COPY train.py ./train.py
COPY api.py ./api.py

RUN python3 train.py

```

To build this image, we run the following command:

```
docker build -t my-kube-api -f Dockerfile .
```

Let us now test our container in action.

6.7.4 Running Docker Online Inference

The goal now is to run our online inference locally to verify that everything runs well. Running our docker container means that each time a client issues a POST request to the /line/<Line>, /prediction/<Line>, /score endpoints, we will show the requested data (row), predict the class of the data we inject using our pre-trained models, and present the score of our pre-trained models using all the available data. To launch the web server, we will run our Docker container and **api.py**:

```
docker run -it -p 5000:5000 my-kube-api python3 api.py
```

The **-p** flag exposes port 5000 in the container to port 5000 on our host machine, and the **-it** flag allows us to see the logs from the container; we run **python3 api.py** in the **my-api** image.

The output is the following:

```
[xavi@xavi:~/Public/Code/Kubernetes/Kubernetes-ML-Online$ sudo docker run -it -p 5000:5000 my-kube-api python3 api.py
Loading model from: /home/jovyan/my-model/clf_lda.joblib
Loading model from: /home/jovyan/my-model/clf_nn.joblib
* Serving Flask app 'api' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://172.17.0.2:5000/ (Press CTRL+C to quit)
* Restarting with stat
Loading model from: /home/jovyan/my-model/clf_lda.joblib
Loading model from: /home/jovyan/my-model/clf_nn.joblib
* Debugger is active!
* Debugger PIN: 129-273-518
```

It can be seen that we are running on <http://172.17.0.2:5000/>; we can now use our web browser or the **curl** command to issue a POST request to the IP address:

```
Curl http://172.17.0.2:5000/line/23
```

We will get the row number 23 extracted from our data (test.json).

If we type

```
curl http://172.17.0.2:5000/prediction/23
```

we will see the following output:

```
[xavi@xavi:~/Public/Code/Kubernetes/Kubernetes$ curl http://172.17.0.2:5000/prediction/23
{
  "prediction LDA": 21,
  "prediction Neural Network": 0
}
xavi@xavi:~/Public/Code/Kubernetes/Kubernetes$ ]
```

The above output means that the LDA model has classified the provided data (row 23) as letter 21 (U) while multilayer perceptron neural network has classified the data as letter 0 (A). The two models do not agree.

If we type

```
curl http://172.17.0.2:5000/score
```

we will see the score of our models on the entire dataset:

```
xavi@xavi:~/Public/Code/Kubernetes/Kubernetes$ curl http://172.17.0.2:5000/score
{
  "Score LDA": 0.17846153846153845,
  "Score Neural Network": 0.5969230769230769
}
xavi@xavi:~/Public/Code/Kubernetes/Kubernetes$
```

As can be seen, we should trust the multilayer perceptron neural network more, with its accuracy score of 0.59, even though the score is not so high. There is some work to do to improve the accuracy!

Now that our application is working properly, we can move to the next step and deploy it in a Kubernetes Cluster. Before doing that, let us push the image to a repository using the CLI:

```
docker login
docker tag my-kube-api:latest xaviervasques/my-kube-api:latest
docker push xaviervasques/my-kube-api:latest
```

6.7.5 Create and Prepare the Virtual Machines

To install our Kubernetes Cluster, we can create virtual machines or use bare-metal servers. All instructions in this section have been tested using virtual machines created with Ubuntu version 20.04:

- **kubmaster:** 2 vCPUs, 4096 MB of RAM, 20 GB of drive space.
- **kubenode1:** 2 vCPUs, 4096 MB of RAM, 20 GB of drive space.

We enable traffic between the VM and the host machine.

We change to root and make sure to turn off the swap and comment out the reference swap in /etc/fstab:

```
swapoff -a
vim /etc/fstab
```

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
# / was on /dev/sda5 during installation
UUID=10d50295-cad2-4748-b6ec-38962a551e45 /          ext4    errors=remount-ro 0      1
# /boot/efi was on /dev/sda1 during installation
UUID=7506-7FD9 /boot/efi    vfat    umask=0077    0      1
# swapfile
swapfile          swap    none    sw      0      0
```

6.7.6 Kubeadm Installation

First, we will install Docker Engine on each virtual machine (in our case, kubmaster and kubenode1). To install it, we choose our preferred methodology (<https://docs.docker.com/engine/install>). We will use the repository. The different steps below will update the apt package index, install the packages to allow apt to use a repository over an HTTPS connection, add Docker's official GPG key, set up the stable repository, and install the latest version of Docker Engine and containers:

```
sudo apt-get update
```

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

```
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Then, after changing to root (`sudo -s`), we perform a `curl` to obtain the gpg key that will allow us to use the Kubernetes binaries `kubectl`, `kubeadm`, and `kubelet`:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

We add access to the Google repository (`http://apt.kubernetes.io`), which will allow us to download and install the binaries:

```
add-apt-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
```

To install the binaries, we enter the following:

```
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
systemctl enable kubelet
```

All these steps must be performed in all nodes of the cluster (master and nodes).

6.7.7 Create a Kubernetes Cluster

Now that we have installed the necessary packages in all our nodes, we will work on the initialization and network to join the different parts of the Kubernetes cluster.

To initiate the master node, we connect to the master node and type the following:

```
root@kubmaster:~# kubeadm init --apiserver-advertise-address=192.168.1.55
--node-name $HOSTNAME --pod-network-cidr=10.244.0.0/16
```

192.168.1.55 is the IP address of the master node (kubmaster) we defined previously, and 10.244.0.0/16 is a mask for the Kubernetes internal network defining the range that Kubernetes will use to assign IP addresses within its network.

We receive the following output:

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.1.55:6443 --token 2xt6km.20ro31607vnflleq \
--discovery-token-ca-cert-hash sha256:f51c6d28ffad6729b7f974e602036473774a26da92c38f47263625f7dd14c7ca
root@kubmaster:/home/xavi/Public#
```

As can be read in the output, to start using our cluster, we need to create a configuration file to work with kubectl (as a regular user):

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

To establish the internal network, we need to provide a network between nodes in the cluster. For this task, we will use Flannel, which is a very simple way to configure a layer 3 network fabric designed for Kubernetes. We need to provide the possibility of managing the internal network (command to be launched in all nodes):

```
sysctl net.bridge.bridge-nf-call-iptables=1
```

We then install our Flannel network using a configuration file (kube-flannel.yml, available online) by typing the following command in the master node:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
Documentation/kube-flannel.yml
```

We can check the status of the Pods in the master node (Flannel network, kube-scheduler, kube-apiserver, kube-controller-manager, kube-proxy, Pods managing internal DNS, a Pod that stores configurations with etcd, etc.):

```
kubectl get pods --all-namespaces
```

If everything is running, it is time to join the nodes to the master. For this, we copy the previously generated token and type the following command in the nodes (kubenode1):

```
kubeadm join 192.168.1.55:6443 --token 08hcql.zbbieukn1h96f1px \
--discovery-token-ca-cert-hash
sha256:851d02642c9b2177dd89c8e7cf7178c36185d61799eaaec4fec99b172809373f
```

We come back to the master node and type the following command to check the status:

```
kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-558bd4d5db-b2tq8	1/1	Running	0	11m
kube-system	coredns-558bd4d5db-q9v2t	1/1	Running	0	11m
kube-system	etcd-kubmaster	1/1	Running	0	11m
kube-system	kube-apiserver-kubmaster	1/1	Running	0	11m
kube-system	kube-controller-manager-kubmaster	1/1	Running	0	11m
kube-system	kube-proxy-2b52z	1/1	Running	0	7m31
kube-system	kube-proxy-lv4tf	1/1	Running	0	11m
kube-system	kube-scheduler-kubmaster	1/1	Running	0	11m

If we type the command below in the master node, we can see that our master and kubenode1 are ready:

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kubenode1	Ready	<none>	5s	v1.21.1
kubmaster	Ready	control-plane,master	4m13s	v1.21.1

6.7.8 Deploying the Containerized Machine Learning Model to Kubernetes

When we have multiple YAML files that we need to execute one by one, it can be difficult to manage or become repetitive. To make it easier, we can use the Kustomize utility.

We connect to the master node (kubmaster) and install Kustomize:

```
curl -s https://api.github.com/repos/kubernetes-sigs/kustomize/releases | \
grep browser_download | \
grep linux | cut -d '"' -f 4 | \
grep /kustomize/v | \
sort | tail -n 1 | \
xargs curl -O -L && \
tar xzf ./kustomize_v*_linux_amd64.tar.gz && \
mv kustomize /usr/bin/
```

We then create a folder named “base” in the master node and create the following YAML files inside it:

- namespace.yaml
- deployment.yaml
- service.yaml
- kustomization.yaml

The **namespace.yaml** file provides a scope for Kubernetes resources:

```
apiVersion: v1
kind: Namespace
metadata:
  name: mlops
```

The **deployment.yaml** file will let us manage a set of identical Pods. If we do not use a deployment, we would need to create, update, and delete many Pods manually. It is also a way to easily autoscale our applications. In our example, we have decided to create two Pods (replicas), load the Docker image that we had pushed previously, and run our **api.py** script:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-app
    env: qa
  name: my-app
  namespace: mlops
spec:
  replicas: 2 # Creating two PODs for our app
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
        env: qa
    spec:
      containers:
        - image: xaviervasques/my-kube-api:latest # Docker image name, that we pushed to GCR
          name: my-kube-api      # POD name
          command: ["python3", "api.py" ]
          ports:
            - containerPort: 5000
              protocol: TCP
```

The **service.yaml** file will expose our application running on a set of Pods as a network service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app
  labels:
    app: my-app
  namespace: mlops
spec:
  type: LoadBalancer
  ports:
    - port: 5000
      targetPort: 5000
  selector:
    app: my-app
```

Finally, we create the **kustomization.yaml** file:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- namespace.yaml
- deployment.yaml
- service.yaml
```

To deploy our application, we use this single command in our master node:

```
kubectl apply --kustomize=${PWD}/base/ --record=true
```

To see all components deployed into this namespace, we can enter the following:

```
kubectl get ns
```

We should obtain the following output:

NAME	STATUS	AGE
default	Active	22h
kube-node-lease	Active	22h
kube-public	Active	22h
kube-system	Active	22h
mlops	Active	39m

To see the status of the deployment, we can use the following command:

```
kubectl get deployment -n mlops
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
my-app	2/2	2	2	40m

To see the status of the service, we use this command:

```
kubectl get service -n mlops
```

```
root@kubmaster:/home/xavi/Public/base# kubectl get service -n mlops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
my-app   LoadBalancer   10.97.99.101    <pending>      5000:32331/TCP   41m
```

We are now ready to use our deployed model by using curl or a web browser:

```
curl http://10.97.99.101:5000/line/23
```

We will get the row number 23 extracted from our data (**test.json**).

If we type

```
curl http://10.97.99.101:5000/prediction/23
```

we will see the following output:

```
[root@kubmaster:/home/xavi/Public/base# curl http://10.97.99.101:5000/prediction/23
{
  "prediction LDA": 21,
  "prediction Neural Network": 0
}
```

In addition, we can use the following command:

```
curl http://10.97.99.101:5000/score
```

We will then see the score of our models on the entire dataset:

```
[root@kubmaster:/home/xavi/Public/base# curl http://10.97.99.101:5000/score
{
  "Score LDA": 0.17846153846153845,
  "Score Neural Network": 0.5969230769230769
}
```

6.8 A Machine Learning Application that Deploys to the IBM Cloud Kubernetes Service: Python, Docker, Kubernetes

We are going to see that compared to the previous descriptions; it is very easy to create a Kubernetes cluster with IBM Cloud. The wealth of Kubernetes resources can make it difficult to find the basics. An easy way to simplify Kubernetes development and make it easy to deploy is to use solutions such as IBM Cloud Kubernetes Services. To create a machine learning application that deploys to the IBM Cloud Kubernetes Service, we need an IBM Cloud account (sign up for a free account: <https://cloud.ibm.com/registration>), IBM Cloud CLI, Docker CLI, and Kubernetes CLI.

6.8.1 Create Kubernetes Service on IBM Cloud

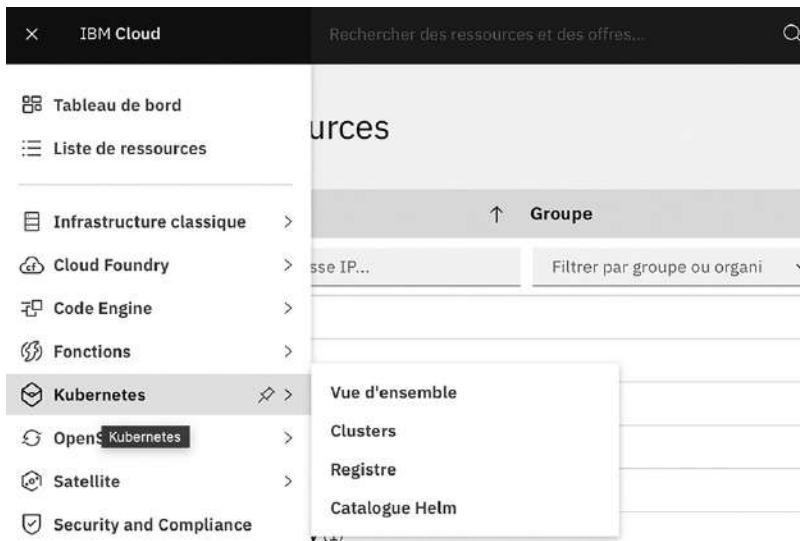
The Kubernetes service on IBM Cloud provides two cluster types:

- A free cluster (one worker pool with a single virtual-shared worker node with 2 cores, 4 GB RAM, and 100 GB SAN).
- A fully customizable standard cluster (virtual-shared, virtual-dedicated, or bare metal) for the heavy lifting.

If we only want to explore, the free cluster is ideal.

In IBM Cloud, with a few clicks, we can automatically create a Kubernetes service. First, we need to connect to our IBM Cloud Dashboard at <https://cloud.ibm.com/dashboard/apps>.

We go to IBM Kubernetes Service, click on Create clusters, and type in a name for our cluster. Depending on our account (paid or free), we can select the appropriate cluster type (in our case we will only create a worker node with 2 vCPUs and 4 GB of RAM). After a few minutes, the cluster is created:



Once the cluster is ready, we can click on our cluster's name, and we will be directed to a new page with information about our cluster and worker node:

The screenshot shows the IBM Cloud interface for managing clusters. The main title is 'IBM_Cloud_node'. Below it, there are status indicators: 'Normal' (green checkmark) and 'Expire dans 30 jours' (red warning box). A button 'Ajouter des étiquettes' is also present. The 'Présentation' section includes links for 'Noeuds worker', 'Pools de noeuds worker', and 'DevOps Nouveau'. A prominent message box says: 'Expire dans 30 jours : Assurez-vous de sauvegarder vos données car votre cluster va être supprimé dans 30 jours. Pour accéder aux fonctions complètes du service, essayez d'utiliser un cluster standard.' Below this, four status boxes show: 'Statut du nœud 1 de 1 Normal', 'Statut du module complémentaire 0 de 0 Normal', 'Etat du maître Normal', and 'Statut Ingress Inconnu'. Each box has a 'Détails' link.

To connect to our cluster, we can click on our worker node tab to get the public IP address of the cluster:

The screenshot shows the 'Nœuds worker' table. The table header includes columns: 'Nom', 'Statut', 'Pool de noeuds worker', 'Zone', 'Adresse IP privée', 'Adresse IP publique', and 'Version'. There is one row of data: '000000d3', 'Normal', 'default', 'mex01', '10.131.79.223', '169.57.43.155', and '1.20.7_1541'. Below the table, there are filters ('Pool: Filtrer...', 'Filtrer le tableau'), pagination ('Eléments par page: 25', '1-1 sur 1 élément'), and navigation buttons ('Page 1 sur 1').

Done! We can have fast access using the IBM Cloud Shell at https://cloud.ibm.com/docs/containers?topic=containers-cs_cli_install#cloud-shell.

If we want to use our own terminal, we need some prerequisites (if they are not already installed). We need to install the required CLI tools: IBM Cloud CLI, Kubernetes Service plug-in (ibmcloud ks), and Kubernetes CLI (kubectl).

To install the IBM Cloud CLI, we will type the following in a terminal to install the stand-alone IBM Cloud CLI (ibmcloud):

```
curl -fsSL https://clis.cloud.ibm.com/install/linux | sh
```

The above command is for Linux. All necessary commands for various distributions can be found at https://cloud.ibm.com/docs/containers?topic=containers-cs_cli_install.

We log in to the IBM Cloud CLI by entering our IBM Cloud credentials when prompted:

```
ibmcloud login
```

If we have a federated ID, we can use ibmcloud login –sso to log in to the IBM Cloud CLI.

Otherwise, we can also connect with an IBM Cloud API key as follows:

```
ibmcloud login --apikey < IBM CLOUD API KEY >
```

If one is not already available, we can create an IBM Cloud API key. To do this, we need to go to the IBM Cloud console, then go to Manage > Access (IAM), and select API keys:

Statut	Nom	Description	Date de création
	api-key-kube		2021-06-03 20:02 GMT

We can click create an IBM Cloud API key, add a name and description, and copy or download the API key to a secure location. We can then log in using the command above.

We can install the IBM Cloud plug-in for the IBM Cloud Kubernetes Service (ibmcloud ks):

```
ibmcloud plugin install container-service
```

We can also install the IBM Cloud plug-in for the IBM Cloud Container Registry (ibmcloud cr):

```
ibmcloud plugin install container-registry
```

We can further install the IBM Cloud Kubernetes Service observability plug-in (ibmcloud ob):

```
ibmcloud plugin install observe-service
```

The Kubernetes CLI is already installed in our environment. If it is not already installed, we can follow the steps at https://cloud.ibm.com/docs/containers?topic=containers-cs_cli_install.

If we want to list all the clusters in the account, we can input the following:

```
ibmcloud ks cluster ls
```

```
xavi@xavi:~/Public$ ibmcloud ks cluster ls
OK
Name          ID           State    Created      Workers   Location  Version       Resource Group Name  Provider
IBM_Cloud_node c2nvvf3d0rdcmi81bvs0  normal  23 hours ago  1        hou02    1.20.7_1540  default
xavi@xavi:~/Public$
```

We can check if our cluster is in a healthy state by running the following command:

```
ibmcloud ks cluster get -c IBM_Cloud_node
```

Here, *IBM_Cloud_node* is our cluster name; we can also use the ID of the cluster.

```
[xavi@xavi:~/Public$ ibmcloud ks cluster get -c IBM_Cloud_node
Retrieving cluster IBM_Cloud_node...
OK

Name:           IBM_Cloud_node
ID:            c2nvvf3d0rdcmi81bvs0
State:          normal
Status:         All Workers Normal
Created:        2021-05-27T20:23:24+0000
Location:       mex01
Pod Subnet:    172.30.0.0/16
Service Subnet: 172.21.0.0/16
Master URL:    https://c6.hou02.containers.cloud.ibm.com:26813
Public Service Endpoint URL: https://c6.hou02.containers.cloud.ibm.com:26813
Private Service Endpoint URL: -
Master Location: hou02
Master Status:   Ready (23 hours ago)
Master State:    deployed
Master Health:   normal
Ingress Subdomain: -
Ingress Secret:  -
Ingress Status:  -
Ingress Message: -
Workers:         1
Worker Zones:   mex01
Version:        1.20.7_1540
Creator:        -
Monitoring Dashboard: -
Resource Group ID: c710597f351b4d00985238dca652d63f
Resource Group Name: default
```

6.8.2 Containerization of a Machine Learning Application

This short example will demonstrate how to create a Docker container to perform online inference with a trained machine learning model using Python API with Flask. To do this, we will train a simple C-Support Vector Classification model using scikit-learn and the Iris dataset, which we will split into training data and test data.

To start, let us consider the following files:

- **Dockerfile**
- **train.py**
- **api.py**
- **requirements.txt**

All files can be found on GitHub at https://github.com/xaviervasques/ML_Kubernetes_IBM_Cloud.

The **train.py** file is a Python script that loads and trains our model. The **Dockerfile** will be used to build our Docker image, requirements.txt (flask, flask-restful, joblib) is for the Python dependencies, and **api.py** is the script that will be called to perform the online inference using an API.

The **train.py** file is the following:

```
# Deploy ML with Kubernetes on IBM Cloud
# train.py
# Xavier Vasques 03/06/2021

from sklearn import svm
from joblib import dump, load
from sklearn.model_selection import train_test_split
```

```

from sklearn.datasets import load_iris

def train():

    # Load and split the data
    iris = load_iris()
    X, y = iris.data, iris.target
    X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,shuffle=False)

    # Print the data
    print(X_train)
    print(X_test)

    # Train the model
    clf = svm.SVC()
    clf.fit(X_train, y_train)

    print ("svm.SVC Model finished training")

    # Save the trained model for online inference
    dump(clf, 'svc_model.model')

if __name__ == '__main__':
    train()

```

We also need to build an API that will ingest the data (`X_test`) and output what we want. In our case, we will only request the classification score of the model:

```

#!/usr/bin/python3
# api.py
# Xavier Vasques 03/06/2021

import os
from sklearn import svm
from joblib import dump, load
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from joblib import load

from flask import Flask

# Set environment variables
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE = os.environ["MODEL_FILE"]
MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)

# Loading model
print("Loading model from: {}".format(MODEL_PATH))
inference = load(MODEL_PATH)

# Creation of the Flask app
app = Flask(__name__)

```

```

# API
# Flask route so that we can serve HTTP traffic on that route
@app.route('/score',methods=['POST', 'GET'])
# Return predictions of inference using Iris Test Data
def prediction():

    # Load and split the data
    iris = load_iris()
    X, y = iris.data, iris.target
    X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,shuffle=False)

    # Classification score
    clf = load(MODEL_PATH)
    score = clf.score(X_test, y_test)

    return {'score': score}

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0')

```

We are now ready to containerize the Flask application. In our project directory, we have created our **Dockerfile** with **jupyter/scipy-notebook** image as our base image, set our environment variables, and installed **joblib** and **flask**; we copy **train.py** and **api.py** files into the image:

```

FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE=svc_model.model
COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY train.py ./train.py
COPY api.py ./api.py

EXPOSE 5000

RUN python3 train.py

```

We want to expose the port (5000) on which the Flask application runs, so we have used EXPOSE. To verify that our application is running without issue, let us build and run our image locally:

```
docker build -t my-kube-api -f Dockerfile .
```

```

svm.SVC Model finished training
Removing intermediate container 0696c5253645
--> b191177e565f
Successfully built b191177e565f
Successfully tagged my-kube-api:latest

```

```
docker run -it -p 5000:5000 my-kube-api python3 api.py
```

```
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ sudo docker run -it -p 5000:5000 my-kube-api python3 api.py
Loading model from: /home/jovyan/my-model/svc_model.model
 * Serving Flask app 'api' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.17.0.2:5000/ (Press CTRL+C to quit)
 * Restarting with stat
Loading model from: /home/jovyan/my-model/svc_model.model
 * Debugger is active!
 * Debugger PIN: 593-961-866
```

We can now test the application using curl:

```
curl http://172.17.0.2:5000/score
```

```
xavi@xavi:~$ curl http://172.17.0.2:5000/score
{
  "score": 0.7
}
xavi@xavi:~$
```

Everything is working fine.

6.8.3 Push the Image to the IBM Cloud Registry

It works! Now that our application is working properly, we can move to the next step and deploy it in a Kubernetes Cluster. Before doing that, we need to push the image to a repository. Here, we will push the image to the IBM Cloud Registry (a private repository). From our account dashboard, we can select **Container Registry**:

The screenshot shows the IBM Cloud Container Registry dashboard. On the left, there's a sidebar with various service icons. The 'Container Registry' icon is selected and highlighted in blue. The main area displays a summary of the cluster status:

- Statut du noeud:** 1 de 1 (Normal)
- Statut du module complémentaire:** 0 de 0 (Normal)
- Etat du maître:** Normal
- Statut Ingress:** Inconnu

Below this, there's a section titled 'Détails' with the following information:

ID de cluster: c2a9h5dd015mu0v173g	Version: 1.20.7_1540	Infrastructure: Classique	Zones: mex01
Créé: 03/06/2021 à 10:53	Groupe de ressources: default	Contrôle de la sécurité des images: Activer	

At the bottom, there's a section titled 'État de santé du noeud' showing '1 nœuds au total' with a green progress bar.

We need to install the Container Registry plug-in locally using the following command:

```
ibmcloud plugin install container-registry -r "IBM Cloud"
```

```
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ibmcloud plugin install container-registry -r "IBM Cloud"
Looking up 'container-registry' from repository 'IBM Cloud'...
Plug-in 'container-registry 0.1.025' found in repository 'IBM Cloud'
Attempting to download the binary file...
24.87 MB / 24.87 MB [=====] 100.00% 6s
26979232 bytes downloaded
Installing binary...
OK
Plug-in 'container-registry 0.1.025' was successfully installed into /home/xavi/.bluemix/plugins/container-registry. Use 'ibmcloud plugin show container-registry' to show its details.
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$
```

Then, we log in to our account:

```
ibmcloud login
```

We create and name our namespace:

```
ibmcloud cr namespace-add xaviervasques
```

```
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ibmcloud cr namespace-add xaviervasques
No resource group is targeted. Therefore, the default resource group for the account ('default') is targeted.

Adding namespace 'xaviervasques' in resource group 'default' for account XAVIER VASQUES's Account in registry de.icr.io...
Successfully added namespace 'xaviervasques'

OK
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$
```

We log our local Docker daemon into the IBM Cloud Container Registry using the following command:

```
docker login -u iamapikey -p <YOUR API KEY> de.icr.io
```

We choose a repository and tag by which we can identify the image:

```
docker tag my-kube-api de.icr.io/xaviervasques/my-kube-api:latest
```

Next, we push the image (docker push <region_url>/<namespace>/<image_name>:<tag>):

```
docker push de.icr.io/xaviervasques/my-kube-api:latest
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ sudo docker push de.icr.io/xaviervasques/my-kube-api:latest
The push refers to repository [de.icr.io/xaviervasques/my-kube-api]
42c5cbc5c209: Pushed
360508fa3244: Pushed
0e4207508317: Pushed
8c7d3eda04ae: Pushed
86daaebf426: Pushed
0408c6ff3136: Pushed
5f70bf18a086: Pushed
4a05dfe4e033: Pushed
33e5be1e164e: Pushed
d65958a5e793: Pushed
48e972418bf7: Pushed
70caa2609c09: Pushed
7f58425f5431: Pushed
87c83311cb18: Pushed
4151bbd61219: Pushed
34533dcea084: Pushed
5eb1652a177e: Pushed
7276a293c227: Pushed
af42e00bd2fb: Pushed
d394ba5f93e7: Pushed
0bea4afc52e5: Pushed
d1c91abbed83: Pushed
5b52bace2640: Pushed
6b2b9901e732: Pushed
ef7d15ebf232: Pushed
346be19f13b0: Pushed
935f303ebf75: Pushed
0e64bafdc7ee: Pushed
latest: digest: sha256:8097135fecc06903ab43c18b86fc2e3b39224af500cc7070c5ded164cf33c6f1 size: 6994
```

We can verify the status of our image by checking whether it is on our private registry:

```
ibmcloud cr image-list
```

6.8.4 Deploy the Application to Kubernetes

Once the image has been uploaded to the private registry, we can deploy the application to Kubernetes. We can use the user interface or the CLI; for this example, we will use the CLI. We create our Kubernetes cluster using the steps above (we can also create one using the command line: `ibmcloud ks cluster create classic --name my-cluster`). To see the status, we type the following command:

```
ibmcloud ks clusters
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ibmcloud ks clusters
OK
Name      ID          State     Created      Workers   Location    Version      Resource Group Name  Provider
my_k8s   c2s9h5dd015mhu0v173g  normal   11 hours ago  1        Dallas     1.20.7_1540  default
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ]
```

Our my-k8s Kubernetes cluster is up and running. We can connect kubectl to the cluster:

```
ibmcloud ks cluster config --cluster my_k8s
```

We can verify that we are connected to the cluster:

```
kubectl get nodes
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get nodes
NAME      STATUS  ROLES   AGE   VERSION
10.131.77.168  Ready  <none>  11h  v1.20.7+IKS
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ]
```

We will create a folder named “base” in the master node and create the following YAML files inside it:

- `namespace.yaml`
- `deployment.yaml`
- `service.yaml`
- `service_port.yaml`
- `kustomization.yaml`

The `namespace.yaml` file provides a scope for Kubernetes resources:

```
apiVersion: v1
kind: Namespace
metadata:
  name: mlapi
```

The `deployment.yaml` will let us manage a set of identical Pods. If we do not use a deployment, we would need to create, update, and delete many Pods manually. It is also a way to easily autoscale our applications. In our example, we have decided to create two Pods (replicas), load our Docker image that we pushed previously, and run our `api.py` script:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
```

```

app: my-app
env: qa
name: my-app
namespace: mlapi
spec:
  replicas: 1 # Creating PODs for our app
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
        env: qa
    spec:
      containers:
        - image: de.icr.io/xaviervasques/my-kube-api:latest # Docker image name, that we
uploaded
          name: my-kube-api      # POD name
          command: ["python3", "api.py" ]
          ports:
            - containerPort: 5000
              protocol: TCP
        imagePullSecrets:
          - name: all-icr-io

```

The **service.yaml** file will expose our application running on a set of Pods as a network service:

```

apiVersion: v1
kind: Service
metadata:
  name: my-app
  labels:
    app: my-app
    namespace: mlapi
spec:
  type: LoadBalancer
  ports:
    - port: 5000
      targetPort: 5000
  selector:
    app: my-app

```

We also need to create the **service_port.yaml** file:

```

apiVersion: v1
kind: Service
metadata:
  name: nodeport
spec:

```

```
type: NodePort
ports:
- port: 32743
```

The reason we create the service_port.yaml file is to make our containerized app accessible over the internet by using the public IP address of any worker node in a Kubernetes cluster and exposing a node port (NodePort). We can use this option for testing the IBM Cloud Kubernetes Service and for short-term public access (<https://cloud.ibm.com/docs/containers?topic=containers-nodeport>).

Finally, we create the **kustomization.yaml** file:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- namespace.yaml
- deployment.yaml
- service_port.yaml
- service.yaml
```

We can configure our own image pull secret to deploy containers in Kubernetes namespaces other than the default namespace. With this methodology, we can use images stored in other IBM Cloud accounts or images stored in external private registries. In addition, we can create our own image pull secret to enforce IAM access rules that restrict rights to specific registry image namespaces or actions (such as push or pull). We have several options to do that; one of them is to copy the image fetch secret from the Kubernetes default namespace to other namespaces in our cluster (<https://cloud.ibm.com/docs/containers?topic=containers-registry#other>).

Let us start by listing the namespaces in our cluster:

```
kubectl get namespaces
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get ns
NAME      STATUS  AGE
default   Active  63m
ibm-cert-store  Active  51m
ibm-operators  Active  58m
ibm-system   Active  61m
kube-node-lease  Active  63m
kube-public   Active  63m
kube-system   Active  63m
mlapi       Active  24s
```

Then, let us list the image pull secrets in the Kubernetes default namespaces for the IBM Cloud Container Registry:

```
kubectl get secrets -n default | grep icr-io
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get secrets -n default | grep icr-io
all-icr-io          kubernetes.io/dockerconfigjson    1      50m
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ]
```

To deploy our application, we use the following single command in our master node:

```
kubectl apply --kustomize=${PWD}/base/ --record=true
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl apply --kustomize=${PWD}/base/ --record=true
namespace/mlapi created
service/my-app created
service/nodeport created
deployment.apps/my-app created
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ]
```

We copy the **all-icr-io** image extraction secret from the default namespace to the namespace of our choice. The new image fetch secrets are named <namespace_name>-icr-<region>-io:

```
kubectl get secret all-icr-io -n default -o yaml | sed 's/default/mlapi/g' | kubectl
create -n mlapi -f -
```

We can check that the creation of secrets was successful:

```
kubectl get secrets -n mlapi | grep icr-io
```

To see all components deployed into this namespace, we can use the following command:

```
kubectl get ns
```

We should obtain the following output:

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get ns
NAME      STATUS  AGE
default   Active  63m
ibm-cert-store  Active  51m
ibm-operators  Active  58m
ibm-system   Active  61m
kube-node-lease  Active  63m
kube-public   Active  63m
kube-system   Active  63m
mlapi      Active  24s
```

To see the status of the deployment, we can use the following command:

```
kubectl get deployment -n mlapi
```

```
[xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get deployment -n mlapi
NAME    READY  UP-TO-DATE  AVAILABLE  AGE
my-app  1/1    1           1          2m48s
```

To see the status of the service, we use this command:

```
kubectl get service -n mlapi
```

```
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ kubectl get service -n mlapi
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
my-app    LoadBalancer   172.21.193.80   <pending>      5000:31261/TCP   3m1s
```

We can obtain the public IP address of a worker node in the cluster. If we want to access the worker node on a private network or a VPC cluster, we obtain the private IP address instead:

```
ibmcloud ks worker ls --cluster my_k8s
```

We are now ready to use our deployed model by using curl or a web browser:

```
curl http://172.21.193.80:31261/score
```

```
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ ibmcloud ks worker ls --cluster my_k8s
OK
ID                                     Public IP      Private IP      Flavor      State      Status      Zone      Version
kube-c2v4p73f0tr0u3bvvcgg-myk8s-default-000000ef  169.51.206.82  10.144.222.56  free       normal     Ready      mil01    1.20.7_1541
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$ curl http://169.51.206.82:31261/score
{
  "score": 0.7
}
xavi@xavi:~/Public/Code/Kubernetes/IBM_Cloud$
```

We can navigate further through our **Kubernetes Dashboard** to check our services and many other features.

When we work on putting machine or deep learning models into production, there are questions that arise at some point: Where can I deploy my code for training, and where can I deploy my code for batch or online inference? It can often happen that we need to deploy our machine learning flow on a multi-architecture environment and hybrid cloud or multi-cloud environment. We have seen how to deploy an application on IBM Cloud and how to deploy using on-premises or virtual machines. Kubernetes can run on a variety of platforms: from simple clusters to complex ones, from laptops to multi-architectures, and on hybrid cloud or multi-cloud Kubernetes Clusters. The question is what solution best suits our needs.

6.9 Red Hat OpenShift to Develop and Deploy Enterprise ML/DL Applications

Investment is significantly increasing in ML/DL to create value, with different objectives such as masking complexity, producing automation, reducing costs, growing businesses, better serving customers, making discoveries, performing research and innovation, and other goals. Strong open-source communities for ML/DL on Kubernetes and OpenShift have been created and are evolving. These communities are working to allow data scientists and developers to access and consume ML/DL technologies. Working on a local computer and putting ML/DL models into production requires navigating across a vast and complex space. Where do we deploy our code for training, and where do we deploy our code for batch or online inferences? There are situations in which we will need to deploy machine learning workflows in a multi-architecture environment or in a hybrid cloud environment.

Today's data centers are made of heterogeneous systems (x86, IBM Power Systems, IBM Z, high-performance computing, accelerators such as GPUs or FPGAs, and others), running heterogeneous workloads with specialized ML/DL frameworks, each with its strengths. In addition, we can see the cloud in all its dimensions (public and private cloud, hybrid cloud, multi-cloud, distributed cloud). For instance, we can maintain a database with critical data running on an IBM Power System that we want to leverage for our models, run our training code using GPUs, deploy batch or online inference on IBM Z or LinuxONE in which critical transactional applications can avoid latency, and perform another inference on a cloud or at the edge. There are an important number of options to consider depending on one's business. A typical ML/DL workflow starts with a business objective and involves a design to understand users, challenge assumptions, redefine problems, co-create (for instance, putting IT and data scientist teams in the same

room) a solution to prototype, and test by iteration. We then collect private and public data, refine and store the data, and create and validate models until we put everything into production for the real world. We need to consider scalability of the application, resilience, versioning, security, availability, and other aspects. This requires additional expertise, often related to specialized hardware resources, increasing the need for resource management and utilization.

Data scientists cannot manage the entire process, which can be complex; for those I personally know, they want to have access to high-performance hardware and be focused on the data and the creation of models. This is also why we can see machine learning applications not being completely exploited because they are not prepared for production. Containers and Kubernetes can avoid this kind of situation by accelerating ML/DL adoption and breaking all these barriers. There is a clear movement to embrace Linux containers and Kubernetes to develop ML/DL applications and deploy them. Containers and Kubernetes are a way to simplify the access to underlying infrastructure by masking the complexity, allowing management of the different workflows such as development or application lifecycles. Red Hat OpenShift will provide additional capabilities that are well suited for enterprise environments.

6.9.1 What is OpenShift?

Kubernetes is an open-source project, and Red Hat OpenShift is a certified Kubernetes platform and distribution. It is a container application platform based on Kubernetes for enterprise application development and deployment. Red Hat is one of the top contributors to the Kubernetes community. OpenShift is a family of containerization software such as OpenShift Online, OpenShift Dedicated, or OpenShift Container Platform, which is an on-premises platform-as-a-service with Docker containers orchestrated and managed by Kubernetes on Red Hat Enterprise Linux. OKD is the open-source version of OpenShift, known until August 2018 as OpenShift Origin (Origin Community Distribution). It is the upstream community project. The idea behind OpenShift involves enhancing the management and developer experience for deploying enterprise (at-scale) applications on Kubernetes.

6.9.2 What Is the Difference Between OpenShift and Kubernetes?

Kubernetes has many distributions and relies mostly on communities or external expertise if help is needed. Companies often ask for official support, especially for critical business applications, but not only for those elements. To fully operationalize a Kubernetes environment and run containerized applications across a distributed system environment, we need more than only Kubernetes expertise when the objective is to deploy enterprise applications on Kubernetes. We need to consider a number of aspects such as a robust security, developer-friendly environments, cluster management, integrated builds and CI/CD services, multi-architecture and multi-platform deployment wherever it is in the data center, public cloud, multi-cloud, the edge, virtual machines, bare metal, x86, IBM Power Systems, IBM Z, registries to deploy images, automation of operations, secure container images, management and automatic container updates, management of hybrid storage, multitenancy and multiple clusters management, and other factors.

The topic of security is a key element, and if we can avoid a headache by having a more secure foundation by default it will certainly help. The default policies on Red Hat OpenShift are stricter than on Kubernetes. For instance, role-based access control (RBAC) is an integral part of OpenShift. It may be acceptable to use Kubernetes without RBAC security for a small development or test setup, but when real life and production come it is necessary to have some level of permissions. Kubernetes is like a Linux kernel, but we need more than just a Linux kernel; we need a Linux platform distribution to run Linux applications. In practice, for those who have already installed Kubernetes, Red Hat OpenShift supports the use of **kubectl**, and users can use the native Kubernetes command-line interface. We can also use the command line for developers and take advantage of additional capabilities with other command line tools such as **oc**, which is an equivalent of Kubernetes's kubectl but with some differences such as the possibility to build a container image from a source and deploy it into environments with a single command, or **odo**, which allows users to write, build, and debug applications on a cluster without the need to administer the cluster itself. Red Hat OpenShift supports Kubernetes Operators and Deployments as well as third-party tools such as Helm Charts (application deployment), Prometheus (monitoring and alerts management), Istio (for management of a distributed microservice architecture), Knative (serverless), Internal Container Registry, a logging stack based on EFK (ElasticSearch, Fluentd, Kibana), or Jenkins, which makes it easy to deploy applications with CI/CD pipelines. We can use a single account to authenticate, making permissions management easier.

One great advantage of Red Hat OpenShift is the management of container images with *Image Stream*, allowing, for example, changing a tag for an image in a container registry without downloading the image; we can tag it locally and push it back. With OpenShift, once an image is uploaded, it can be managed within OpenShift by its virtual tag. It is also possible to define triggers that, for example, start deployment when a tag changes its reference (e.g., from *devel* to *stable* or *prod* tag) or if a new image appears. Another difference between Kubernetes and OpenShift is the web-based user interface. The Kubernetes dashboard must be installed separately, and we can access it via `kube-proxy`. Red Hat OpenShift's web console has a login page that is easy to access and very helpful for daily administrative work, as resources can be created and changed via a form. We can install Red Hat OpenShift clusters in the cloud using managed services (Red Hat OpenShift on IBM Cloud, Red Hat OpenShift Service on AWS, Azure Red Hat OpenShift) or we can run them on our own by installing from another cloud provider (AWS, Azure, Google Cloud, platform-agnostic). We also have the possibility to create clusters on supported infrastructure (bare metal, IBM Z, Power, Red Hat OpenStack, Red Hat Virtualization, vSphere, platform-agnostic) or a minimal cluster on our laptop (MacOS, Linux, Windows), which is useful for local development and testing.

6.9.3 Why Red Hat OpenShift for ML/DL? To Build a Production-Ready ML/DL Environment

I believe everybody would agree that creating high-performant ML/DL models and deploying ML/DL in production require different sets of skills. To allow deployment of an ML/DL application in production, we need to put in place an iterative process involving setting the business goals, gathering and preparing the data, developing models, deploying models, inferencing, monitoring, and managing accuracy over time. To execute this process, we need to implement an ML/DL architecture with ML/DL tools, DevOps tools, data pipelines, and access to resources (computing, storage, network) whether in private, public, hybrid, or multi-cloud environments. Red Hat OpenShift is making a difference because it allows data scientists and developers to focus on their models and code and deploy them on Kubernetes without the need to learn Kubernetes in depth. We automate once, and then we simply develop the IT environment. In other words, it is possible to manage the complexity of ML/DL model deployments and democratize access to the techniques, allowing the deployment of any containerized ML/DL stack at scale in any environment.

Red Hat OpenShift has many features and benefits that can help data scientists and developers to focus on their business and use the tools and languages with which they are most comfortable. OpenShift puts into action additional security controls as well as the tools to manage multiple applications (multitenancy environment). OpenShift makes all IT environments much easier to manage.

6.10 Deploying a Machine Learning Model as an API on the Red Hat OpenShift Container Platform: From Source Code in a GitHub Repository with Flask, Scikit-Learn, and Docker

ML/DL applications have become more popular than ever. As we have seen, Red Hat OpenShift container, an enterprise Kubernetes platform, helps data scientists and developers to focus on value creation using their preferred tools by bringing additional security controls into place and making environments much easier to manage. It provides the ability to deploy, serve, secure, and optimize machine learning models at enterprise scale and in highly available clusters, allowing data scientists to focus on the value of data. We can install Red Hat OpenShift clusters in the cloud using managed services (Red Hat OpenShift on IBM Cloud, Red Hat OpenShift Service on AWS, Azure Red Hat OpenShift), or we can run them on our own by installing from another cloud provider (AWS, Azure, Google Cloud, platform-agnostic). We can also create clusters on supported infrastructure (bare metal, IBM Z, Power, Red Hat OpenStack, Red Hat Virtualization, vSphere, platform-agnostic) or a minimal cluster on a laptop (MacOS, Linux, Windows), which is useful for local development and testing. There is a lot of freedom here.

In this section, we will demonstrate how to deploy a simple machine learning model developed in Python on an OpenShift cluster in the cloud. We will create an OpenShift cluster on IBM Cloud and show how to deploy a machine learning application from a GitHub repository and expose the application to public access (with and without a Dockerfile).

We can do all of this with a few simple steps.

6.10.1 Create an OpenShift Cluster Instance

We connect to our IBM Cloud account and click on the Navigation Menu, OpenShift, Clusters:

The screenshot shows the IBM Cloud dashboard. The left sidebar has a tree view with 'OpenShift' expanded, and 'Clusters' is selected. The main area displays several cards: 'Get started with IBM Cloud for Financial Services' (Build and deploy services that comply with security), 'Create and deploy an application' (Build and run your app), 'Get started with machine learning + Watson Studio' (Build, run, and manage AI models), 'Build a web app with Watson Speech to Text' (Deploy a conversational interface), and 'Set up your IBM Cloud account' (Learn how to set up your account). Below these cards, there's a 'Planned maintenance' section and a 'For you' sidebar with links to server hosting and cloud infrastructure management.

We then click on Create cluster:

The screenshot shows the 'OpenShift clusters' creation page. The left sidebar has 'Clusters' selected. The main area has a table header with columns: Name, State, Location, Worker Count, Created, Version, and Infrastructure. Below the table, there's a 'Clusters' section with a 'Create cluster' button. The bottom right corner has a blue circular icon with a white question mark.

We can select options such as location, computing environment to run the cluster, or worker pool (number of vCPUs, memory, encrypt local disk, etc.) and click Create. One interesting option is that we can choose Satellite, which allows us to run our cluster in our own data center. For this example, we have chosen Classic:

Orchestration service

Select the container platform type and version for your cluster. For more information about versions, including links to the container platform community release notes, see the docs.

OpenShift

4.6.34

Infrastructure

Choose which network and compute environment to run your cluster on. Learn more about the differences.

Classic Run your cluster with native subnet and VLAN networking on our classic infrastructure.	VPC Create a fully customizable, software-defined virtual network with superior isolation using IBM Cloud VPC.	Satellite Run your cluster in one of your connected data centers.
--	--	---

Done!

Clusters / mycluster-par01-b3c.4x16

Normal [Add tags](#)

Overview

Worker nodes	Node status	Add-on status	Master status	Ingress status
1 of 1	<input checked="" type="checkbox"/> Normal	0 of 0 <input checked="" type="checkbox"/> Normal	Normal	Unknown
Details ↴	Details ↴	Details ↴	Docs ↗	Docs ↗

Details

Cluster ID c3jknejf0e1gpnsl3	Version 4.6.34_1547	Infrastructure Classic	Zones par01
Created 08/07/2021, 21:02	Resource group default	Image security enforcement <input checked="" type="button"/> Enable	

Node health

1 total nodes

■ Critical 0% ■ Warning 0% ■ Normal 100% ■ Pending 0%

We can also create clusters using the CLI: <https://cloud.ibm.com/docs/openshift?topic=openshift-clusters>.

6.10.1.1 Deploying an Application from Source Code in a GitHub Repository

We can create a new OpenShift Enterprise application from source code, images, or templates, and we can do it through the OpenShift web console or the CLI. A GitHub repository has been created with all the sources and a Dockerfile in order to assemble a Docker image and later deploy it to the OpenShift cluster: <https://github.com/xaviervasques/OpenShift-ML-Online.git>.

First, let us view and test our application. We clone our repository:

```
git clone https://github.com/xaviervasques/OpenShift-ML-Online.git
```

In the folder, we should find the following files:

Dockerfile
train.py
api.py
requirements.txt

OpenShift will automatically detect whether the Docker or source-build strategy is being used. In our repository, there is a **Dockerfile**. OpenShift Enterprise will generate a Docker build strategy. The **train.py** file is a Python script that loads and splits the Iris dataset, which is a classic and very simple multi-class classification dataset consisting of petal and sepal lengths of three different types of iris (Setosa, Versicolour, and Virginica), stored in a 150×4 numpy.ndarray. We have used scikit-learn for both dataset and model creation (support vector machine [SVM] classifier). The file **requirements.txt** (flask, flask-restful, joblib) is for the Python dependencies, and **api.py** is the script that will be called to perform the inference using a REST API. The API will return the classification score of the SVM model on the test data.

The **train.py** file is the following:

```
import os
from sklearn import svm
from joblib import dump, load
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

def train():
    # Load directory paths for persisting model

    MODEL_DIR = os.environ["MODEL_DIR"]
    MODEL_FILE = os.environ["MODEL_FILE"]
    MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)

    # Load and split the data
    iris = load_iris()
    X, y = iris.data, iris.target
    X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,shuffle=False)

    # Print the data
    print(X_train)
    print(X_test)

    # Train the model
    clf = svm.SVC()
    clf.fit(X_train, y_train)
```

```

print ("svm.SVC Model finished training")

# Save the trained model for online inference
dump(clf, MODEL_PATH)

if __name__ == '__main__':
    train()

```

The **api.py** file is the following:

```

import os
from sklearn import svm
from joblib import dump, load
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from joblib import load

from flask import Flask

# Set environment variables
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE = os.environ["MODEL_FILE"]
MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)

# Loading model
print("Loading model from: {}".format(MODEL_PATH))
inference = load(MODEL_PATH)

# Creation of the Flask app
app = Flask(__name__)

# API
# Flask route so that we can serve HTTP traffic on that route
@app.route('/', methods=['POST', 'GET'])
# Return predictions of inference using Iris Test Data
def prediction():

    # Load and split the data
    iris = load_iris()
    X, y = iris.data, iris.target
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)

    # Classification score
    clf = load(MODEL_PATH)
    score = clf.score(X_test, y_test)

    return {'score': score}

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=8080) # Launch built-in web server and run
this Flask webapp

```

The Dockerfile is the following:

```
FROM jupyter/scipy-notebook

RUN mkdir my-model
ENV MODEL_DIR=/home/jovyan/my-model
ENV MODEL_FILE=svc_model.model

COPY requirements.txt ./requirements.txt
RUN pip install -r requirements.txt

COPY train.py ./train.py
COPY api.py ./api.py

#USER 1001
EXPOSE 8080

RUN python3 train.py
CMD ["python3", "api.py", "8080"]
```

Finally, the **requirements.txt** file is the following:

```
flask
flask-restful
joblib
```

To verify that everything is working, let us build and run the Docker image on our local machine:

```
cd OpenShift-ML-Online
docker build -t my-ml-api:latest .
docker run my-ml-api
```

The output is the following:

```
[xavi@xavi:~/Public/Code/OpenShift/OpenShift-ML-Online]$ sudo docker run my-ml-api
Loading model from: /home/jovyan/my-model/svc_model.model
 * Serving Flask app 'api' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.17.0.2:8080/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 281-797-315
```

We can use the API with a curl:

```
curl http://172.17.0.2:8080/
```

We should receive the following output:

```
[xavi@xavi:~$ curl http://172.17.0.2:8080/
{
  "score": 0.7
}
xavi@xavi:~$ ]
```

We are now ready to deploy.

To create a new project, we can use either the CLI (using IBM Shell or our own terminal) or the OpenShift web console. With the console, to create a new project, we select our cluster (mycluster-par01-b3c.4x16) from the OpenShift clusters console and click on OpenShift web console:

From the perspective switcher, we select Developer to switch to the Developer perspective. We can see that the menu offers items such as +Add, Builds, and Topology:

We click on **+Add** and create the project:

In this section, we will use the CLI on our local machine. We have installed our OpenShift cluster on the IBM Cloud; if this has not already been performed, it is necessary to install the following:

- IBM Cloud CLI (<https://cloud.ibm.com/docs/cli?topic=cli-install-ibmcloud-cli>).
- OpenShift Origin CLI (https://docs.openshift.com/container-platform/4.2/cli_reference/openshift_cli/getting-started-cli.html).

We then need to log in to OpenShift and create a new project. To do this, we need to copy the login command:

We log in to our account:

```
ibmcloud login
```

Then, we copy and paste the login command:

```
oc login --token=sha256~IWefYlUvt1St8K9QAXXXXXX0frXXX2-5LAXXXNq-S9E
--server=https://c101-e.eu-de.containers.cloud.ibm.com:30785
```

The new-app command allows the creation of applications using source code in a local or remote Git repository. To create an application using a Git repository, we can type the following command in the terminal:

```
oc new-app https://github.com/xaviervasques/OpenShift-ML-Online.git
```

This command has several options, such as using a subdirectory of our source code repository by specifying a --context-dir flag, specifying a Git branch, or setting the --strategy flag to specify a build strategy. In our case, we have a Dockerfile that will automatically generate a Socker build strategy.

```
[xavi@xavi:~$ oc new-app https://github.com/xaviervasques/OpenShift-ML-Online.git
--> Found container image 3d4570e (3 days old) from Docker Hub for "jupyter/scipy-notebook"
    * An image stream tag will be created as "scipy-notebook:latest" that will track the source image
    * A Docker build using source code from https://github.com/xaviervasques/OpenShift-ML-Online.git will be created
    * The resulting image will be pushed to image stream tag "openshift-ml-online:latest"
    * Every time "scipy-notebook:latest" changes a new build will be triggered

--> Creating resources ...
imagestream.image.openshift.io "scipy-notebook" created
imagestream.image.openshift.io "openshift-ml-online" created
buildconfig.build.openshift.io "openshift-ml-online" created
deployment.apps "openshift-ml-online" created
service "openshift-ml-online" created
--> Success
Build scheduled, use 'oc logs -f buildconfig/openshift-ml-online' to track its progress.
Application is not exposed. You can expose services to the outside world by executing one or more of the commands below:
'oc expose service/openshift-ml-online'
Run 'oc status' to view your app.
```

The application is deployed! The application now needs to be exposed to the outside world. As specified by the previous output, we can do this by executing the following command:

```
oc expose service/openshift-ml-online
```

```
[xavi@xavi:~$ oc expose service/openshift-ml-online
route.route.openshift.io/openshift-ml-online exposed
xavi@xavi:~$ ]
```

We can check the status:

```
oc status
```

We can also find the route of our API in the output and use it to get the expected output (the classification score of the SVM model on the test data):

```
curl http://openshift-ml-online-default.mycluster-par01-b-948990-d8688dbc29e56a145f8196fa85f1481a-0000.par01.containers.appdomain.cloud
```

```
[xavi@xavi:~$ curl http://openshift-ml-online-default.mycluster-par01-b-948990-d8688dbc29e56a145f8196fa85f1481a-0000.par01.containers.appdomain.cloud
{
  "score": 0.7
}
xavi@xavi:~$ ]
```

```
oc get pods --watch
```

NAME	READY	STATUS	RESTARTS	AGE
openshift-ml-online-1-build	0/1	Completed	0	113m
openshift-ml-online-68f9d84f94-nb6x1	1/1	Running	0	110m

We can also find all information about our service in the OpenShift web console.

In the perspective switcher, if we select Developer to switch to the Developer perspective and click on Topology.

At the bottom-right of the screen, the panel displays the public URL at which the application can be accessed. It can be seen under Routes. If we click on the link, we also obtain the expected result from our application.

We can create unsecured and secured routes using the web console or the CLI. Unsecured routes are the simplest to set up and represent the default configuration. However, if we want to offer security for connections to remain private, we can use the *create route* command and provide certificates and a key.

If we want to delete the application from OpenShift, we can use the *oc delete all* command:

```
oc delete all --selector app=myapp
```

In a case in which there is no Dockerfile, the OpenShift Source to Image (S2I) toolkit will create a Docker image. The source code language is auto-detected. OpenShift S2I uses a builder image and its sources to create a new Docker image that is deployed to the cluster.

It becomes simple to containerize and deploy a machine or deep learning application using Docker, Python, Flask, and OpenShift. When we begin to migrate workloads into OpenShift, the application is containerized into a container image that will be deployed in the testing and production environments, decreasing the number of missing dependencies and misconfiguration issues that we see when we deploy applications in the real world.

Real life is what it is all about. Machine learning operations requires cross-collaboration among data scientists, developers, and IT operations, which can be time-consuming in terms of coordination. Building, testing, and training ML/DL models on Kubernetes hybrid cloud platforms such as OpenShift allows consistent, at-scale application deployments and helps to deploy, update, and redeploy as often as needed in the production environment. The integrated DevOps CI/CD capabilities in Red Hat OpenShift allow us to automate the integration of our models with the process of development.

Further Reading

- <https://www.ibm.com/cloud/learn/docker>
- <https://mlinproduction.com>
- <https://docs.docker.com/engine/>
- <https://www.ibm.com/cloud/learn/docker>
- <https://mlfromscratch.com/deployment-introduction/#/>
- <https://mlinproduction.com/docker-for-ml-part-3/>
- <https://docs.docker.com/engine/reference/builder/>
- <https://mlinproduction.com/docker-for-ml-part-3/>
- <https://scikit-learn.org/stable/>
- <https://theaisummer.com/docker/>
- <https://www.fernandomc.com/posts/your-first-flask-api/>
- <https://mlinproduction.com/docker-for-ml-part-4/>
- <https://www.kdnuggets.com/2019/10/easily-deploy-machine-learning-models-using-flask.html>
- <https://medium.com/@fmirikar5119/ci-cd-with-jenkins-and-machine-learning-477e927c430d>
- <https://www.jenkins.io>
- <https://cloud.ibm.com/catalog/content/jenkins>
- <https://towardsdatascience.com/automating-data-science-projects-with-jenkins-8e843771aa02>
- <https://phoenixnap.com/blog/kubernetes-vs-docker-swarm>

<https://kubernetes.io/fr/docs/concepts/>
<https://kubernetes.io/fr/docs/tasks/tools/install-kubectl/>
<https://www.vagrantup.com/downloads>
<https://gitlab.com/xavki/presentations-kubernetes/-/tree/master>
<https://github.com/flannel-io/flannel>
<https://kubernetes.io/docs/concepts/workloads/controllers/job/>
<https://vsupalov.com/build-docker-image-clone-private-repo-ssh-key/>
<https://kubernetes.io/fr/docs/concepts/configuration/secret/>
<https://developer.ibm.com/technologies/containers/tutorials/scalable-python-app-with-kubernetes/>
<https://docs.docker.com/engine/install/ubuntu/>
<https://developer.ibm.com/technologies/containers/tutorials/scalable-python-app-with-kubernetes/>
<https://mlinproduction.com/k8s-jobs/>
<https://developer.ibm.com/technologies/containers/tutorials/scalable-python-app-with-kubernetes/>
<https://cloud.google.com/community/tutorials/kubernetes-mlops>
<https://developer.ibm.com/technologies/containers/tutorials/scalable-python-app-with-kubernetes/>
<https://cloud.google.com/community/tutorials/kubernetes-mlops>
<https://github.com/IBM/deploy-ibm-cloud-private>
<https://kubernetes.io/fr/docs/setup/pick-right-solution/#solutions-clés-en-main>
<https://www.ibm.com/cloud/architecture/tutorials/microservices-app-on-kubernetes?task=1>
<https://cloud.ibm.com/docs/containers?topic=containers-registry#other>
<https://cloud.ibm.com/docs/containers?topic=containers-nodeport>
<https://cloud.ibm.com/docs/containers>
<https://www.openshift.com>
<https://en.wikipedia.org/wiki/OpenShift>
<https://www.openshift.com/learn/topics/ai-ml>
<https://cloudowski.com/articles/10-differences-between-openshift-and-kubernetes/>
https://docs.openshift.com/containerplatform/4.5/cli_reference/developer_cli_odo/understanding-odo.html
<https://developer.ibm.com/technologies/containers/tutorials/scalable-python-app-with-kubernetes/>
<https://cloud.google.com/community/tutorials/kubernetes-mlops>
https://developer.ibm.com/tutorials/deploy-python-app-to-openshift-cluster-source-to-image/?mhsrc=ibmsearch_a&mhq=deploy%20python%20app%20to%20openshift%20cluster%20source%20to%20image
https://docs.openshift.com/enterprise/3.1/dev_guide/new_app.html
<https://github.com/jjasghar/cloud-native-python-example-app/blob/master/Dockerfile>
https://docs.openshift.com/container-platform/3.10/dev_guide/routes.html
<https://docs.okd.io/3.11/minishift/openshift/exposing-services.html>
<https://docs.openshift.com/container-platform/3.10/architecture/networking/routes.html#secured-routes>
<https://docs.openshift.com/container-platform/4.7/installing/index.html#ocp-installation-overview>
<https://www.openshift.com/blog/serving-machine-learning-models-on-openshift-part-1>
<https://developer.ibm.com/technologies/containers/tutorials/deploy-python-app-to-openshift-cluster-source-to-image/>
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html

Conclusion: The Future of Computing for Data Science?



Photo by Annamária Borsos

Classical computing has experienced remarkable progress guided by Moore's law. This law states that every two years, we double the number of transistors in a processor and at the same time increase performance by or reduce costs by twofold. This pace has slowed down over the past decade, forcing a transition. We must rethink information technology (IT) and in particular move toward heterogeneous system architectures with specific accelerators in order to meet the need for performance. The progress that has been made in raw computing power has nevertheless brought us to a point at which biologically inspired computing models are now highly regarded as the state of the art.

Artificial intelligence (AI) is an area that brings opportunities for progress but also challenges alongside. The capabilities of AI have greatly increased in their ability to interpret and analyze data. AI is also demanding in terms of computing power because of the complexity of workflows. At the same time, AI can also be applied to the management and optimization of entire IT systems.

In parallel with conventional or biologically inspired accelerators, programmable quantum computing is emerging, thanks to several decades of investment in research to overcome traditional physical limitations. This new era of computing will potentially have the capacity to make calculations that are not possible today with conventional computers. Future systems will need to integrate quantum computing capabilities to perform specific calculations. Research is advancing rapidly. IBM made programmable quantum computers available to the cloud for the first time in May 2016, and has announced its ambition to double the quantum volume each year, called "Gambetta's law."

The cloud is also an element that brings considerable challenges and opportunities in data science, and it has an important role to play. The data centers of tomorrow will be piloted by the cloud and equipped with heterogeneous systems that will run heterogeneous workloads in a secure manner. Data will no longer be centralized or decentralized but instead will be organized in hubs. Storage systems are also full of challenges to improve, not only in terms of availability, performance, and management but also regarding data fidelity. We have to design architectures to allow for the extraction of more and more complex and often regulated data, which poses multiple challenges, in particular security, encryption, confidentiality, and traceability.

The future of computing, as described by Dario Gil and William Green, will be built with heterogeneous systems made up of classical computing, called binary or bit systems, biologically inspired computing, and quantum computing, called quantum or qubit systems. These heterogeneous components will be orchestrated and deployed in a hybrid cloud architecture that masks complexity while allowing the secure use and sharing of private and public systems and data.

Binary Systems

The first binary computers were built in the 1940s: Colossus (1943) and then the Electronic Numerical Integrator and Computer (ENIAC, IBM, 1945). Colossus was designed to decrypt secret German messages, and the ENIAC was designed to calculate ballistic trajectories. The ENIAC was the first fully electronic system built to be Turing-complete: it can be reprogrammed to solve, in principle, all computational problems. The ENIAC was programmed by women called the “ENIAC women.” The most famous of them were Kay McNulty, Betty Jennings, Betty Holberton, Marlyn Wescoff, Frances Bilas, and Ruth Teitelbaum. These women had previously performed ballistic calculations on mechanical desktop computers for the military. The ENIAC weighed 30 tons, occupied an area of 72 m², and consumed 140 kW.

Regardless of the task performed by a computer, the underlying process is always the same: an instance of the task is described by an algorithm that is translated into a sequence of 0s and 1s to give rise to execution in the processor, memory, and input/output devices of the computer. This is the basis of binary calculation, which in practice is based on electrical circuits provided with transistors that can be in two modes: “ON,” allowing the current to pass, and “OFF,” such that the current does not pass. From these 0s and 1s, over the past 80 years we have developed a classical information theory constructed from Boolean operators (XAND, XOR, etc.), words (bytes), and a simple arithmetic based on the following operations: $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, $1 + 1 = 0$ (with restraint), and checking whether $1 = 1$, $0 = 0$, and $1 \neq 0$. From these operations, it is possible to build much more complex operations, which the most powerful computers can perform millions of billions of times per second. All this has become so “natural” that we have completely forgotten that each transaction on a computer server, a PC, a calculator, or a smartphone breaks down into these basic binary operations. In a computer, these 0s and 1s are contained in “BInary digiTs,” or “bits,” which represent the smallest amount of information contained in a computer system. The electrical engineer and mathematician Claude Shannon (1916–2001) was one of the founding fathers of information theory. For 20 years, Shannon worked at the Massachusetts Institute of Technology (MIT); in addition to his academic activities, he worked at Bell Laboratories. In 1949, he married Madame Moore. During the Second World War, Shannon worked for the American secret service in cryptography to locate messages hidden in German codes. One of his essential contributions concerns the theory of signal transmission. It is in this context that he developed an information theory, in particular by understanding that any data, even voice or images, can be transmitted using a sequence of 0s and 1s.

The binary systems used by conventional computers appeared in the middle of the twentieth century, when mathematics and information were combined in a new way to form information theory, launching both the computer industry and telecommunications. The strength of the binary system lies in its simplicity and reliability. A bit is either 0 or 1, a state that can be easily measured, calculated, communicated, or stored. This powerful theory has allowed us to build the systems that are running critical workloads around the world today. Thanks to this method, various calculations and data storage systems have emerged, up to the storage of digital data on a DNA molecule.

Today, we have examples of binary systems with incredible possibilities. An example is the mainframe today called IBM Z. At the time I am writing this book, an IBM Z processor single-chip module (SCM) is using silicon-on-insulator

(SOI) technology at 14 nm. It contains 9.1 billion transistors, and there are 12 cores per PU SCM at 5.2 GHz. This technology allows a single system to be able to process 19 billion encrypted transactions per day and 1000 billion web transactions per day. The IBM Zs installed in the world today process 87% of bank card transactions and 8 trillion payments per year.

We can also cite two of the most powerful computers in the world, Summit and Sierra. They are located in the Oak Ridge Laboratory in Tennessee and the National Lawrence Laboratory in Livermore, California, respectively. These computers help model supernovas or new materials, explore solutions against cancer, and study genetics and the environment. Summit is capable of delivering a computer power of 200 petaflops with a storage capacity of 250 petabytes. It is composed of 9216 IBM Power9 CPUs, 27,648 NVIDIA Tesla GPUs, and a network communication of 25 gigabytes per second between nodes. Despite all of these abilities, even the most powerful computer in the world, equipped with GPU accelerators, cannot calculate everything.

Today, this type of technology (high-performance computing) is essential for medical research, as we saw during the COVID-19 crisis. We can take the example of using the power of supercomputers with the HPC COVID-19 consortium (<https://covid19-hpc-consortium.org>). This is a public-private effort initiated by several institutions, including IBM, aimed at making the power of supercomputers available to researchers working on projects related to COVID-19 to help them identify potential short-term therapies for patients affected by the virus.

Together, the Consortium has helped support many research projects, including understanding how long respiratory droplets persist in the air. They found that droplets from breathing stay in the air much longer than previously thought, due to their small size compared to droplets from coughs and sneezes. Another project concerns research into the reuse of drugs for potential treatments. A project by a Michigan State University team has examined data from approximately 1600 FDA-approved drugs to determine whether there are possible combinations that could help treat COVID-19. They have found at least two drugs approved by the FDA that appear to be promising: proflavin, a disinfectant against many bacteria, and chloroxin, another antibacterial drug.

As we may have thousands of candidate molecules for potential therapeutic treatment, the use of accelerated systems and deep learning allows the best matches to be filtered in order to provide a selection of chemical compounds capable of attaching to pathogen proteins. By doing this, the speed of the drug design process could be increased considerably. AI will also help researchers to better profile the protein–protein interactions involved in the development of pathologies as well as to better understand the dynamics of infections in human cells. Thanks to this innovative approach, the development cycle of therapeutic treatment could be accelerated, potentially going from several years to a few months or even a few weeks.

Computers, smartphones and their applications, and the internet that we use in our everyday lives work with 0s and 1s. The binary system coupled with Moore's law, a 50-year heritage, has made it possible to build systems that are robust and reliable. For 50 years, we have seen incremental, linear evolution to produce performance gains. The next few years will bring further innovations to produce performance gains, particularly in terms of materials, control processes, or etching methods, including three-dimensional transistors, extreme ultraviolet lithography, or new materials such as hafnium or germanium. Binary systems therefore continue to evolve and will play a central role in the data center of tomorrow.

Recently, IBM took a major step forward in chip technology by manufacturing the first 2-nm chip, packing 50 billion transistors onto a fingernail-sized chip. This architecture can help processor manufacturers improve performance by 45% with the same amount of power as today's 7-nm chips, producing the same level of performance using 75% less power. Mobile devices with 2-nm-based processors could have up to four times the battery life of those with 7-nm chipsets. Laptops would benefit from an increase in the speed of these processors, while autonomous vehicles may detect and respond to objects faster. This technology will benefit data center energy efficiency, space exploration, AI, 5G and 6G communication, and quantum computing.

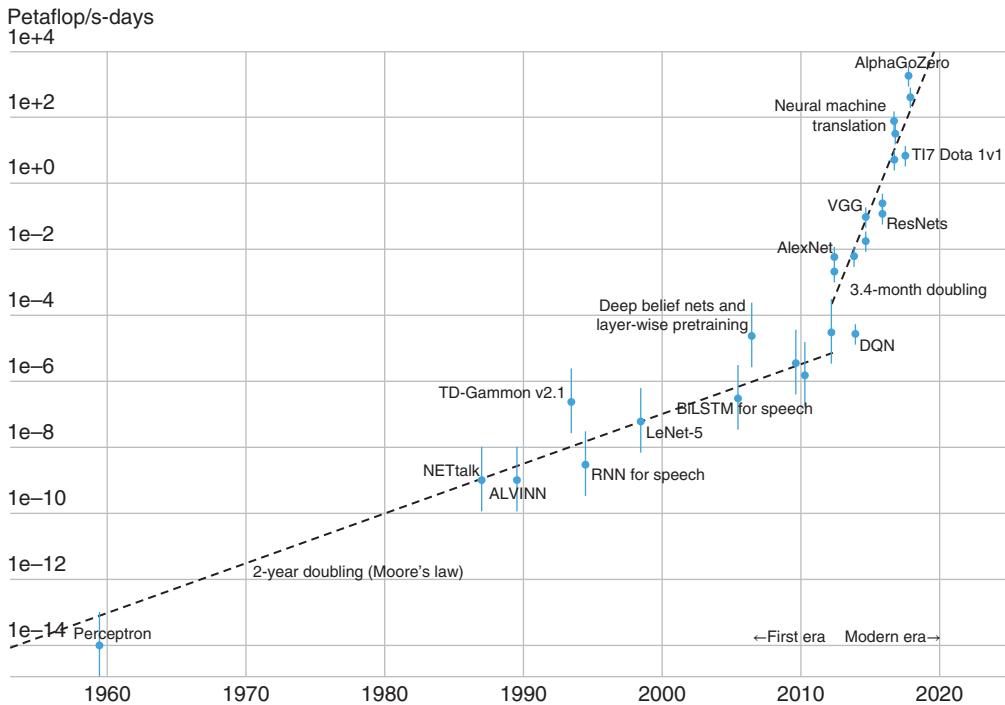
Despite comments about the limitations of Moore's law and the bottlenecks of current architectures, innovations continue, and binary systems will play a central role in the data center of tomorrow. Nevertheless, some challenges cannot be addressed only with binary computing. For these challenges, we need to rethink computing with new approaches, drawing inspiration from natural processes such as biology and physics.

Biologically Inspired Systems

DNA is also a source of inspiration. In 2017, a team published in the journal *Science* the capacity to store digital data on a DNA molecule. They were able to store an operating system, a French film from 1895 (*L'Arrivée d'un train à La Ciotat* by Louis Lumière), a scientific article, a photograph, a virus, and a \$50 gift card in DNA strands and retrieve the data without errors.

Indeed, a DNA molecule is intended to store information by nature. Genetic information is encoded in four nitrogenous bases that make up a DNA molecule (A, C, T, and G). Today, it is possible to transcribe digital data into a new code. DNA sequencing then makes it possible to read the stored information. Encoding is automated through software. A DNA molecule is 3 billion nucleotides (nitrogenous bases). In 1 g of DNA, 215 petabytes of data can be stored; it would be possible to store all the data created by humans in one room using DNA. In addition, DNA can theoretically keep data in perfect condition for an extremely long time. Under ideal conditions, it is estimated that DNA could still be deciphered after several million years, thanks to “longevity genes.” DNA can withstand the most extreme weather conditions. The main weak points today are high costs and processing times that can be extremely long.

The term AI appeared as such in 1956. Several American researchers, including John McCarthy, Marvin Minsky, Claude Shannon, and Nathan Rochester of IBM, which had been very advanced in research that used computers for other than scientific calculations, met at the University of Dartmouth in the United States. Three years after the Dartmouth seminar, the two fathers of AI, McCarthy and Minsky, founded the AI lab at MIT. There was considerable investment, with too much ambition, in imitating the human brain, and much of that hope was not realized at the time. The promises were broken. A more pragmatic approach appeared in the 1970s and 1980s, which saw the emergence of machine learning and the reappearance of neural networks in the late 1980s. This more pragmatic approach, an increase in computing power, and an explosion of data have made it possible for AI to be present in all areas today; it is a transversal subject. The massive use of AI poses some challenges, such as the need to label the data at our disposal. The problem with automation is that it requires considerable manual work. AI needs education, and this is performed by tens of thousands of workers around the world, which does not really appear to be a futuristic vision. Another challenge is the need for computing power. AI needs to be trained, and AI has become more and more greedy for this process in terms of calculations. Training requires a doubling of computing capacity every 3.5 months.



Source: AI and Compute, OpenAI, <https://openai.com/blog/ai-and-compute/#fn1>.

Several approaches are currently being used or envisioned. For example, in the Summit supercomputer today, the calculation of certain workloads is deported to accelerators such as GPUs. There are others, such as field-programmable gate arrays (FPGAs) or “programmable logic networks,” which can realize desired digital functions. The advantage is that the same chip can be used in many different electronic systems.

Progress in the field of neuroscience will allow the design of processors directly inspired by the brain. The way our brain transmits information is not binary, and it is thanks to Santiago Ramón y Cajal (1852–1934), Spanish histologist and neuroscientist and winner of the Nobel Prize in physiology or medicine in 1906 with Camillo Golgi, that we now understand better the architecture of the nervous system. Neurons are cellular entities separated by fine spaces (synapses); they are not fibers of an unbroken network. The axon of a neuron transmits nerve impulses, action potentials, to target cells. The next step in developing new types of AI-inspired and brain-inspired processors is to think differently about how we compute today. Today, one of the major performance problems is the movement of data between the different components of the von Neumann architecture: processor, memory, and storage. It is therefore imperative to add analog accelerators. What dominates numerical calculations today, and deep learning calculations, in particular, is floating-point multiplication. One of the methods envisioned as an effective means of gaining computational power is to go back in the computer history by reducing precision, also called approximate calculation. For example, 16-bit precision engines are more than fourfold smaller than 32-bit precision engines. This gain increases performance and energy efficiency. In simple terms, in approximate calculation, we can make a compromise by exchanging numerical precision for the efficiency of calculation. Certain conditions are nevertheless necessary, such as developing algorithmic improvements in parallel to guarantee iso-precision. IBM has recently demonstrated the success of this approach with 8-bit floating-point numbers, using new techniques to maintain the accuracy of gradient calculations and updating weights during backpropagation. Likewise, for inference by a model resulting from deep learning algorithm training, the unique use of whole arithmetic on four or two precision bits achieves accuracy comparable to a range of popular models and datasets. This progression will lead to a dramatic increase in computing capacity for deep learning algorithms over the next decade.

Analog accelerators are another way of avoiding the bottleneck of von Neumann’s architecture. The analog approach uses non-volatile programmable resistive processing units (RPUs) that can encode the weights of a neural network. Calculations such as matrix or vector multiplication or the operations of matrix elements can be performed in parallel and in constant time, without movement of the weights. However, unlike digital solutions, analog AI is more sensitive to the properties of materials and intrinsically sensitive to noise and variability. These factors must be addressed by architectural solutions, new circuits, and new algorithms. For example, analogous non-volatile memories (NVMs) can effectively speed up backpropagation algorithms. By combining long-term storage in phase-change memory (PCM) devices, quasi-linear updating of conventional complementary metal-oxide-semiconductor (CMOS) capacitors, and new techniques to eliminate device-to-device variability, significant results have begun to emerge for the calculation of Deep Neural Network.

The research has also embarked on a quest to build a chip directly inspired by the brain. In an article published in *Science*, IBM and its university partners describe a processor called SyNAPSE, which is made of a million neurons. The chip consumes only 70 mW and is capable of 46 billion synaptic operations per second per watt, literally a synaptic supercomputer that can be held in a hand. We have moved from neuroscience to supercomputers, new computing architectures, programming languages, algorithms, and applications, and now a new chip called TrueNorth. TrueNorth is a neuromorphic CMOS integrated circuit produced by IBM in 2014. It is a many-core processor network, with 4096 cores, each having 256 programmable simulated neurons for a total of just over 1 million neurons. In turn, each neuron has 256 programmable synapses, allowing the transport of signals. Therefore, the total number of programmable synapses is slightly more than 268 million. The number of basic transistors is 5.4 billion. Because memory, computation, and communication are managed in each of the 4096 neurosynaptic cores, TrueNorth bypasses the bottleneck of the von Neumann architecture and is highly energy-efficient. It has a power density 1/10,000 that of conventional microprocessors.

Quantum Systems: Qubits

In an article published in *Nature*, IBM physicists and engineers have described the feat of writing and reading data in an atom of holmium, a rare-earth element. This is a symbolic step forward but proves that this approach works and that we might one day have atomic data storage. To illustrate what it means, imagine that we can store the entire iTunes library of 35 million songs on a device the size of a credit card. In the paper, the nanoscientists demonstrate the ability to read and

write one bit of data on one atom. For comparison, today's hard disk drives use 100,000 to 1 million atoms to store a single bit of information.

Of course, we cannot avoid discussing quantum computing. "Quantum bits" – or qubits – combine physics with information and are the basic units of a quantum computer. Quantum computers use qubits in a computational model based on the laws of quantum physics. Properly designed quantum algorithms will be capable of solving problems of great complexity by exploiting quantum superposition and entanglement to access an exponential state space and then amplifying the probability of calculating the correct response by constructive interference. It was at the beginning of the 1980s, from the encouragement of the physicist and Nobel laureate Richard Feynman, that the idea of the design and development of quantum computers was born: Whereas a classical computer works with bits of values 0 or 1, the quantum computer uses the fundamental properties of quantum physics and is based on qubits. With this technological progress, quantum computing opens the way to the processing of computer tasks whose complexity is beyond the reach of current computers. But let us start from the beginning.

At the beginning of the twentieth century, the theories of so-called classical physics were unable to explain certain problems observed by physicists. They therefore needed to be reformulated and enriched. Under the impetus of scientists, they evolved initially toward a "new mechanics," which became "wave mechanics" and finally "quantum mechanics." Quantum mechanics is the mathematical and physical theory that describes the fundamental structure of matter and the evolution over time and space of the phenomena of the infinitely small. An essential notion of quantum mechanics is the duality of the "wave–particle." Until the 1890s, physicists had considered that the world is composed of two types of objects or particles: on the one hand those that have a mass (such as electrons, protons, neutrons, atoms, and others) and, on the other hand, those that do not (such as photons, waves, and others). To the physicists of the time, these particles were governed by the laws of Newtonian mechanics for those with a mass and by the laws of electromagnetism for waves. We therefore had two theories of physics to describe two different types of objects. Quantum mechanics invalidates this dichotomy and introduces the fundamental idea of particle–wave duality. Particles of matter or waves must be treated with the same laws of physics. The theory of wave mechanics became quantum mechanics a few years later. Big names are associated with the development of quantum mechanics, including Niels Bohr, Paul Dirac, Albert Einstein, Werner Heisenberg, Max Planck, Erwin Schrödinger, and many others. Planck and Einstein, being interested in the radiation emitted by a heated body and in the photoelectric effect, were the first to understand that the exchanges of light energy could only be performed by "packets." Moreover, Einstein obtained the Nobel Prize in physics following the publication of his theory on the quantified aspect of energy exchanges in 1921. Bohr extended the quantum postulates of Planck and Einstein from light to matter by proposing a model reproducing the spectrum of the hydrogen atom. He obtained the Nobel Prize in physics in 1922 by defining a model of the atom that could dictate the behavior of quanta of light. Passing from one energy level to a lower one, an electron exchanges a quantum of energy. Step by step, rules were found to calculate the properties of atoms, molecules, and their interactions with light.

From 1925 to 1927, a series of works by several physicists and mathematicians gave substance to two general theories applicable to these problems:

- The wave mechanics of Louis de Broglie and especially of Schrödinger.
- The matrix mechanics of Werner Heisenberg, Max Born, and Pascual Jordan.

These two types of mechanics were unified by Schrödinger from the physical point of view and by John von Neumann from the mathematical point of view. Finally, Dirac formulated the synthesis or rather the complete generalization of these two mechanics, which today we call quantum mechanics. The fundamental equation of quantum mechanics is the Schrödinger equation.

Quantum computing began with a small, now famous, conference on the physics of computing in 1981, jointly organized by IBM and MIT. Feynman challenged computer scientists to invent a new type of computer based on quantum principles to better simulate and predict the behavior of actual material: "I'm not satisfied with all the analyses that go with just classical theory, because nature is not classic, damn it, and if you want to make a simulation of nature, you'd better do it with quantum mechanics"

Matter, Feynman explained, is made of particles, such as electrons and protons, that obey the same quantum laws that would govern the operation of this new computer. Since then, scientists have addressed Feynman's double challenge: understanding the capabilities of a quantum computer and figuring out how to build one. Quantum computers will be very different from today's computers, not only in what they look like and how they are made but also, and more importantly, in what they can do. We can also quote a famous phrase from Rolf Landauer, a physicist who worked at

IBM: “Information is physical.” Computers are, of course, physical machines. It is therefore necessary to take into account the energy costs generated by calculations and the reading and recording of bits of information as well as energy dissipation in the form of heat. In a context in which the links between thermodynamics and information were the subject of many questions, Landauer sought to determine the minimum amount of energy necessary to manipulate a single bit of information in a given physical system. There should therefore be a limit, today called the Landauer limit and discovered in 1961, that specifies that any computer system is obliged to dissipate a minimum amount of heat and therefore consume a minimum amount of electricity. This research was fundamental because it showed that any computer system has a minimum thermal and electrical threshold that cannot be exceeded. Thus, we will reach the minimum consumption of a computer chip; it will not be able to release less energy. This concept is not relevant in classical systems, but scientists explain that this limit will be especially important in designing quantum chips. Recent work by Charles Henry Bennett at IBM has consisted of a re-examination of the physical basis of information and the application of quantum physics to the problems of information flows. His work has played a major role in the development of an interconnection between physics and information.

For a quantum computer, the qubit is the basic entity that represents, like the bit, the smallest entity allowing manipulation of information. It has two fundamental properties of quantum mechanics: superposition and entanglement.

A quantum object (on a microscopic scale) can exist in an infinite number of states (as long as one does not measure this state). A qubit can therefore exist in any state between 0 and 1. Qubits can take both the value 0 and the value 1, or rather “a certain amount of 0 and a certain amount of 1,” as a combination linear of two states denoted $|0\rangle$ and $|1\rangle$, with the coefficients α and β . Thus, whereas a classic bit describes only two states (0 or 1), the qubit can represent an infinite number. This is one of the potential advantages of quantum computing from the point of view of information theory. We can obtain an idea of the superposition of states using the analogy of the lottery ticket: A lottery ticket is either winning or losing once we know the outcome of the game. However, before the draw, this ticket was neither a winner nor a loser. It only had a certain probability of being a winner and a certain probability of being a loser; it was therefore both a winner and a loser at the same time. In the quantum world, all the characteristics of particles can be subject to this indeterminacy. For example, the position of a particle is uncertain. Before measurement, the particle is neither at point A nor at point B; it has a certain probability of being at point A and a certain probability of being at point B. However, after measurement, the state of the particle is well defined: It is either at point A or at point B.

Another amazing property of quantum physics is entanglement. When we consider a system composed of several qubits, they can sometimes “link their destiny,” that is to say not remain independent of each other even if they are separated in space (while classical bits are completely independent of each other). This phenomenon is called quantum entanglement. If we consider a system of two entangled qubits, then the measurement of the state of one of these two qubits gives us an immediate indication of the result of an observation on the other qubit.

To illustrate this property, we can use another analogy: We can imagine two light bulbs, each in two different houses. By entangling them, it becomes possible to know the state of one bulb (on or off) by simply observing the second, because the two would be immediately linked, or entangled, even if the houses are very far from each other.

The entanglement phenomenon makes it possible to describe correlations between qubits. If we increase the number of qubits, the number of these correlations increases exponentially: For N qubits, there are $2N$ correlations. This property gives the quantum computer the possibility of carrying out manipulations on enormous quantities of values, quantities beyond the reach of a conventional computer.

The uncertainty principle discovered by Werner Heisenberg in 1927 tells us that whatever our measuring tools, we are unable to precisely determine both the position and the speed of a quantum object (at the atomic scale). Either we know exactly where the object is, and the speed will seem to fluctuate and become blurry, or we have a clear idea of the speed, but its position will escape us.

In his book *La quantique autrement: garanti sans équation!*, Julien Bobroff describes the quantum experiment in three acts:

The first moment is before the quantum object behaves like a wave. The Schrödinger equation allows us to accurately predict how the latter spreads, how fast, in what direction, whether it spreads or contracts. Then, it is decoherence that makes its appearance. Decoherence happens extremely quickly, almost instantaneously. It is at this precise moment that the wave comes into contact with a measuring tool (e.g., a fluorescent screen). This wave is forced to interact with the particles that make up this device. This is the moment when the wave is shrinking. The last step is the random choice among all the possible states. The draw is related to the shape of the wave function at the time of measurement. In fact, only the shape of the wave function at the end of the first act dictates how likely it is to appear here or there.

Another phenomenon of quantum mechanics is the tunnel effect. Bobroff uses the example of a tennis ball. If we throw a tennis ball against a wall, it will come back to us! In the world of quantum, if the ball is a quantum wave function, it will only partially bounce against a barrier. A small part can tunnel through to the other side. This implies that if the particle is measured, it will sometimes materialize on the left of the wall, sometimes on the right.

A quantum computer uses the laws of quantum mechanics to make calculations. It has to be under certain conditions, sometimes extreme, such as immersing a system in liquid helium to reach temperatures close to absolute zero, or -273.15°C .

Building a quantum computer relies on the ability to develop a computer chip on which qubits are engraved. From a technological point of view, there are several ways of constituting qubits; they can be made of atoms, photons, electrons, molecules, or superconductive metals. In most cases, a quantum computer needs extreme conditions to operate such as temperatures close to absolute zero. The choice of IBM, for example, is to use superconducting qubits constructed with aluminum oxides (this technology is also called transmons qubits). As mentioned above, to allow quantum effects (superposition and entanglement), the qubits must be cooled to a temperature as close as possible to absolute zero (i.e., approximately -273°C). At IBM, this operating threshold is approximately 20 mK! IBM demonstrated the ability to design a single qubit in 2007 and 2016 and announced the availability in the cloud of the first operational physical system with five qubits and the development environment “QISKit” (Quantum Information Science Kit), allowing the design, testing, and optimization of algorithms for commercial and scientific applications. The “IBM Q Experience” initiative is a first in the industrial world. At the time of this writing, IBM now has several quantum computers, including a 127-qubit system, and has recently published its roadmap.

The number of qubits will progressively increase, but this is not enough. In the race to develop quantum computers, other components are essential beyond qubits. We can speak of “quantum volume” as a relevant measure of performance and technological progress. Other measures have also been offered by companies and laboratories. We can define “quantum advantage” as the point at which quantum computing applications offer a significant and demonstrable practical advantage that exceeds the capabilities of conventional computers. The concept of “quantum volume” was introduced by IBM in 2017 and is beginning to spread to other manufacturers. Quantum volume is a measure that determines the power of a quantum computer system, taking into account gate and measurement errors, crosstalk of the device, connectivity of the device, and efficiency of the circuit compiler. The quantum volume can be used for any noisy intermediate-scale quantum (NISQ) computer system based on gates and circuits. For example, if we lower the error rate of $\times 10$ without adding extra qubits, we can have a quantum volume increase of $500\times$. On the contrary, if we add 50 additional qubits but do not decrease error rates, we will have an increased quantum volume of $0\times$. Adding qubits is not everything.

The challenges researchers face today are technological, such as stability over time. When we run a quantum algorithm on a real quantum computer, there are many externalities that can disrupt the quantum state of the program, which is already fragile. Another technological challenge concerns the quantity of qubits. Every time we increase the capacity of a quantum computer by one qubit, we reduce its stability. Another challenge is that we will be forced to rethink the entirety of the algorithms we know to adapt them to quantum computing.

And, of course, we need to be able to run tasks on these machines, which is why IBM has developed the QISKit programming library. This open-source library for the Python language is available at qiskit.org. Its development is very active, and all contributors, including IBM, regularly update the functionality of this programming environment.

Quantum computers will be added to conventional computers to address problems that are unsolved today. For example, conventional computers can calculate complex problems that a quantum computer cannot, and there are problems that both classical and quantum computers can solve. Finally, there are challenges that a conventional computer cannot solve but that a quantum computer can address. Many applications are possible in the fields of chemistry, materials science, machine learning, and optimization.

For example, it is difficult for a classical computer to calculate the energy of the caffeine molecule (with 24 atoms) exactly (that is to say without any approximation), it is a very complex problem. We would need approximately 10^{48} bits to represent the energy configuration of a single caffeine molecule at a time t . That is almost the number of atoms contained on earth, which is 10^{50} . However, we believe it is possible to perform this calculation with 160 qubits. Today, quantum computers are used to address simple chemistry problems, with a small number of atoms, but the objective is of course to be able to address much more complex molecules.

But that is not the only limitation. To provide a simple illustration, we can consider the so-called itinerant seller problem, or the problem of delivery by delivery trucks as a modern example. If we want to choose the most efficient route for a truck to deliver packages to 5 addresses, there are 12 possible routes, so it is possible to identify the best one. However, as we add

more addresses, the problem becomes exponentially more difficult – by the time we have 15 deliveries, there are over 43 billion possible routes, making it virtually impossible to find the best. For example, in 71 cities, the number of candidate paths is greater than 5×10^{80} .

Currently, quantum computing is suitable for certain algorithms such as optimization, machine learning, or simulation. With these types of algorithms, use cases apply in several industrial sectors. Financial services such as portfolio risk optimization, fraud detection, health (drug research, protein study, etc.), supply chains and logistics, chemicals, research for new materials, and oil exploration are all areas that will be primarily impacted. We can also address the future of medical research with quantum computing, which should eventually allow the synthesis of new therapeutic molecules. In addition, if we are to meet the challenge of climate change, we need to solve many problems such as designing better batteries, finding less energy-intensive ways to grow our food, and planning our supply chains to minimize transport. Solving these problems effectively requires radically improved computational approaches in areas such as chemistry and materials science as well as in the fields of optimization and simulation – areas in which classical computing faces serious limitations.

For a conventional computer, considering the product of two numbers and obtaining the result is a very simple operation: $7 \times 3 = 21$ or $6739 \times 892,721 = 6,016,046,819$. This remains true even for very large numbers. But the opposite problem is much more complex. Knowing a large number N , it is more complicated to find P and Q such that $P \times Q = N$. This difficulty forms the basis of current cryptographic techniques. Yet, it is estimated that a similar problem that would last 1025 days on a conventional computer could be resolved in a few tens of seconds on a quantum machine. We speak in this case of exponential acceleration. With quantum computers, we can approach problems in an entirely new way by taking advantage of entanglement, superposition, and interference: modeling the physical procedures of nature, performing many more scenario simulations, and finding better optimization solutions or models in AI and ML processes. There are many cases of optimization that are eligible for quantum computing, including those in the fields of logistics (shortest path), finance (risk assessment, evaluation of asset portfolios), marketing, industry, and the design of complex systems. The field of AI is also an active research field, and learning methods for artificial neural networks are beginning to emerge; thus, it is the whole of human activities related to the processing of information that is potentially relevant in the future of quantum computing. The domain of cybersecurity and cryptography is also a subject of attention. The Shor algorithm was demonstrated over 20 years ago, and it could weaken the encryption commonly used on the internet; we will have to wait until quantum machines are powerful enough to process this type of calculation. On the other hand, encryption solutions beyond the reach of this algorithm have already been demonstrated. Quantum technology itself will also provide solutions to protect data. Therefore, the field of quantum technologies, and quantum computing in particular, is considered strategic.

We can find many use cases in banks and financial institutions to improve trading strategies and management of client portfolios and to better analyze financial risks. A quantum algorithm in development, for example, could potentially provide quadratic acceleration when using derivative pricing – a complex financial instrument that requires 10,000 simulations to be valued on a conventional computer but would only require 100 quantum operations on a quantum device.

Another use case for quantum computing is the optimization of trading. It will be possible for banks to accelerate portfolio optimizations such as Monte Carlo simulations. The simulation of buying and selling of products (trading) such as derivatives can be improved using quantum computing. The complexity of trading activities in financial markets is skyrocketing. Investment managers struggle to integrate real constraints such as market volatility and changes in client life events, into portfolio optimization. Currently, rebalancing of investment portfolios to follow market movements is strongly impacted by calculation constraints and transaction costs. Quantum technology could help reduce the complexity of today's business environments. The combinatorial optimization capabilities of quantum computing can enable investment managers to improve portfolio diversification, to rebalance portfolio investments to respond to market conditions and investor objectives more precisely, and to streamline more cost-effective transaction settlement processes. Machine learning is also used for portfolio optimization and scenario simulation. Banks and financial institutions such as hedge funds are increasingly interested because they see it as a way to minimize risks while maximizing gains with dynamic products that can adapt according to new, simulated data. Personalized finance is also an area that is being explored. Customers demand personalized products and services that can quickly anticipate changing needs and behaviors. Small- and medium-sized financial institutions can lose customers because of offers that do not favor the customer experience. It is difficult to create analytical models using behavioral data quickly and precisely enough to target and predict the products that customers need in nearly real time.

A similar problem exists in detecting fraud to find patterns of unusual behavior. Financial institutions are estimated to lose between \$10 billion and \$40 billion in revenue annually due to fraud and poor data-management practices. For customer targeting and forecast modeling, quantum computing could be a game-changer. The data-modeling capabilities of

quantum computers are expected to be superior in finding models, performing classifications, and making predictions that are not possible today with conventional computers due to the challenges of complex data structures.

Another use case in the world of finance is risk analysis. Risk analysis calculations are difficult because it is difficult to analyze many scenarios. Compliance costs are expected to more than double in the coming years. Financial services institutions are under increasing pressure to balance risk, hedge positions more effectively, and perform a wider range of stress tests to comply with regulatory requirements. Monte Carlo simulations, the preferred technique for analyzing the impact of risk and uncertainty in financial models, are currently limited by the scaling of the estimation error. Quantum computers have the potential to sample data differently by testing more results with greater accuracy, providing quadratic acceleration for these types of simulations.

Molecular modeling may allow for discoveries such as more efficient lithium batteries. Quantum computing will empower modeling of atomic interactions much more precisely and at much larger scales; we can use again the example of the caffeine molecule. New materials will be able to be used everywhere, whether in consumer products, cars, batteries, or other places. Quantum computing will allow molecular orbit calculations to be performed without approximation. Other applications are the optimization of a country's electricity network, more predictive environmental modeling, and the search for energy sources with lower emissions.

Aeronautics will also be a source of use cases. For each landing of an airplane, hundreds of operations are performed: crew change, refueling, cabin cleaning, baggage delivery, and inspections; each transaction has suboperations. Refueling requires an available tanker, a truck driver, and two people to add fuel; it must also be ensured in advance that the tanker is full. With hundreds of aircraft landing and flights that are sometimes delayed, the problem becomes more and more complex. It is then necessary to recalculate all of these factors for all planes in real time.

Electric vehicles have a weakness, namely the capacity and speed of charging their batteries. A breakthrough in quantum computing made by researchers from IBM and the automobile manufacturer Daimler could help meet this challenge. Daimler is very interested in the impact of quantum computing to optimize transport logistics and to predict future materials for electric mobility, in particular the next generation of batteries. There is every reason to hope that quantum computers will yield results in the years to come to accurately simulate the chemistry of battery cells, aging processes, and the performance limits of battery cells.

The problem of the commercial traveler can be extended to many fields such as energy, telecommunications, logistics, production chains, and resource allocation. For example, in sea freight, there is great complexity in the management of containers from start to finish. Loading, conveying, delivering, and then unloading in several ports in the world is a multi-parameter problem that can be addressed by quantum computing.

A better understanding of the interactions between atoms and molecules will make it possible to discover new drugs. Detailed analysis of DNA sequences will help detect cancer earlier by developing models that can determine how diseases develop. The advantage of quantum computing will be the ability to analyze the behavior of molecules in detail and on a scale never reached before. Chemical simulations will allow the discovery of new drugs and better prediction of protein structures, scenario simulations will better predict the risks of a disease or its spread, the resolution of optimization problems will optimize the chains of distribution of drugs, and finally the use of AI will speed up diagnoses and analyze genetic data more precisely.

The data center of tomorrow will be made of heterogeneous systems, which will run heterogeneous workloads. The systems will be located as close as possible to the data. These heterogeneous systems will be equipped with binary, biologically inspired, and quantum accelerators. These architectures will be the foundations for addressing challenges. Like an orchestra conductor, the hybrid cloud will make it possible to set these systems to music thanks to a layer of security and intelligent automation.

Final Thoughts

As you can see, there are many things coming from the hardware side that will certainly allow machine learning to progress dramatically. Today, we are at the beginning of broad AI and foundation models. We define foundation models as models that are trained on large datasets (usually using large-scale self-supervision) and that can be adapted to a wide range of downstream tasks. The rise of these models (e.g., BERT, DALL-E, GPT-3) represents a paradigm shift. Models are injected with various data (text, audio, video, images, structured data, etc.); the models train on this data and can then perform

functions such as answering questions (“Is AI dangerous for humans?”), writing texts (e.g., a philosophical text on a given subject), generating code, performing object recognition, translating, and other tasks. Training these models requires a large amount of computing power, with a cost of several million for the creation and training of a single basic model. That is why progress in hardware will also contribute significantly to the progress of AI.

I hope you have enjoyed reading and learning from this book as much as I have enjoyed writing it.

Further Reading

- Ambrogio, S., Narayanan, P., Tsai, H. et al. (2018). Equivalent-accuracy accelerated neural-network training using analog memory. *Nature* 558: 60–67.
- Athmanathan, A., Stanisavljevic, M., Papandreou, N. et al. (2016). Multilevel-cell phase-change memory: a viable technology. *IEEE Journal of Emerging and Selected Topics in Circuits and Systems* 6 (1): 87–100.
- Burr, G.W., Brightsky, M.J., Sebastian, A. et al. (2016). Recent progress in phase-change memory technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6 (2): 146–162.
- Burr, G.W., Shelby, R.M., Sebastian, A. et al. (2016). Neuromorphic computing using non-volatile memory. *Advances in Physics: X* 2: 89–124.
- Boybat, I., Manuel Le Gallo, S.R., Nandakumar, T.M. et al. (2018). Neuromorphic computing with multi-memristive synapses. *Nature Communications* 9 (1): 2514. <https://doi.org/10.1038/s41467-018-04933-y>.
- Ceze, L., Nivala, J., and Strauss, K. (2019). Molecular digital data storage using DNA. *Nature Reviews Genetics* 20 (8): 456–466. <https://doi.org/10.1038/s41576-019-0125-3>.
- Choi, J., Wang, Z., Venkataramani, S., et al. (2018). PACT: Parameterized clipping activation for quantized neural networks. arXiv: 1805.06085v2 [cs.CV].
- Cross, A.W., Bishop, L.S., Sheldon, S., et al. (2019). Validating quantum computers using randomized model circuits. arXiv: 1811.12926v2 [quant-ph].
- DeBole, M.V., Taba, B., Amir, A. et al. (2019). TrueNorth: accelerating from zero to 64 million neurons in 10 years. *IEEE Computer* 52: 20–28.
- DeBole, M.V., Taba, B., and Amir, A. et al. (2019). TrueNorth: accelerating from zero to 64 million neurons in 10 years. *Computer* 52 (5): 20–29.
- Egger, D.J., Gutiérrez, R.G., Mestre, J.C. et al. (2019). Credit risk analysis using quantum computers. arXiv: 1907.03044 [quant-ph].
- Feynman, R. (1982). Simulating physics with computers. *International Journal of Theoretical Physics* 21 (6/7).
- Gao, Q., Nakamura, H., Gujarati, T.P., et al. (2019). Computational investigations of the lithium superoxide dimer rearrangement on noisy quantum devices. arXiv: 1906.10675 [quant-ph].
- Green, D.G. and William MJ. The future of computing: bits + neurons + qubits. arXiv: 1911.08446 [physics.pop-ph].
- Gupta, S., Agrawal, A., Gopal Krishnan, K. et al. (2015). Deep learning with limited numerical precision. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning – Volume 37* (ICML’15), pp. 1737–1746. JM:R.org.
- Harwood, S.M., Trenev, D., Stober, S.T. et al. (2022). Improving the variational quantum eigensolver using variational adiabatic quantum computing. *ACM Transactions on Quantum Computing* 3, 1, Article 1 (March 2022). <https://doi.org/10.1145/3479197>.
- Havlíček, V., Córcoles, A.D., Temme, K. et al. (2019). Supervised learning with quantum-enhanced feature spaces. *Nature* 567: 209–212.
- Haensch, W., Gokmen, T., and Puri, R. (2018). The next generation of deep learning hardware: analog computing. *Proceedings of the IEEE* 107: 108–122.
- Kandala, A., Mezzacapo, A., Temme, K. et al. (ed.) (2017). Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549: 242–246.
- LeCun, Y., Bottou, L., Bengio, Y. et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86: 2278–2324.
- Mackin, C., Tsai, H., Ambrogio, S. et al. (2019). Weight programming in DNN analog hardware accelerators in the presence of NVM variability. *Advanced Electronic Materials* 5: 1900026.
- Merolla, P.A., Arthur, J.V., Alvarez-Icaza, R. et al. (2014). A million spiking-neuron integrated circuit with scalable communication network and interface. *Science* 345 (6197): 668–673.

- Rice, J.E., Gujarati, T.P., Takeshita, T.T. et al. (2020). Quantum chemistry simulations of dominant products in lithium-sulfur batteries. arXiv: 2001.01120 [physics.chem-ph].
- Shannon, C.E. (1940). *A Symbolic Analysis of Relay and Switching Circuits*. Thesis. MIT, Department of Electrical Engineering.
- Shannon, C.E. (1948). A mathematical theory of communication. *Bell System Technical Journal* 27: 379–423 and 623–656.
- Shannon, C.E. and Weaver, W. (1949). *The Mathematical Theory of Communication*. The University of Illinois Press.
- Stamatopoulos, N., Egger, D.J., Sun, Y. et al. (2020). Option pricing using quantum computers. *Quantum* 4: 291. <https://doi.org/10.22331/q-2020-07-06-291>.
- Suzuki, Y., Uno, S., Raymond, R. et al. (2020). Amplitude estimation without phase estimation. *Quantum Information Processing* 19: 75.
- Suzuki, Y., Yano, H., Gao, Q., et al. (2019). Analysis and synthesis of feature map for kernel-based quantum classifier. arXiv: 1906.10467 [quant-ph].
- Tang, J., Bishop, D., Kim, S. et al. (2018). *ECRAM as Scalable Synaptic Cell for High-Speed, Low-Power Neuromorphic Computing*. IEEE-IEDM.
- Woerner, S. and Egger, D. J. (2019). Quantum risk analysis. *npj Quantum Information* 5: 15. <https://doi.org/10.1038/s41534-019-0130-6>.
- Yuste, R. The discovery of dendritic spines by Cajal. *Frontiers in Neuroanatomy* 9: 18. <https://doi.org/10.3389/fnana.2015.00018>.
- Zoufal, C., Lucchi, A., and Woerner, S. Quantum generative adversarial networks for learning and loading random distributions. *npj Quantum Information* 5: 103 (2019). <https://doi.org/10.1038/s41534-019-0223-2>.
- <https://www.research.ibm.com/frontiers/ibm-q.html>
- <https://news.exxonmobil.com/press-release/exxonmobil-and-ibm-advance-energy-sector-application-quantum-computing>

Index

Note: Page numbers referring to figures should be *italics* and those referring to tables should be **bold**

a

- absolute error loss 8
- accelerators 452
- accuracy score 234
- ACF (autocorrelation function) 80, 81
- activation function 223, 224, 224, 227
- aeronautics 474
- affinity propagation 252, 259–262
- agglomerative algorithms 252
- AI (artificial intelligence) 1, 273, 375, 465
- alternative hypothesis 132
- analysis of variance (ANOVA) F statistic 137–140, 168
- ANN. *see* artificial neural networks
- ANOVA (analysis of variance) F statistic 137–140, 168
- api.py file 428, 443, 445, 458
- apiVersion 423
- artificial intelligence (AI) 1, 375, 465
- artificial neural networks (ANN) 175, 223–249
 - backpropagation 227–230
 - convolutional neural networks 230–232
 - estimation of parameters 225–230
 - multilayer perceptron 224–225
 - multilayer perceptron neural networks 233–242
 - recurrent neural networks 232–233, 233
 - schematic representation of 223
- autocorrelation function (ACF) 80, 81

b

- backpropagation
 - binary classification 226–227
 - multi-class classification 227–230
- backward difference encoding method 72–73
- backward elimination 151–153
- backward propagation of errors 226
- bag-of-words method 274, 278, 280, 296
- bash command 378

- batch gradient descent 181
- batch inference 389, 415, 424–428
- Bayesian approach 76
- Bayesian encoders 58
- Bayesian machine learning models 175
- BCSS (between-cluster sum of squares) 253
- Bernoulli random variables 132
- BERT. *see* Bidirectional Encoder Representations from Transformers
- between-cluster sum of squares (BCSS) 253
- bias 4
- Bidirectional Encoder Representations from Transformers (BERT) 11, 251, 286–287
- bidirectional
 - training 287
- for binary text classification using tensorflow 288–294
- functionality 287–288
- language models 287
- objective 287
- for question answering 296–297
- for text summarization 294–295
- bidirectional training approach 287
- binary classification 2, 225, 333–337
 - with EstimatorQNN 338–343
 - regression 351
 - with SamplerQNN 343–348
 - with variational quantum classifier 348–351
- binary cross-entropy 225
 - as loss functions 8, 352
- binary encoding method, 64, 64–65
- binary logistic regression 202–204
 - cost function 203–204
 - gradient descent 204
 - with Keras on TensorFlow 210–211
- binary systems 466–467
- biologically inspired systems 468–469

bottom-up clustering 252
 Box-Cox transformation 46, 47

c

Caffe2 12
 California Cooperative Oceanic Fisheries Investigations (CalCOFI) dataset 185
 Captum 12
 categorical cross-entropy loss 8
 categorical (discrete) data 17–18, 57–77
 encoding methods 57–58, 58. *see also* encoding methods
 nominal 57
 types of 57
 using hephAistos to encode 77
 workflow with continuous and 58
 CD (continuous delivery) 389, 396, 397
 CentOS 378, 397
 CERN 304
 ChatGPT 286
 Chebyshev norm 43
 chi-squared test 134–137, 168
 for feature selection 135
 percentiles of 136
 CI (continuous integration) 389, 396, 397
 class-dependent LDA 110
 classical computing 465
 classical information theory 300
 classical kernel method 307
 classical neural network 352
 classical physics 299, 470
 classical support vector machines 303
 classification accuracy 6, 327
 classification algorithms 23–31, 274
 on CPUs 14
 on GPUs 14, 24–25, 266
 support vector machine using 220–222
 class-independent LDA 110
 clustering 252–264
 Kubernetes 407–409, 416–418, 426, 435–437
 objective of 175
 techniques 252
 clustering algorithm 252–264
 affinity propagation 259–262
 DBSCAN 252, 262–264
 k-means 252–255
 mean shift clustering 252, 257–259
 mini-batch k-means 252, 255–256
 CNNs (convolutional neural networks) 12, 32, 223, 230–232, 246–249, 269, 274
 CNOT (Controlled-NOT) gate 302
 “cocktail party problem” 102

conditional maximum likelihood estimation 203
 configuration files, for Kubernetes 422–423, 427–428
 confusion matrix 5, 5–7
 containerization 376
 of machine learning application 443–446
 continuous delivery (CD) 389, 396, 397
 continuous integration (CI) 389, 396, 397
 continuous uniform distribution 48
 continuous variables 57, 134
 Controlled-NOT (CNOT) gate 302
 Control Plane 405
 conventional support vector machines 303
 convolutional layers 230
 convolutional neural networks (CNNs) 12, 32, 223, 230–232, 246–249, 269, 274
 CoreferenceHandler class 294
 correlation matrix 143
 cost function 8, 176, 203–204
 gradient descent to 177–181
 covariance 98
 covariance matrix 98
 covariant quantum kernels 309
 CPUs
 classification algorithms on 14
 machine learning on 13–32
 regression algorithms on 14–15, 267
 critical value 132
 cross-entropy loss function 8, 203, 225
 cross-validation score 4, 5, 234
 cryptography 473
 CUDA toolkit 11
 cuDNN libraries 11
 cumulative distribution function 43, 44, 48
 curl command 395, 433, 439
 cybersecurity 473

d

DataFrame 15
 data points 91
 data preprocessing 35
 for machine learning 36
 data rescaling method 18–19
 data scaling 36
 datasets 35
 standardization 37
 data transformation 37–50
 logistic data transformation 43
 lognormal transformation 43–44
 MaxAbsScaler 40
 MinMaxScaler 39
 to normal distribution 44–48
 normalization 42–43

- quantile transformation 48–50
- RobustScaler 40–41
- StandardScaler 37–38
- using unsupervised learning 98
- data visualization 128
- DBSCAN (density-based spatial clustering of applications with noise) 252, 262–264
- decision tree algorithms 4, 154
- deep learning (DL) 1, 3
 - algorithms 88, 168
 - approaches 274
 - machine learning and. *see* machine learning
 - Red Hat OpenShift to 452–454
 - deep residual learning 11
 - default data mapping 308
 - density-based spatial clustering of applications with noise (DBSCAN) 252, 262–264
 - deployment.yaml file 438, 448
 - descriptive analysis 35
 - deviation encoding, effect of 68
 - DevOps tools 396–405, 454
 - diff()* method 85
 - discrete values 57, 202
 - distance-based models 36
 - distribution 69
 - divisive clustering algorithms 252
 - Docker Engine 377
 - Dockerfile 380–381, 383–384, 390, 393, 418, 424, 432, 443, 459
 - Docker, for machine learning 375–389
 - batch inference 424–440
 - build and run 381–389
 - containerization 376
 - Dockerfile 380–381
 - GitHub repository with 454–463
 - Cloud Kubernetes service 440–452
 - install 377–378
 - integrate machine learning models using 396–405
 - and Kubernetes. *see* Kubernetes
 - machine learning prediction in real time using 428–440
 - microservices 375–376
 - for online inference 393–394, 432–434
 - and Python APIs with Flask 389–396
 - training models 415–424
 - use of 378–379
 - Docker Hub 377
 - dummy variable 61
- e**
 - EEGs (electroencephalograms) 102
 - eigenvalues 99, 110
 - eigenvectors 99, 110
 - elastic net method 157
 - electroencephalograms (EEGs) 102
 - Electronic Numerical Integrator and Computer (ENIAC) 300
 - element-wise multiplication 232
 - ELIZA program 273
 - embedded methods 22–23, 154–167, 169
 - elastic net method 157
 - lasso regression 154–156
 - ridge regression 156–157
 - tree-based algorithms 161–165
 - embedded-type feature selection 131
 - encoding functions 308, 310
 - encoding methods 57–58, 58
 - backward difference encoding method 72–73
 - binary encoding method 64, 64–65
 - frequency encoding method 65–66
 - hashing encoding 71–72
 - Helmert encoding method 58, 63–64
 - hephAIstos encoding method 77
 - James-Stein encoding method 74–75
 - label encoding method 62
 - leave-one-out encoding 73–74
 - mean encoding 66–67, 67
 - M-estimator encoding 76
 - one-hot encoding method 61, 61–62
 - ordinal encoding method 59–60
 - probability ratio encoding 70–71
 - sum encoding method 68
 - weight of evidence method 68–70
 - ENIAC (Electronic Numerical Integrator and Computer) 300
 - equivalent docker command 423
 - error function 218
 - error rate 6
 - EstimatorQNN 338–343
 - Euclidean distance 116
 - Euclidian norm 43
 - exhaustive feature selection 153–154
 - expanding window feature 84
 - explained variance ratio 102
- f**
 - feature engineering techniques 35
 - feature extraction 97–130
 - feature rescaling 36–57
 - time-related features engineering 77–88
 - work with categorical data. *see* categorical (discrete) data
 - feature extraction method 19–20, 97–130
 - advantage 97
 - with hephAIstos 130
 - independent component analysis 102–109
 - linear discriminant analysis 110–115
 - locally linear embedding method 115–123
 - manifold learning techniques 125–130

- feature extraction method (*cont'd*)
 principal component analysis 98–102
t-Distributed Stochastic Neighbor Embedding technique 123, 123–125
 feature rescaling 36–57
 data transformation. *see* data transformation and feature selection techniques 369
 SVM model 50–57
 feature selection method 20–23, 97, 131–169
 embedded methods 22–23, 131, 154–167
 filter methods 132–146
 flow of 131
 permutation feature importance 165–167
 tree-based algorithms for 165
 using GPUs 167–168
 using hephAIstos 168–169
 wrapper methods 21–22, 146–154
 feature space 306
 filter methods 132–146
 advantage 146
 ANOVA F-value 137–140
 chi-squared test 134–137
 disadvantage 146
 many more possibilities 146
 Pearson correlation coefficient 140–146
 using statistical tests 132–134
 variance threshold 132
 fine needle aspirate (FNA) 138
 Fisher's formula 110
`fit_transform()` method 63, 64
 Flask
 GitHub repository with 454–463
 Python APIs with 389–396, 428–440
 Flask-RESTful APIs 390–392, 428–431
 fMRI (functional magnetic resonance imaging) analysis 102
 FMs (foundation models) 286
 FNA (fine needle aspirate) 138
 forward stepwise selection 146, 146–150
 foundation models (FMs) 286
 frequency encoding method 65–66
 F-score 6–7
 F-test statistic 137–140
 fully connected layers 230
 functional magnetic resonance imaging (fMRI) analysis 102
- g**
 GABAergic interneurons 307
 Gambetta's law 465
 Gaussian distribution 44–48
 quantile transformation with 49
 Gaussian scaling 37
 Generative Pre-Training (GPT) 286
- get_dummies function 61
 Gibbs sampling 92
 Gini index 161
 GitHub repository 33, 294, 402
 source code in 454–463
 glial cells 220
 GPT (Generative Pre-Training) 286
 GPUs
 classification algorithms on 14, 24–25, 266
 feature selection method using 167–168
 machine learning on 13–32
 regression algorithms on 15, 267–268
 gradient descent 177–181, 204
- h**
 Hadamard gate 302, 308
 ham messages 274–281
 hashing encoding methods 71–72
 head() method 255
 HelmertEncoder function 63
 Helmert encoding method 58, 63–64, 72
 hephAIstos
 categorical data 17–18
 classification algorithms 23–31
 data rescaling method 18–19
 encoding method 77
 feature extraction method 19–20, 130
 feature selection method using 20–23, 168–169
 function 15–32
 installation 13–15
 for machine learning 13–32
 machine learning algorithms with 264–269
 quantum algorithms with 368–372
 time series transformation 16–17
 time series with 88
 training and testing datasets 15–16
 Hessian locally linear embedding (HLLE) 121, 122
 hierarchical clustering 251, 252
 Higgs boson 304
 high-performance computing 467
 Hilbert-Schmidt space 307
 Hilbert space 306
 hinge loss function 8
 HLLE (Hessian locally linear embedding)
 121, 122
 Huber loss 8
 HuggingFace Pytorch transformers 294
 hybrid cloud environment 286
 hybrid quantum-classical algorithm 352
 hyperparameter optimization 303
 hyperplane 306
 hypothesis testing 132

i

IBM Cloud Container Registry 446, 447, 450
 IBM Cloud Kubernetes service 440–452
 IBM Power System 452
 IBM Quantum framework 310, 318
 ICA (independent component analysis) 102–109
 identity function 223
 independent component analysis (ICA) 102–109
 independent variables 70
inference.py script 382–384, 387, 388–389, 424–426
 information value (IV) 69
 integer encoding method 59
 integrate machine learning models 396–405
 intercept-only model 137
 internal network 410–414
 interneuron phenotypes 307
 interneurons 219–220
 interquartile range (IQR) 40, 41
 inverse document frequency 278
 inverse linear correlation 140
 IQR (interquartile range) 40, 41
 IV (information value) 69

j

James-Stein encoding method 74–75
 Jenkins 389, 400
 installation 397–399
 integrate machine learning models using 396–405
 scenario implementation 399–405
 JupyterLab 13
 Jupyter Notebooks 13, 33, 77, 98, 182, 353, 379

k

KDE (kernel density estimation) 257
 Keras 12, 167
 binary logistic regression with 210–211
 logistic regression with 208–210
 kernel density estimation (KDE) 257
 kernel functions 212, 217, 303, 306
 kernel trick. *see* kernel function
 k-fold cross-validation 4
 k-means clustering algorithm 252–255
 k-nearest neighbors (KNN) 274
 imputation method 93–97
 permutation feature importance with 165–166
 KNN. *see* k-nearest neighbors
 Kubeadm installation 434–435
 kubectl 453
 kubelet 405
 kube-proxy 405
 Kubernetes
 application to 448–452

batch inference 424–428
 CLI 442
 cluster 407–409, 416–418, 424, 426, 435–437
 commands to delete 424
 configuration files for 422–423, 427–428
 containerized machine learning model to 437–440
 distribution 376
 initialization and internal network 410–414
 installation 406–407, 415–416
 internal network 417
 Jobs 415
 kubectl 453
 machine learning with 405–414
 OpenShift and 453–454
 Python APIs with 428–440
 service on Public Cloud 440–452
 training models 415–424
 virtual machines 415
 vocabulary 405–406
 Kubernetes Dashboard 452
 Kullback–Leibler divergence 123
kustomization.yaml file 439, 450

l

label encoding method 62, 130
 Lagrange multipliers 213
 lag variables 79–82
 language modeling 287
 Large Models, Large Language Models (LMs/LLMs) 286
 LDA (linear discriminant analysis) 7, 110–115, 381, 430, 431
 learning algorithm 177
 learning styles, for machine learning 2–9
 methods 9
 reinforcement learning 9
 semi-supervised learning 9
 supervised learning. *see* supervised learning
 unsupervised learning 9
 least absolute shrinkage and selection operator (lasso)
 regression 154–156, 169
 leave-one-out encoding method 73–74
 linear algorithms 131
 linear discriminant analysis (LDA) 7, 110–115, 381, 430, 431
 linear interpolation 91–92
 linear regression 68, 137, 176–202
 gradient descent to cost function 177–181
 implementation 182–202
 math 176–177
 multiple 185–202
 univariate 182–185
 linear support vector classification algorithm 157
 LLE (locally linear embedding) method 115–123
 L-Measure 219

LMs/LLMs (Large Models, Large Language Models) 286
l2 norm 43
locally linear embedding (LLE) method 115–123
local tangent space alignment (LTSA) 121
loc method 255
logistic data transformation 43
logistic function 202
logistic regression 202–211
 binary 202–204, 210–211
 with Keras on TensorFlow 208–210
 multinomial. *see* multinomial logistic regression
 with sklearn 205–208
log loss 8
lognormal cumulative distribution functions 43, 44
lognormal transformation 43
log transformation 44
long short-term memory (LSTM) 233, 242–246
loss functions 7–9, 203, 225–226
 binary cross-entropy as 352
L2 regularization (ridge regression) 156–157
LSTM (long short-term memory) 233, 242–246
LTSA (local tangent space alignment) 121

m

machine learning (ML) 1. *see also* machine learning
 algorithms
 application 443–446
 data preprocessing for 36
 Docker. *see* Docker, for machine learning
 feature engineering. *see* feature engineering techniques
 goal 4
 handling missing values in 88–97
 hephAIstos for running. *see* hephAIstos, for machine
 learning
 learning styles for 2–9
 models 392–393, 431–432, 454–463
 production. *see* production, machine learning in
 Python tools for 9–13
 quantum advantage for 303
 quantum computing and. *see* quantum computing
 Red Hat OpenShift to 452–454
 workflow 2
machine learning algorithms 94
 artificial neural networks 223–249
 with hephAIstos 264–269
 linear regression 176–202
 logistic regression 202–211
 many more algorithms to explore 249–251
 in quantum computing. *see* quantum machine learning
 rule-based 176
 support vector machine 211–222
 unsupervised 251–264

machine learning operations (MLOps) 396–405
MAE (mean of the absolute errors) 8
magnetoencephalograms (MEGs) 102
Mahalanobis distance transformation 318, 327
make_blobs() function 253
manifold learners 116
MAR (missing at random) 92
Masked Language Modeling (MLM) 287, 288
math 176–177
matplotlib 10, 101
MaxAbsScaler 40, 57
maximum-margin hyperplane 212, 306
max-norm 43
MCAR (missing completely at random) 92
mean 37, 37
 imputation 90
mean encoding methods 66–67, 67
mean of the absolute errors (MAE) 8
mean shift clustering 252, 257–259
mean squared error (MSE) 8
median imputation 90
MEGs (magnetoencephalograms) 102
M-estimator encoding methods 76
method(s)
 backward difference encoding 72–73
 bag-of-words 274, 278, 280, 296
 binary encoding 64, 64–65
 data rescaling 18–19
 diff() 85
 elastic net 157
 embedded. *see* embedded methods
 encoding. *see* encoding methods
 feature extraction. *see* feature extraction method
 feature selection. *see* feature selection method
 filter 132–146
 head() 255
 KNN imputation 93–97
 loc 255
 quantum kernel 307
 regularization 154
 shift() 79
 weight of evidence 68–70
wrapper. *see* wrapper methods
MICE (multivariate imputation by chained equation)
 imputations 92–93, 93
microservice approach 375–376
mini-batch k-means clustering algorithm 252, 255–256
Minkowski norm 43
MinMaxScaler 39, 57, 120
misclassification rate 6
missing at random (MAR) 92
missing completely at random (MCAR) 92

- missing_method 97
 missing not at random (MNAR) 92
 missing values handling, in machine learning 88–97
 KNN imputation method 93–97
 linear interpolation 91–92
 multivariate imputation by chained equation 92–93, 93
 row or column removal 89–90
 statistical imputation 90–91
 ML. *see* machine learning
 MLLE (modified locally linear embedding) 121, 122
 MLM (Masked Language Modeling) 287, 288
 MLOps (machine learning operations) 396–405
 MLP 224–225
 MLP neural networks 233–242, 395, 430, 433–434
 MNAR (missing not at random) 92
 MNIST dataset 12
 mode imputation 90
 model-building process 131
 modified locally linear embedding (MLLE) 121, 122
 monolithic approach 375
 Monte Carlo simulations 473
 Moore’s law 465, 467
 MSE (mean squared error) 8
 multi-class classification 2, 227–230
 multinomial logistic regression 154, 204
 applied to fashion MNIST 204–210
 multiple linear regression 176, 181, 185–202
 with Keras on TensorFlow 196–202
 with scikit-learn 188–191
 with *statsmodels*, 192–193
 with TensorFlow 193–196
 multivariate imputation by chained equation (MICE)
 imputations 92–93, 93
- n**
- namespace.yaml file 437
 NaN (Not a Number) 80
 natural language generation (NLG) 274
 natural language processing (NLP) 97, 251, 273
 BERT. *see* Bidirectional Encoder Representations from
 Transformers
 deep learning approaches 274
 domain of 296
 messages as spam or ham 274–281
 neural approaches 274
 sentiment analysis 281–286
 statistical approaches 274
 symbolic approach 274
 tools and libraries for 273
 Natural Language Toolkit (NLTK) 273, 274
 natural language understanding (NLU) 274
 negative log-likelihood loss 203
 NeuralNetworkClassifier 337, 338, 343, 348
 neural network multilayer perceptron 233–242, 381, 395,
 430, 431
 NeuralNetworkRegressor 337
 neural networks 27, 175, 224
 layers 224
 neural NLP approaches 274
 NeuroM 219
 neuronal classification 307
 neuron morphology 219, 307, 327
 new-app command 462
 Next Sentence Prediction (NSP) technique 288
 NLG (natural language generation) 274
 NLP. *see* natural language processing
 NLTK (Natural Language Toolkit) 273, 274
 NLU (natural language understanding) 274
 nodes 405
 nominal categorical data 57
 non-Gaussian processes 102, 103, 105
 nonlinear support vector machines 216, 216–217, 217
 nonlinear SVM 306
 nonparametric quantile transformation 48
 normal distribution (Gaussian distribution) 44–48
 quantile transformation with 49
 normalization 36, 42–43
 normal probability distribution
 right-tailed test with 133
 two-sided test with 133
 norms, concept of 42
 Not a Number (NaN) 80
 not fully linearly separable data 214–216
 NSP (Next Sentence Prediction) technique 288
 null hypothesis 132
 numerical variable 57
 Numpy 10, 242
- o**
- objective function 36
 one-hot encoding method 61, 61–62
 one-hot vector 204
 one-to-one process 220
 one-to-rest process 220
 online inference 393–394, 428–440
 Open Java Development Kit (OpenJDK) 397
 OpenShift 376, 453
 cluster instance 455–463
 and Kubernetes 453–454
 Red Hat 452–454
 web console 460
 OpenShift Enterprise 457

- open-source world 12
optimal fitting 3
optimal hyperplane 212
ordinal categorical data 57
OrdinalEncoder class 60
ordinal encoding method 59–60
ordinal logistic regression 204
ordinary least squares linear regression 160
overfitting 3, 3–4, 131
- p**
- PACF (partial autocorrelation function) 80, 81
pandas 10, 58, 59, 242
 - one-hot encoding method 61
 - ordinal encoding method 59
partial autocorrelation function (PACF) 80, 81
partitional clustering 251
PauliFeatureMap 308
Pauli-X gate 302
Pauli-Y gate 302
Pauli-Z gate 302
PCA (principal component analysis) 9, 62, 98–102, 104, 106
Pearson correlation coefficient 20, 140–146, 168
pegasos algorithms 28, 333, 369
pegasos QSVC 333–337
penalization methods 154
p-norm 43
Pods 405, 411, 414, 438
Poisson model 93
polylogarithmic function 306
pooling layers 230
port 8888, 379
precision score 6, 234
pre-trained models 286
primal optimization problem 213
principal cells 219
principal component analysis (PCA) 9, 62, 98–102, 104, 106
probability distribution 225
probability ratio encoding 70–71
probability value (*p*-value) 132, 137
production, machine learning in
 - DevOps to MLOPS 396–405
 - Docker containers for 375–389
 - Kubernetes. *see* Kubernetes
 - in real time using Docker and Python 389–396
punctuation 274
p-value (probability value) 132, 137
Python APIs
 - with Flask 389–396, 428–440
 - with Kubernetes 428–440
Python ecosystem 380
- Python language 472
Python tools 9–13
 - data manipulation with 10
 - Cloud Kubernetes service 440–452
 - JupyterLab 13
 - Jupyter Notebook 13
 - Keras 12
 - libraries 10–13
 - PyTorch 12
 - scikit-learn 10
 - TensorFlow 10–12, 288
PyTorch 12, 168, 352
- q**
- qGAN (quantum generative adversarial network) 352–368
Qiskit 304, 305
Qiskit Runtime 305–306
QKA (quantum kernel alignment) 27, 309, 328
QKE (quantum kernel estimation) 307
QNNs (quantum neural networks) 303, 337–351
QPUs. *see* quantum processing units
QSVC 333–337
QSVM 304, 306
qualitative variables 35
quantile function 48
quantile transformation 48–50
quantitative variables 35
quantum computing 474
 - algorithms with hephAistos 368–372
 - characteristics of 305
 - mechanics 299
 - 127-qubit superconducting 305
 - Pegasos QSVC 333–337
 - potential use cases for 304
 - QNN. *see* quantum neural networks
 - quantum Generative Adversarial Network 352–368
 - quantum kernel machine learning 306–327
 - quantum kernel training 328–333
 - quantum machine learning 303–306
quantum feature mapping 311
quantum gates 302
quantum generative adversarial network (qGAN) 352–368
quantum information theory 300
quantum kernel alignment (QKA) 27, 309, 328
quantum kernel estimation (QKE) 307
quantum kernel machine learning 306–327
quantum kernel methods 307
QuantumKernelTrained.fit method 309, 328
QuantumKernelTrainer 309
quantum kernel training 328–333
quantum machine learning 303–306

quantum mapping function 307
 quantum mechanics 470
 development of 300
 fundamental equation of 300
 notion of 299
 quantum neural networks (QNNs) 337–351
 quantum processing units (QPUs) 13–32
 classification algorithms 27
 quantum state space 303
 quantum systems 469–474
 qubits 300, 302, 469–474, 470
 connectivity, 305

r

radial basis function (RBF) kernel 14
 random forest algorithm 161–162
 RBAC (role-based access control) 453
 RBF (radial basis function) kernel 14
 recall score 6, 234
 receiver operating characteristic curve (ROC AUC) 147
 reciprocal transformation 46
 rectified linear unit (ReLU) 223
 recurrent neural networks (RNNs) 29, 30–31, 87, 232–233, 233, 268–269, 274
 long short-term memory (LSTM) 242–246
 Red Hat 378, 397
 Red Hat OpenShift 452–454
 container platform 454–463
 regression algorithms
 binary classification 351
 on CPUs 14–15, 267
 on GPUs 15, 267–268
 regression models 2, 93
 with an ϵ -insensitive tube, 218
 support vector machine using 222
 regularization methods 154
 reinforcement learning 9
 ReLU (rectified linear unit) 223
 residual error 93
 residual variance 93
 responsibility matrix 260
 RGB model 230, 230
 ridge regression 156–157
 right-tailed test 133
 RMSE (root-mean-squared error) 177, 183
 RNNs. *see* recurrent neural networks
 RobustScaler 40–41, 57
 role-based access control (RBAC) 453
 rolling window features 82–83
 root-mean-squared error (RMSE) 177, 183
 R-squared score 183
 rule-based machine learning algorithms 176

s

SamplerQNN 338, 343–348
 scaling 36
 Schrödinger equation 300, 471
 scikit-learn 10, 21, 36, 43, 58, 112
 digits dataset using 123
 Gaussian distribution using 49
 GitHub repository with 454–463
 locally linear embedding using 118
 logistic regression with 205–208
 multiple linear regression with 188–191
 naive Bayes with 280
 support vector machines using 220–222
 SciPy 10, 134
 Seaborn 10
 self-supervised learning 287
 semi-supervised learning 9
 sentiment analysis 281–286
 sequential forward selection (SFS) 148
 service_port.yaml file 449–450
 service.yaml file 438, 449
 SFS (sequential forward selection) 148
 SGD optimizer 26, 267
 shift() method 79
 Shor algorithm 473
 sigmoid function 202, 202, 223, 227
 simple linear regression model 137, 177
 simple word count 274
 sklearn. *see* scikit-learn
 skorch 12
 slack variable 215
 softmax function 204
 softmax loss 8
 SOTA (State-of-the-Art) models 286
 spam messages 274–281
 Spearman’s rank correlation 134
 .spec.template 423
 squared deviation 102
 squared error loss 8
 square root transformation 45
 standard deviation 37, 37, 98
 standardization 36, 37, 110
 StandardScaler 37–38, 57, 100, 130
 State-of-the-Art (SOTA) models 286
 statistical imputation 90–91
 statistical learning 1, 223
 statistical NLP approaches 274
 statistical tests 132–134
 stochastic gradient descent 181, 226, 227, 232
 storage systems 466
 sudo command 400
 sum encoding method 68

- summation 223
superposition 300, 301
supervised learning 2–9, 110
algorithms 2
confusion matrix 5, 5–7
k-fold cross-validation 4
loss functions 7–9
objectives 2
overfitting and underfitting 3, 3–4
train/test split 4–5
support vector machines (SVM) 36, 50–57, 211–222, 212
application 219–222
classical 303
conventional 303
linearly separable data 212, 212–214
nonlinear 216, 216–217, 217
not fully linearly separable data 214–216
for regression 217–219, 218
using linear kernel 280
using sklearn for regression 222
support vectors 306
SVCLoss 309
SVM. *see* support vector machines
Swiss Roll manifold 116
symbolic NLP approaches 274
target encoding 66
- t**
t-Distributed Stochastic Neighbor Embedding (t-SNE)
technique 123, 123–125
TensorFlow 10–12, 167
BERT for binary text classification using 288–294
Keras on. *see* Keras
multiple linear regression with 193–196
Python 288
term frequency-inverse document frequency (TF-IDF) 278
text summarization, BERT for 294–295
TF-IDF (term frequency-inverse document frequency) 278
three-dimensional space 115
time-related features engineering 77–88
date-related features 79
expanding window feature 84
lag variables 79–82
rolling window features 82–83
time series data 85
time series 77
with hephAistos 88
transformation 16–17
trends and seasonal components 85
time transformation 88
tokens 274
- top-down clustering 252
TorchConnector 352
“TotalGrayVol” data 36
TrainableFidelityQuantumKernel 309
training Job 415
training model 376
Kubernetes Cluster 424
Python application 418–422
train.py file 381–382, 387, 390, 392, 418, 420, 428, 431, 443, 445, 457
train/test split 4–5
tree-based algorithms 36, 57, 161–165
t-SNE (*t*-Distributed Stochastic Neighbor Embedding)
technique 123, 123–125
two-dimensional complex vector space 301
two-sided test 133
- u**
Ubuntu 377
underfitting 3, 3–4
uniform distribution 49
unit vector normalization 42–43
univariate linear regression 176, 182–185, 196
univariate tests 131
universal approximation theorem 225
unrestricted model 137
unsupervised machine learning 9
algorithms 110, 251–264
clustering algorithms 252–264
data transformation using 98
methods 1
- v**
Vagrantfile 407, 408, 409
variance 4, 98
variance threshold 132, 168
variational quantum classifier (VQC) 337, 338, 348–351
variational quantum regressor (VQR) 337
vector norms 42
video games, types of 57
virtual machines (VMs) 376, 415, 434
VMs (virtual machines) 376, 415, 434
VQC (variational quantum classifier) 337, 338, 348–351
VQR (variational quantum regressor) 337
- w**
Ward 252
wave mechanics 299, 300, 470
wave-particle duality 299
“wave–particle duality” 299
WCSS (within-cluster sum of squares) 253

weight of evidence (WoE) method 68–70
within-cluster sum of squares (WCSS) 253
WoE (weight of evidence) method 68–70
wrapper methods 21–22, 131, 146–154,
 168–169
backward elimination 151–153
exhaustive feature selection 153–154
forward stepwise selection 146, 146–150

y

YAML files 437–439
Yeo-Johnson transformation 47

z

ZFeatureMap 308
Z-score normalization 37
ZZFeatureMap 308, 318

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.