2ND EDITION

MATLAB for Machine Learning

Unlock the power of deep learning for swift and enhanced results

GIUSEPPE CIABURRO

本书版权归Packt Publishing所有

MATLAB for Machine Learning

Unlock the power of deep learning for swift
and enhanced results

Giuseppe Ciaburro

‹packt›

MATLAB for Machine Learning

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Niranjan Naikwadi
Publishing Product Manager: Tejashwini R
Book Project Manager: Kirti Pisat
Content Development Editor: Priyanka Soam
Technical Editor: Rahul Limbachiya
Copy Editor: Safis Editing
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Prafulla Nikalje
DevRel Marketing Coordinator: Vinishka Kalra

First published: August 2017
Second edition: January 2024

Production reference: 1190124

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN: 978-1-83508-769-5
www.packtpub.com

To the memory of my father, Luigi, and my mother, Caterina, for their sacrifices and for exemplifying the power of determination. To my sons, Luigi and Simone, the bright stars in my universe, my pillars of strength, and my source of inspiration – may this book be an example of passion and dedication to one's work.

– Giuseppe Ciaburro

Contributors

About the author

Giuseppe Ciaburro holds a PhD in environmental engineering physics and two master's degrees in chemical engineering and in acoustics and noise control. He works at the Built Environment Control Laboratory at "Università degli studi della Campania Luigi Vanvitelli " and has over 20 years of experience in programming, first in the field of combustion and then in acoustics and noise control. His core programming skills are in MATLAB, Python, and R. As an expert in acoustics and noise control, Giuseppe has extensive experience in teaching and researching. He is currently researching machine learning applications in acoustics and noise control. He has written for several publications, and for the last two years, he has been ranked by Stanford University as one of the top 2% of scientists in the world.

I want to thank the people who have been close to me and supported me in the writing of the book, especially the Packt team.

About the reviewers

Dr. Sankar P is a seasoned academician and a dynamic researcher with a diverse background in electronics and communication engineering and information technology, working at Hindustan Institute of Technology and Science, Chennai, India. His academic journey spans over two decades, during which he has made significant contributions to the field through teaching, research, and administrative roles. His expertise as a reviewer for prestigious books and journals such as Pearson, Oxford, Elsevier, and IEEE was recognized with the Best Reviewer Award in 2017.

Dr. S. Mahesh Anand is the founder of Learn AI with Anand. Dr. Anand's journey unfolded through education, innovation, and a passion for data science and AI. Starting his career at VIT University, Dr. Anand's early years shaped a teaching philosophy beyond conventional boundaries. Transitioning to corporate training, he meticulously designed and delivered comprehensive learning programs tailored for aspiring data scientists. He was recognized as the Best Data Science and AI Educator by AI Global Media (UK) and by *Corporate Vision Magazine* in 2022 for his outstanding contributions to education and training. Early in 2000, he received the AT&T Labs Award from the IEEE Headquarters (USA) and the MV Chauhan Award from the IEEE India Council, adding honors to his professional journey.

Table of Contents

Preface xiii

Part 1: Getting Started with Matlab

1

Exploring MATLAB for Machine Learning 3

Technical requirements 3
Introducing ML 4
How to define ML 4
Analysis of logical reasoning 4
Learning strategy typologies 6
Discovering the different types of learning processes 8
Supervised learning 9
Unsupervised learning 10
Reinforcement learning 11
Semi-supervised learning 12
Transfer learning 13
Using ML techniques 13
Selecting the ML paradigm 14
Step-by-step guide on how to build ML models 16
Exploring MATLAB toolboxes for ML 19
Statistics and Machine Learning Toolbox 20
Deep Learning Toolbox 21
Reinforcement Learning Toolbox 22
Computer Vision Toolbox 23
Text Analytics Toolbox 24
ML applications in real life 25
Summary 26

2

Working with Data in MATLAB 29

Technical requirements 30
Importing data into MATLAB 30
Exploring the Import Tool 32
Using the load() function to import files 34
Reading ASCII-delimited files 36
Exporting data from MATLAB 39
Working with different types of data 42
Working with images 42

Audio data handling 43
Exploring data wrangling 44
Introducing data cleaning 45
Discovering exploratory statistics 47
EDA 48
EDA in practice 48
Introducing exploratory visualization 51
Understanding advanced data preprocessing techniques in MATLAB 52
Data normalization for feature scaling 53
Introducing correlation analysis in MATLAB 54
Summary 58

Part 2: Understanding Machine Learning Algorithms in MATLAB

3

Prediction Using Classification and Regression 63

Technical requirements 63
Introducing classification methods using MATLAB 64
Decision trees for decision-making 64
Exploring decision trees in MATLAB 66
Building an effective and accurate classifier 72
SVMs explained 72
Supervised classification using SVM 75
Exploring different types of regression 78
Introducing linear regression 78
Linear regression model in MATLAB 79
Making predictions with regression analysis in MATLAB 82
Multiple linear regression with categorical predictor 82
Evaluating model performance 86
Reducing outlier effects 87
Using advanced techniques for model evaluation and selection in MATLAB 90
Understanding k-fold cross-validation 91
Exploring leave-one-out cross-validation 94
Introducing the bootstrap method 95
Summary 95

4

Clustering Analysis and Dimensionality Reduction 97

Technical requirements 97
Understanding clustering – basic concepts and methods 98
How to measure similarity 98
How to find centroids and centers 100
How to define a grouping 101
Understanding hierarchical clustering 102

Partitioning-based clustering algorithms with MATLAB 108
Introducing the k-means algorithm 108
Using k-means in MATLAB 109

Grouping data using the similarity measures 115
Applying k-medoids in MATLAB 116

Discovering dimensionality reduction techniques 119
Introducing feature selection methods 120
Exploring feature extraction algorithms 122

Feature selection and feature extraction using MATLAB 124
Stepwise regression for feature selection 124
Carrying out PCA 129

Summary 135

5

Introducing Artificial Neural Network Modeling 137

Technical requirements 137
Getting started with ANNs 138
Basic concepts relating to ANNs 138
Understanding how perceptrons work 140
Activation function to introduce non-linearity 141
ANN's architecture explained 143

Training and testing an ANN model in MATLAB 144
How to train an ANN 144
Introducing the MATLAB Neural Network Toolbox 145

Understanding data fitting with ANNs 148

Discovering pattern recognition using ANNs 157

Building a clustering application with an ANN 166

Exploring advanced optimization techniques 171
Understanding SGD 172
Exploring Adam optimization 174
Introducing second-order methods 175

Summary 177

6

Deep Learning and Convolutional Neural Networks 179

Technical requirements 180
Understanding DL basic concepts 180
Automated feature extraction 180
Training a DNN 181

Exploring DL models 182

Approaching CNNs 184
Convolutional layer 185
Pooling layer 187
ReLUs 188
FC layer 188

Building a CNN in MATLAB 189
Exploring the model's results 197
Discovering DL architectures 201
Understanding RNNs 201
Analyzing LSTM networks 202
Introducing transformer models 204
Summary 204

Part 3: Machine Learning in Practice

7

Natural Language Processing Using MATLAB 209

Technical requirements 209
Explaining NLP 210
NLA 212
NLG 213
Analyzing NLP tasks 214
Introducing automatic processing 216
Exploring corpora and word and sentence tokenizers 216
Corpora 217
Words 218
Sentence tokenize 219
Implementing a MATLAB model to label sentences 219
Introducing sentiment analysis 220
Movie review sentiment analysis 221
Using an LSTM model for label sentences 222
Understanding gradient boosting techniques 228
Approaching ensemble learning 228
Bagging definition and meaning 229
Discovering random forest 230
Boosting algorithms explained 231
Summary 233

8

MATLAB for Image Processing and Computer Vision 235

Technical requirements 235
Introducing image processing and computer vision 236
Understanding image processing 236
Explaining computer vision 240
Exploring MATLAB tools for computer vision 241
Building a MATLAB model for object recognition 243
Introducing handwriting recognition (HWR) 244
Training and fine-tuning pretrained deep learning models in MATLAB 250
Introducing the ResNet pretrained network 251
The MATLAB Deep Network Designer app 252

Interpreting and explaining machine learning models 257
Understanding saliency maps 257
Understanding feature importance scores 258
Discovering gradient-based attribution methods 259
Summary 260

9

Time Series Analysis and Forecasting with MATLAB 261

Technical requirements 261
Exploring the basic concepts of time series data 262
Understanding predictive forecasting 262
Introducing forecasting methodologies 263
Time series analysis 265
Extracting statistics from sequential data 269
Converting a dataset into a time series format in MATLAB 270
Understanding time series slicing 272
Resampling time series data in MATLAB 273
Moving average 274
Exponential smoothing 276
Implementing a model to predict the stock market 278
Dealing with imbalanced datasets in MATLAB 288
Understanding oversampling 289
Exploring undersampling 290
Summary 291

10

MATLAB Tools for Recommender Systems 293

Technical requirements 293
Introducing the basic concepts of recommender systems 294
Understanding CF 294
Content-based filtering explained 295
Hybrid recommender systems 296
Finding similar users in data 297
Creating recommender systems for network intrusion detection using MATLAB 303
Recommender system for NIDS 304
NIDS using a recommender system in MATLAB 305
Deploying machine learning models 309
Understanding model compression 310
Discovering model pruning techniques 310
Introducing quantization for efficient inference on edge devices 312
Getting started with knowledge distillation 312
Learning low-rank approximation 313
Summary 314

11

Anomaly Detection in MATLAB 315

Technical requirements 315

Introducing anomaly detection and fault diagnosis systems 316

Anomaly detection overview 316

Fault diagnosis systems explained 317

Approaching fault diagnosis using ML 319

Using ML to identify anomalous functioning 321

Anomaly detection using logistic regression 322

Improving accuracy using the Random Forest algorithm 326

Building a fault diagnosis system using MATLAB 328

Understanding advanced regularization techniques 332

Understanding dropout 332

Exploring L1 and L2 regularization 333

Introducing early stopping 334

Summary 335

Index 337

Other Books You May Enjoy 350

Preface

MATLAB is a comprehensive programming environment used by many researchers and math experts for machine learning. This book will help you learn the basic concepts in machine learning and deep learning using MATLAB, and then refine your basic skills with advanced applications.

You'll start by exploring the tools that the MATLAB environment offers for machine learning and see how to easily interact with the MATLAB workspace. We'll then move on to data cleansing, mining, and analyzing various types of data in machine learning, and you'll see how to visualize data values on a graph. Then, you'll learn about the different types of classification and regression techniques and how to apply them to your data, using MATLAB functions. Further, you will understand the basic concepts of neural networks and perform data fitting, pattern recognition, and clustering analysis. You will also explore feature selection and extraction techniques for dimensionality reduction for performance improvement. Finally, you'll learn how to leverage MATLAB tools for deep learning and managing convolutional neural networks.

By the end of the book, you'll learn how to put it all together in real-world cases, covering major machine learning algorithms, and you'll feel confident as you delve into machine learning with MATLAB.

Who this book is for

This book is suitable for machine learning engineers, data scientists, deep learning engineers, and CV/NLP engineers who want to use MATLAB for machine learning and deep learning. You should have a fundamental understanding of programming concepts.

What this book covers

Chapter 1, *Exploring MATLAB for Machine Learning*, covers machine learning, which is a branch of artificial intelligence that is based on the development of algorithms and mathematical models, capable of "learning" from data and autonomously adapting to improve their performance according to the objectives set. Thanks to this learning ability, machine learning is used in a wide range of applications, such as data analysis, CV, text translation, speech recognition, medical diagnosis, and financial risk prediction. Machine learning is an ever-evolving area of research and is revolutionizing many fields of science and industry. The aim of this chapter is to provide some introduction, background information, and a basic knowledge of MATLAB tools. In addition, the basic concepts of machine learning will be introduced.

Chapter 2, *Working with Data in MATLAB*, looks at how to import and organize our data in MATLAB. Today, the amount of data generated is enormous; smartphones, credit cards, televisions, computers, home appliances, sensors, domestic systems, public and private transport, and so on are just a few examples of devices that generate data seamlessly. Such data is stored and then used for various purposes. One of these is data analysis using machine learning algorithms. To import and organize our data in MATLAB, you should familiarize yourself with the MATLAB workspace to make the operations as simple as possible. Then, we will analyze the different formats available for the data collected and how to move data in and out of MATLAB. We will also explore datatypes to work with grouping variables and categorical data and how to export data from the workspace, including cell array, structure array, and tabular data, and save it in a MATLAB-supported file format. Finally, we will understand how to organize data in the correct format for the next phase of data analysis.

Chapter 3, *Prediction Using Classification and Regression*, shows us how to classify an object using nearest neighbors and how to perform an accurate regression analysis in a MATLAB environment. Classification algorithms return accurate predictions based on our observations. Starting from a set of predefined class labels, the classifier assigns each input data a class label, according to the training model. Regression relates a set of independent variables to a dependent variable. Through this technique, it is possible to understand how the value of the dependent variable changes as the independent variable varies.

Chapter 4, *Clustering Analysis and Dimensionality Reduction*, explores clustering methods, which are designed to find hidden patterns or groupings in a dataset. These algorithms identify a grouping without any label to learn from through the selection of clusters, based on the similarity between the elements. Dimensionality reduction is the process of converting a set of data with many variables into data with lesser dimensions but ensuring similar information. Feature selection approaches try to find a subset of the original variables. Feature extraction reduces the dimensionality of the data by transforming it into new features. This chapter shows us how to divide the data into clusters, or groupings of similar items. We'll also learn how to select a feature that best represents the dataset.

Chapter 5, *Introducing Artificial Neural Networks Modeling*, delves into **artificial neural networks** (**ANNs**), which include data structures and algorithms for the learning and classification of data. Through the neural network techniques, a program can learn by example and create an internal structure of rules to classify different inputs. MATLAB provides algorithms, pretrained models, and apps to create, train, visualize, and simulate ANNs. In this chapter, we will see how to use MATLAB to build an ANN-based model to predict values and classify data.

Chapter 6, *Deep Learning and Convolutional Neural Networks*, examines deep learning, which is a machine learning technology based on multilayer ANNs and has allowed many applications to reach a high degree of accuracy. Deep neural networks are capable of modeling complex relationships between input and output data. Among the most successful applications is computer vision, with tasks that include classification, image regression, and object detection. For example, a deep neural network is able to generate a layered representation of objects in which each object is identified by a set of characteristics that has the form of visual primitives, such as particular edges, oriented lines, textures, and recurring patterns. Convolutional networks are characterized by convolutional layers, which use

filters to analyze data in a local region and produce an activation map. These activation maps are then processed by pooling layers, which aggregate the low-resolution data to reduce the dimensionality of the representation and make processing more computationally efficient. The convolutional and pooling layers are then alternated several times until an image is represented by a low-resolution activation map. In this chapter, we will learn the basic concepts of deep learning and discover how to implement an algorithm based on convolutional networks in the MATLAB environment.

Chapter 7, *Natural Language Processing Using MATLAB*, explores **natural language processing** (**NLP**), which automatically processes information conveyed through spoken or written language. This task is fraught with difficulty and complexity, largely due to the innate ambiguity of human language. To enable machine learning and interaction with the world in ways typical of humans, it is essential not only to store data but also to teach machines how to translate it simultaneously into meaningful concepts. As natural language interacts with the environment, it generates predictive knowledge. In this chapter, we will learn the basic concepts of NLP and how to build a model to label sentences.

Chapter 8, *MATLAB for Image Processing and Computer Vision*, covers computer vision, which is a field that studies how to process, analyze, and understand the contents of visual data. In image content analysis, we use a lot of computer vision algorithms to build our understanding of the objects in an image. Computer vision covers various aspects of image analysis, such as object recognition, shape analysis, pose estimation, 3D modeling, and visual search. Humans are good at identifying and recognizing things around them! The goal of computer vision is to accurately model the human vision system using computers. In this chapter, we will understand the basic concepts of computer vision and how to implement a model for object recognition, using MATLAB.

Chapter 9, *Time Series Analysis and Forecasting with MATLAB*, delves into time series data, which is basically a sequence of measurements that are collected over time. These measurements are taken with respect to a predetermined variable and at regular time intervals. One of the main characteristics of time series data is that the ordering matters. The list of observations that we collect is ordered on a timeline, and the order in which they appear says a lot about underlying patterns. If you change the order, this will totally change the meaning of the data. Sequential data is a generalized notion that encompasses any data that comes in a sequential form, including time series data. In this chapter, we will learn the basic concepts of sequential data and how to build a model that describes the pattern of the time series or any sequence in general.

Chapter 10, *MATLAB Tools for Recommender Systems*, examines the recommendation engine, which is a model that can predict what a user may be interested in. When we apply this to the context of movies, for example, this becomes a movie recommendation engine. We filter items in our database by predicting how the current user might rate them. This helps us in connecting the user to the right content in our dataset. Why is this relevant? If you have a massive catalog, then the user may or may not find all the content that is relevant to them. By recommending the right content, you increase consumption. Companies such as Netflix heavily rely on recommendations to keep the user engaged. In this chapter, we will learn the basic concepts of recommender systems and how to build a movie recommendations system, using MATLAB.

Chapter 11, *Anomaly Detection in MATLAB*, teaches you the basic concepts of an anomaly detection system and how to implement one in MATLAB. A physical system, in its life cycle, can be subject to failures or malfunctions that can compromise its normal operation. It is, therefore, necessary to introduce an anomaly detection system within the capability of preventing critical interruptions. This is called a fault diagnosis system and can identify the possible presence of a malfunction within the monitored system. The search for the fault is one of the most important and qualifying maintenance intervention phases, and it is necessary to act in a systematic and deterministic way. To carry out a complete search for the fault, it is necessary to analyze all the possible causes that may have determined it.

To get the most out of this book

In this book, machine learning algorithms are implemented in the MATLAB environment. So, to reproduce the many examples in this book, you need a new version of MATLAB (R2023b is recommended) and the following toolboxes – a statistics and machine learning toolbox, a neural network toolbox, a deep learning toolbox, and a fuzzy logic toolbox.

Software/hardware covered in the book	**Operating system requirements**
MATLAB	Windows, macOS, or Linux

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

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`. If there's an update to the code, it will be updated in the GitHub repository.

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

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "To create a classification tree, we can utilize the `fitctree()` function."

A block of code is set as follows:

```
gscatter(meas(:,3), meas(:,4), species,'rgb','osd');
xlabel('Petal length');
ylabel('Petal width');
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Now, we can train the network just by clicking on the **train** button of the app. After a few seconds, the ANN will be trained and ready for use."

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

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share Your Thoughts

Once you've read *MATLAB for Machine Learning*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835087695`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Getting Started with Matlab

This part provides background information and essential knowledge about MATLAB tools, along with an introduction to basic machine learning concepts. We also will focus on importing and organizing data in MATLAB, emphasizing familiarity with the MATLAB workspace for simplicity in operations. The discussion covers analyzing various data formats, moving data in and out of MATLAB, exploring datatypes for grouping variables and categorical data, exporting data in different formats such as cell arrays, structure arrays, and tabular data, and saving it in MATLAB-supported file formats. The ultimate goal is to prepare data in the right format for the subsequent phase of data analysis.

This part has the following chapters:

- *Chapter 1, Exploring MATLAB for Machine Learning*
- *Chapter 2, Working with Data in MATLAB*

1
Exploring MATLAB for Machine Learning

Machine learning (**ML**) is a branch of artificial intelligence that is based on the development of algorithms and mathematical models capable of learning from data and autonomously adapting to improve their performance according to a set of objectives. Thanks to this learning ability, ML is used in a wide range of applications, such as data analysis, computer vision, language modeling, speech recognition, medical diagnosis, and financial risk prediction. ML is an ever-evolving area of research and is revolutionizing many fields of science and industry. The aim of this chapter is to provide you with an introduction, background information, and a basic knowledge of ML, as well as an understanding of how to apply these concepts using MATLAB tools.

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

- Introducing ML
- Discovering the different types of learning processes
- Using ML techniques
- Exploring MATLAB toolboxes for ML
- ML applications in real life

Technical requirements

In this chapter, we will introduce basic concepts relating to ML. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. A working knowledge of the MATLAB environment is also required.

Introducing ML

ML is based on the idea of providing computers with a large amount of input data, together with the corresponding correct answers or labels, and allowing them to learn from this data, identifying patterns, relationships, and regularities within them. Unlike traditional programming approaches, in which computers follow precise instructions to perform specific tasks, ML allows machines to independently learn from data and make decisions based on statistical models and predictions.

One of the key concepts of ML is the ability to generalize. This means that a model trained on information in the training dataset should be able to make accurate predictions about new data that it has never seen before. This allows ML to be applied across a wide range of domains.

How to define ML

To better understand the basic concepts of ML, we can start from the definitions formulated by the pioneers in this field. According to Arthur L. Samuel (1959) – "ML is a field of study that gives computers the ability to learn without being explicitly programmed."

The definition mentioned refers to the ability to learn from experience, which humans do in most cases.

ML is an interdisciplinary domain forged at the crossroads and harmonious blending of computer science, statistics, neurobiology, and control theory. Its practical applications have successfully surmounted various challenges across diverse fields, fundamentally altering the paradigm of software development. In essence, ML constitutes a foundational approach that empowers computers with a degree of autonomy. It is evident that ML draws its inspiration from the study of human learning; just as the human brain and its neurons underlie intuition, artificial neural networks serve as the bedrock for computer decision-making. ML facilitates the creation of models that can accurately depict data patterns through a meticulous analysis of datasets.

As an illustration, we can establish a connection between input variables and output variables within a given system. One approach to achieving this is by assuming the existence of a mechanism for generating parametric data, albeit without precise knowledge of the parameter values. This procedure is commonly referred to as employing statistical methods.

Analysis of logical reasoning

Logical reasoning is based on the concepts of induction, deduction, and inference. Here are the differences between them:

- **Induction** is a reasoning process that starts from specific observations or data to arrive at general conclusions. In other words, it is about extracting a general rule or principle based on specific examples. Induction can be useful for making predictions and generalizations, but the conclusions obtained are not necessarily certain.

- **Deduction** is a reasoning process that starts from general premises or rules and arrives at specific conclusions. In other words, it is about applying general principles to obtain specific information. The deduction is based on formal logic and the use of valid inference rules. The deduction produces logically correct conclusions from the given premises.
- Finally, **inference** is a reasoning process that leads us to draw conclusions, deductions, or judgments based on available evidence or information. Inference can involve both induction and deduction as well as other forms of reasoning, such as abduction. Inference can be viewed as the application of reasoning rules or strategies to obtain new information or arrive at conclusions based on existing ones.

In summary, induction is based on extracting general principles from specific examples, deduction is based on applying general principles to obtain specific conclusions, while inference is a broader term that encompasses both induction and deduction, as well as other reasoning processes.

In the realm of ML, the teacher and the learner are two pivotal entities that play significant roles. The teacher possesses the necessary knowledge to execute a given task, while the learner's objective is to acquire that knowledge to perform the task. The strategies employed for learning can be distinguished based on the level of inference carried out by the learner, considering two extremes: **no inference** and **substantial inference**.

When a computer system (the learner) is directly programmed, it acquires knowledge without engaging in any inference, as all cognitive processes are handled by the programmer (the teacher). Quite the opposite, when a system finds new solutions autonomously, it necessitates a substantial amount of inference. In this scenario, organized knowledge is derived through experiments and observations. In the following figure, the difference between induction and deduction for inference tasks is shown.

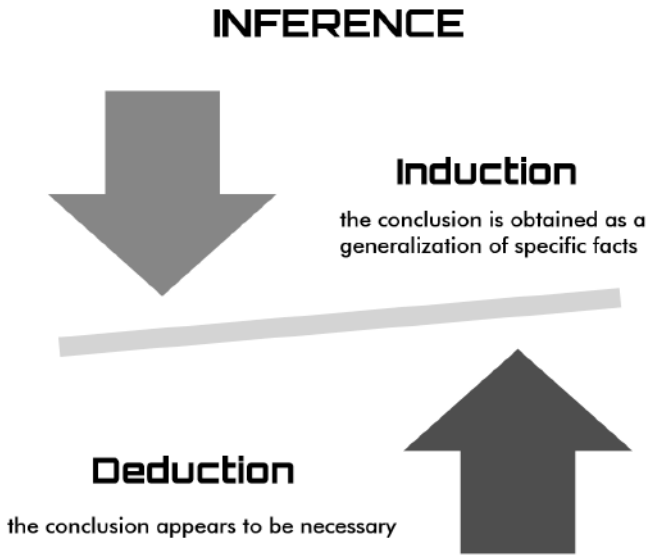


Figure 1.1 – Logical reasoning keys

In between induction and deduction is a midpoint called inference – for example, say a student seeks to work out a math challenge by drawing analogies to solutions provided in a workbook. This activity demands inference, albeit to a lesser extent than the discovery of a new mathematical theorem.

By increasing the learner's capacity for inference, the drain on the teacher diminishes. The following taxonomy of ML endeavors to portray the concept of trade-offs in terms of work demanded by both the learner and the teacher.

Learning strategy typologies

Based on the inference performed, we can identify different mechanisms that a learning system can adopt. Let's see some of them:

- **Rote learning**: Let's start with a basic one, rote learning, which involves the use of content without any transformation being carried out on it. No inference process is activated, and the knowledge is not processed in any way; it is essentially a memorization process programmed with considerable effort on the part of the learner.
- **Learning from instruction**: When acquiring knowledge from a teacher, the learner needs to convert the information presented into a format that can be internally processed. Additionally, it is crucial for the learner to integrate the new knowledge with their existing understanding to effectively utilize it. This process involves some level of inference on the learner's part, but a significant portion of the responsibility lies with the teacher. The teacher must present and organize the knowledge in a manner that progressively enhances the student's existing knowledge. This aligns with the approach used in most formal education methods. Consequently, the task of ML involves developing a system that can receive instruction or advice, store it, and apply the acquired knowledge effectively.

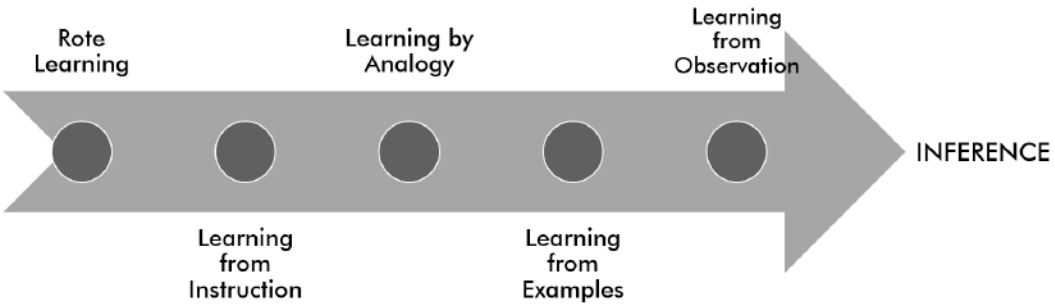


Figure 1.2 – Learning strategy typologies based on the inference performed

- **Learning by analogy**: When acquiring new statements or skills, the process involves converting and enhancing existing knowledge that shares a strong resemblance to the new setting. This allows the transformed knowledge to be effectively applied in the new context. For example, a learning-by-analogy system could be utilized to transform code into something that implements

a closely related function, even if it wasn't originally designed for that purpose. Learning by analogy needs more inference on the part of the learner compared to the previous learning mechanism. The learner must retrieve a relevant fact or skill from their memory that is analogous in terms of relevant parameters. Then, the retrieved knowledge needs to be converted, related to the new context, and archived for future use.

- **Learning from examples**: When confronted with a set of instances, including both examples that support an idea and counterexamples that contradict it, the learner infers a generalized description of the idea. This description encompasses all the positive examples while systematically removing the negative ones. Learning from examples has been extensively studied in the field of artificial intelligence. In this method, the learner engages in a higher degree of inference compared to learning from instruction, as there are no general ideas provided by a teacher. It also involves a slightly greater level of inference than learning by analogy since there are no related ideas provided as starting points for developing the new concept.
- **Learning from observation**: This form of inductive learning is highly versatile and encompasses various tasks, such as discovery systems, theory formation, and establishing grouping principles for creating taxonomic hierarchies. Unlike other approaches discussed, this form of learning does not rely on an external teacher. Instead, the learner is required to engage in extensive inference. They are not presented with a specific set of examples for a specific idea, nor do they have access to a prediction that can group instances produced in an automated way as positive or negative examples of any given concept. Furthermore, instead of focusing on a single idea, the learner must handle multiple concepts simultaneously, which introduces a significant challenge in terms of attention allocation.
- **Learning by chunking**: This is a cognitive strategy that involves breaking down information into smaller, manageable units or "chunks" to enhance memory and comprehension. This technique capitalizes on the brain's natural tendency to organize and process information in meaningful clusters. By grouping related concepts together, learners can more easily absorb and retain complex material. Chunking is particularly effective in various educational settings, from memorizing lists and sequences to mastering intricate subjects. When information is organized into cohesive chunks, it becomes easier to grasp the relationships between different components, facilitating a deeper understanding of the overall content. This approach is especially beneficial in fields such as language acquisition, where breaking down phrases or words into manageable chunks aids in faster and more efficient learning. Moreover, chunking promotes efficient recall and problem-solving. Instead of struggling to remember individual pieces of information, learners can access broader chunks of knowledge, leading to quicker and more accurate retrieval. This cognitive strategy aligns with the brain's capacity to process information in parallel, optimizing the learning process and enhancing overall cognitive performance. In essence, learning by chunking empowers individuals to navigate the complexities of information with greater ease and effectiveness.

Analyzing the learning mechanisms that a system can adopt will help us better understand the different types of learning processes that can be adopted in an ML-based system.

Discovering the different types of learning processes

Learning is based on the idea that perceptions should not only guide actions but also enhance the agent's ability to automatically learn from interactions with the world and the decision-making processes themselves. A system is considered capable of learning when it has an executive component for making decisions and a learning component for modifying the executive component to improve decisions. Learning is influenced by the components learned from the system, by the feedback received after the actions are performed, and by the type of representation used.

ML offers several ways of allowing algorithms to learn from data, which are classified into categories based on the type of feedback on which the learning system is based. Choosing which learning category to use for a specific problem must be done in advance to find the best solution. It is useful to evaluate the robustness of the algorithm, such as its ability to make correct predictions even with missing data, the scalability, the efficiency of the algorithm with small or large datasets, and the interpretability, which refers to the possibility of attributing a more subjective than objective outcome.

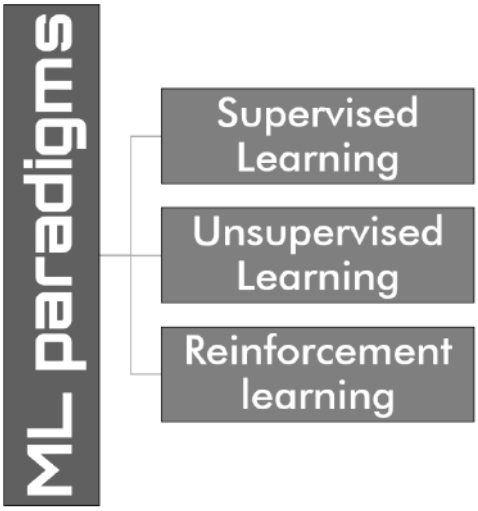


Figure 1.3 – Different types of learning processes

ML algorithms can be categorized according to the type of experience they are subjected to during the learning process. It is common to categorize the paradigms of ML as follows:

- **Supervised learning**: The algorithm generates a function that links input values to a desired output, through the observation of a set of examples in which each data input has its relative output data that is used to construct predictive models.
- **Unsupervised learning**: The algorithm tries to derive knowledge from a general input, without the help of a set of pre-classified examples, which is used to build descriptive models. A typical example of the application of these algorithms is search engines.

- **Reinforcement learning**: The algorithm can learn depending on the changes that occur in the environment in which it is performed. The agent receives feedback in the form of rewards or punishments based on the actions it takes, allowing it to learn optimal strategies over time. In fact, since every action has some effect in the environment concerned, the algorithm is driven by the same feedback environment. Some of these algorithms are used in self-driving cars and autonomous AlphaGo gaming.

Let's see these categories in detail, analyzing their characteristics and trying to understand how to choose the most suitable paradigm for our problem.

Supervised learning

Supervised learning is an ML technique designed to enable a computer system to automatically solve tasks. The process involves providing input data, typically in the form of vectors, which form a set, *I*. The output data is defined as a set, *O*, and a function, *f*, is established to associate each input with the correct answer. This set of data used for training is referred to as the training set.

$$O = f(I)$$

The underlying principle of all supervised learning algorithms is that with enough examples, an algorithm can create a derived function, *B*, that approximates the desired function, *A*. If the approximation is accurate enough, the derived function should provide output answers like those of the desired function, making them acceptable. These algorithms rely on the assumption that similar inputs correspond to similar outputs. This assumption is often not perfectly met. However, there are situations where this approximation is acceptable.

With this type of learning, it is possible, for example, to use correctly labeled images of animals as input, to make the algorithm learn the correlation between the characteristics of a specific animal and the image containing it. This is done in such a way that, subsequently, the algorithm can recognize it, given the input of an image containing that type of animal. The algorithm learns to recognize a certain pattern to understand the correlation between the image and the assigned label, to then deduce a general rule with which to recognize whether the same correlation exists in the subsequent data.

To use supervised learning, it is therefore necessary to have data with known outputs. In the case of fully observable environments, the system can see what effects its actions have and can use the supervised learning method to learn to predict them. If the environment were to be not completely observable, the immediate effects could be invisible. The performance of these algorithms is highly dependent on the input data. If only a few training inputs are provided, the algorithm may lack sufficient experience to produce correct outputs. On the other hand, an excessive number of inputs can make the algorithm excessively slow, as the derived function becomes more complex.

Furthermore, supervised learning algorithms are sensitive to noise. Even a small number of incorrect data points can render the entire system unreliable, leading to erroneous decisions.

When the output value is categorical, such as membership/non-membership in a certain class, it is considered a classification problem. On the other hand, if the output is a continuous real value within a certain range, it is classified as a regression problem.

Unsupervised learning

The objective of unsupervised learning is to automatically extract information from databases without prior knowledge of the content to be analyzed. Unlike supervised learning, there is no information available regarding membership classes or the output corresponding to a given input. The goal is to obtain a model capable of discovering interesting properties, such as groups (clusters) of examples with similar characteristics (clustering). An example of the application of these algorithms is seen in search engines. By providing one or more keywords, they can generate a list of links related to our search. The differences between supervised learning and unsupervised learning are highlighted in the following figure.

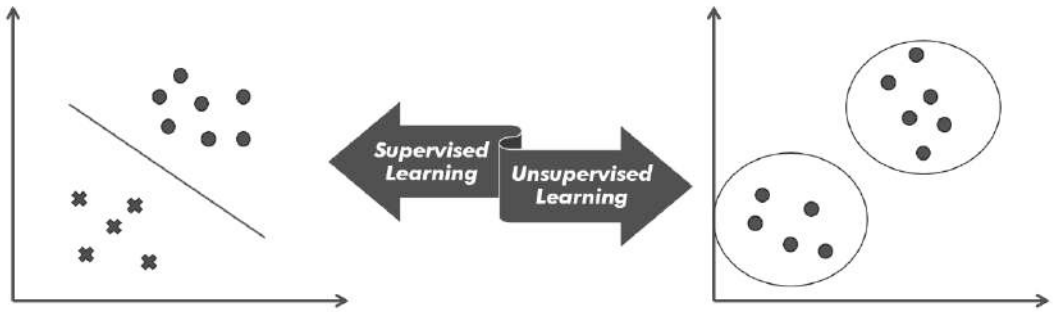


Figure 1.4 – Supervised learning versus unsupervised learning

The objective of unsupervised learning is to autonomously learn the underlying structure of the input data. The outcomes are influenced by decisions regarding which data to input and the order in which they are presented. This approach reorganizes the data in a different manner, creating more meaningful data clusters for subsequent analyses and enabling the discovery of previously unnoticed patterns in the data. A system capable of pure unsupervised learning lacks the ability to learn what actions to take, as it lacks information about what constitutes a correct action or a desirable state to achieve.

Unsupervised learning is commonly employed in clustering problems, where it identifies groups of data based on shared characteristics, resulting in the creation of data clusters. Additionally, this approach is utilized in association learning, which involves identifying associative rules among the input data and in recommendation systems.

The effectiveness of these algorithms hinges on the value of the information they can derive from databases. These algorithms function by scrutinizing data and actively seeking out resemblances or disparities among them. The data at hand pertains exclusively to the array of attributes that characterize each individual example.

Unsupervised learning demonstrates high efficiency when working with numeric elements, but their accuracy decreases when dealing with non-numeric data. Typically, these algorithms perform well when the data exhibits a clear order or distinct grouping that can be easily identified.

Reinforcement learning

Reinforcement learning aims to develop algorithms that can learn and adapt to changes in their environment. This programming technique is based on the concept of receiving external stimuli based on the choices made by the algorithm. Making a correct choice results in a reward, while an incorrect choice leads to a penalty. The ultimate objective of the system is to achieve the best possible outcome.

Unlike supervised learning, where a teacher provides the system with correct outputs (learning with guidance), this is not always feasible. Often, only qualitative information is available, sometimes in a binary form (such as right/wrong or success/failure). These available pieces of information are referred to as reinforcement signals. However, the system does not provide any guidance on how to update the agent as an optimization algorithm and the environment as being characterized by the family of objective functions that we'd like to learn an optimizer for. The goal of the system is to create "intelligent" agents that possess the ability to learn from their experiences.

Reinforcement learning algorithms operate based on rewards that are assigned according to achieved objectives. These rewards are essential for understanding what actions are desirable and should be pursued, as well as what actions should be avoided. Without rewards, the algorithm would struggle to make decisions. Reinforcement learning is employed when algorithms need to make decisions that have consequences. It goes beyond being purely descriptive and becomes prescriptive, providing guidance on what actions to take. This type of learning is highly innovative in the field of ML and is well suited for understanding how environments work. The flow of the reinforcement learning algorithm is shown in the following figure.

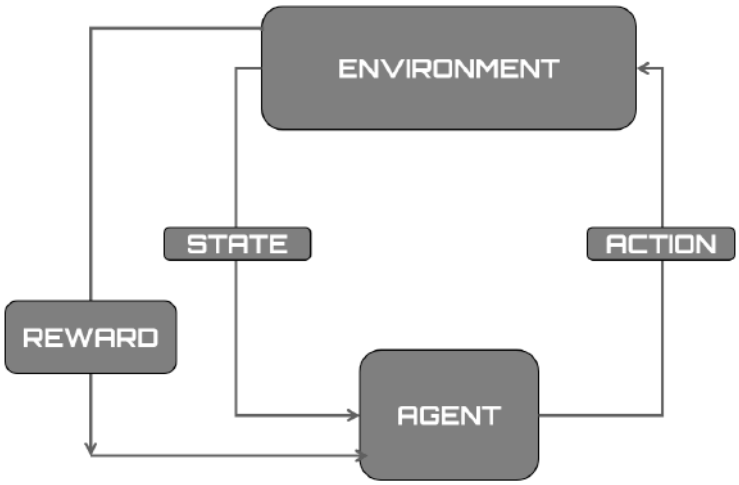


Figure 1.5 – Reinforcement learning mechanism

A crucial aspect is the algorithm's ability to interpret feedback as reinforcement rather than new input. The goal of reinforcement learning is to utilize received feedback to construct an optimal policy, maximizing the total expected reward. A particular policy represents a particular update formula. Hence, learning the policy is equivalent to learning the update formula, and hence the optimization algorithm. This approach allows training an algorithm without explicitly defining rules deduced from feedback. It is particularly useful in complex domains where defining rules can be challenging, and performance can be greatly improved.

Reinforcement learning encompasses both passive learning and active learning. In passive learning, the agent follows a fixed policy and aims to learn the utilities of state-action pairs. In some cases, the agent may also need to learn the model of the environment. In active learning, the agent must learn what actions to take and, through exploration, gain experience on how to behave within the environment. One way to implement this strategy is by having the algorithm play a game without providing it with the rules. Positive feedback is given when the agent takes allowed or beneficial actions, while negative feedback is provided for undesirable actions. This allows the algorithm to learn optimal moves during the game without explicit knowledge of the game rules. The agent learns to behave successfully in its environment solely based on the received feedback and without prior knowledge.

Alongside the three pillars of the learning paradigms just seen (supervised learning, unsupervised learning, and reinforcement learning) are some new typologies derived from these approaches. Let's see some of them.

Semi-supervised learning

Semi-supervised learning combines the strengths of both supervised and unsupervised learning. Initially, the supervised approach is employed by providing inputs along with their corresponding outputs. Then, additional similar inputs are introduced without their associated output references. These inserted inputs and outputs contribute to the creation of a general model, which can be utilized to extrapolate outputs for the remaining inputs.

The process begins with the supervised phase, where the labeled data is used to train a model. This model learns to map the inputs to their corresponding outputs based on the provided labels. Once the initial model is trained, it can be used to make predictions on the unlabeled data.

During the unsupervised phase, the model uses the unlabeled data to extract patterns, structures, or relationships within the data. This can be done through techniques such as clustering, dimensionality reduction, or generative modeling. By leveraging the unlabeled data, the model can gain a better understanding of the underlying distribution of the data and potentially improve its performance.

The goal of semi-supervised learning is to use the knowledge gained from the labeled and unlabeled data to create a more robust and accurate model. By combining the labeled data's explicit guidance with the unsupervised learning's ability to capture hidden patterns, semi-supervised learning can be a powerful approach, especially in scenarios where obtaining labeled data is expensive or time-consuming.

Transfer learning

Transfer learning offers the ability to transfer the acquired knowledge from addressing one problem to effectively tackling a similar problem. The significant advantage of reusing knowledge is evident, although it may not always be feasible due to the need to adapt many ML algorithms to the specific case at hand.

Transfer learning involves leveraging knowledge gained from solving one problem and applying it to a different but related problem. In transfer learning, a pre-trained model that has been trained on a large dataset is used as a starting point for solving a new task or problem.

The basic idea behind transfer learning is that the features learned by a model on one task can be useful for another task. Instead of starting the learning process from scratch, transfer learning allows us to transfer the knowledge and representations acquired by the pre-trained model to the new task. Using such an approach can lead to a substantial reduction in the required amount of training data and computational resources for the new task.

There are typically two main approaches to transfer learning:

- **Fine-tuning**: In this approach, the pre-trained model is taken and further trained on the new task with a smaller dataset specific to the new task. The idea is to adjust the parameters of the pre-trained model to make it more relevant to the new problem while retaining the knowledge it has already learned.
- **Feature extraction**: With this approach, the pre-trained model is used as a fixed feature extractor available for deep learning models. The earlier layers of the pre-trained model are frozen, and only the later layers are replaced or modified to adapt to the new task. The new data is passed through the pre-trained model, and the output features from the last few layers are extracted and used as input for a new classifier or model specific to the new task.

Transfer learning can be particularly effective when the pre-trained model has been trained on a large and diverse dataset, as it tends to learn generic features that are useful for a wide range of tasks. By leveraging pre-existing knowledge, transfer learning enables faster training, better performance, and improved generalization on new tasks, especially when labeled data for the new task is limited or unavailable.

After introducing the different types of learning paradigms, we can move on to analyzing how to approach a problem using techniques based on ML.

Using ML techniques

In the previous section, we explored the various types of ML paradigms in detail. So, we have understood the basic principles that underlie the different approaches. At this point, it is necessary to understand what the elements that allow us to discriminate between the different approaches are; in other words, in this section, we will understand how to adequately choose the learning approach necessary to obtain our results.

Selecting the ML paradigm

Selecting the appropriate ML algorithm can feel overwhelming given the numerous options available, including both supervised and unsupervised approaches, each employing different learning strategies.

There is no universally superior method, nor one that fits all situations. In large part, the search for the right algorithm involves trial and error; even seasoned data scientists cannot determine whether an algorithm will work without testing it. Nonetheless, the algorithm choice is also influenced by factors such as the data format and type, the desired information to be extracted, and how that information will be utilized.

Here are some guidelines to assist in selecting the most suitable approach:

- Opt for supervised learning when the objective is to train a model for making predictions, such as determining future values of a continuous variable such as barometric pressure or stock prices, or performing classification tasks such as identifying vehicle types from webcam footage.
- Choose unsupervised learning when the aim is to analyze data and develop a model that discovers meaningful internal representations, such as clustering the data into distinct groups.
- Select reinforcement learning when the objective is to achieve a goal within an uncertain environment where all variables cannot be predicted. This approach is valuable when there are multiple ways to accomplish a task, but certain rules need to be followed. An example is autonomous driving, where adherence to traffic regulations is essential.

By considering these factors and aligning them with the specific goals and characteristics of the data, one can make a more informed choice when selecting an appropriate ML paradigm.

The selection of the learning paradigm and the specific algorithm is clearly dependent on the characteristics of the data we are working with. This includes factors such as data size, quality, and nature, as well as the desired outcome and the implementation details of the algorithm.

There is no universally superior method or one-size-fits-all solution. The only way to ascertain the suitability of an algorithm is to experiment and evaluate its performance using appropriate metrics.

However, we can conduct a preliminary analysis to better understand the approach that aligns with our requirements. We start by considering what we have (the data), the available tools (algorithms), and the goals we aim to achieve (results). Through this analysis, we can gather valuable information to guide our decision-making process. The following figure shows the different learning paradigms and the activities that can be addressed.

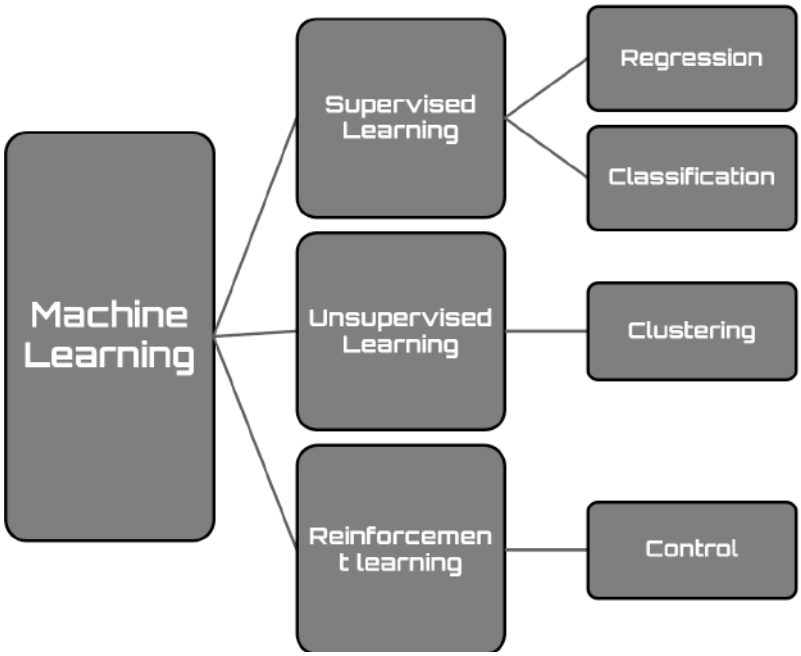


Figure 1.6 – ML algorithm classification

Let's begin with the data and classify its characteristics. This classification helps us determine the following options:

- **Classification based on input**: If we have labeled input data, it indicates a supervised learning problem. If labeling is not available but we seek to uncover the system's structure, it indicates an unsupervised learning problem. Finally, if our goal is to optimize an objective function through interactions with the environment, it indicates a reinforcement learning problem.
- **Classification based on output**: If the model's output is a numerical value, it suggests a regression problem. If the output is categorical, it indicates a classification problem. If the output involves grouping the input data, it indicates a clustering problem.

Once we have classified the problem, we can explore the available tools to solve it. This involves identifying applicable algorithms and focusing our study on the methods required to implement these tools for our specific problem.

After identifying the tools, we need to evaluate their performance. This can be accomplished by applying the selected algorithms to our datasets. By carefully selecting evaluation metrics, we can compare the performance of each algorithm.

By following this process of data analysis, problem classification, tool identification, and performance evaluation, we can make informed decisions and choose the most suitable algorithm for our needs.

Step-by-step guide on how to build ML models

Once the algorithm being applied to our data has been chosen, it is essential to establish a well-defined workflow before diving into the task at hand. Before embarking on the actual implementation, it is crucial to allocate some time to set up the workflow. When developing an ML-based application, this procedure typically involves the following steps:

1. **Define the problem**: Clearly articulate the problem you want to solve with ML. Determine the specific task, such as classification, regression, or clustering, and understand the objectives and requirements.
2. **Collect and preprocess data**: Collect relevant data for your problem. Ensure the data is of high quality, representative, and covers a wide range of scenarios. Preprocess the data by handling missing values and outliers and performing data normalization or feature scaling and data cleaning. The foundation of any data-driven process lies in the data itself, and it is natural to question the origin of such vast amounts of data. Data collection involves a variety of methods, often through extensive procedures such as measurement campaigns or face-to-face interviews. Regardless of the specific method employed, the collected data is typically stored in a database, ready to be analyzed and transformed into valuable insights and knowledge.
3. **Split the data**: Data splitting is a crucial step in ML to effectively evaluate and validate the performance of models. It involves dividing the available data into distinct subsets for training, validation, and testing purposes. A largely used data splitting technique, **train-validation-test split**, allows the division of data into training, validation, and testing sets as follows:

    I. **Training set**: The largest portion of the data used for model training. It is utilized to optimize the model's parameters and learn patterns from the data.

    II. **Validation set**: A smaller subset of the data used to fine-tune the model's hyperparameters and assess its performance during development. It helps in avoiding overfitting and selecting the best-performing model.

    III. **Test set**: A separate portion of the data used as a final evaluation measure to assess the model's generalization and performance on unseen examples. It provides an unbiased estimate of the model's capabilities in real-world scenarios. The following flow chart shows the entire workflow of an ML model.

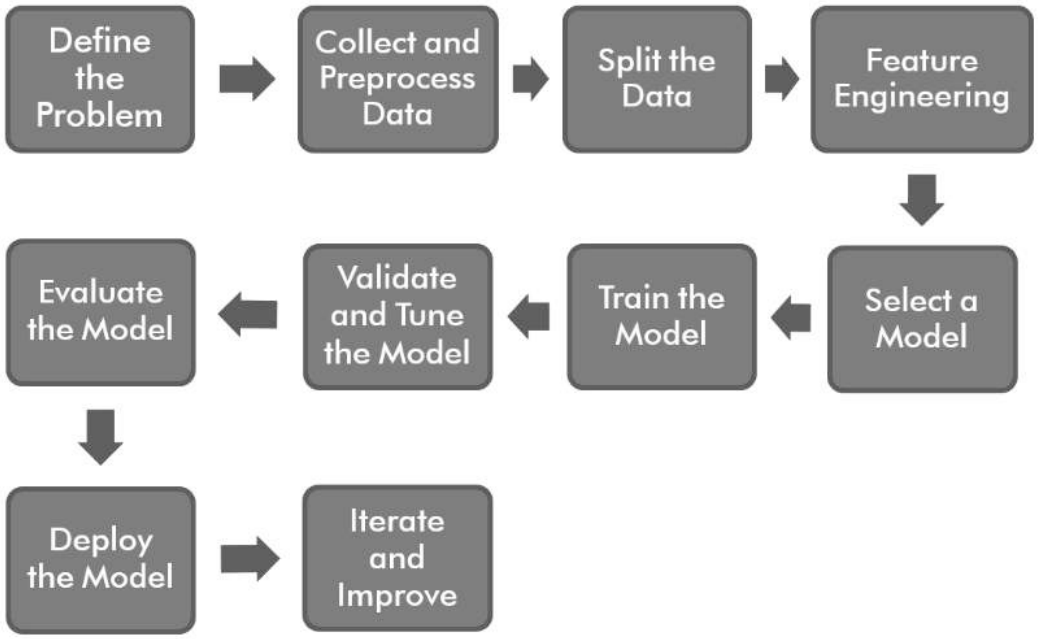


Figure 1.7 – ML model implementation workflow

4. **Feature engineering**: Extract or create meaningful features from the available data that can help the model learn patterns and make accurate predictions. This may involve transforming or combining existing features, encoding categorical variables, or generating new features based on domain knowledge. Essentially, there are two main approaches: **feature extraction** and **selection**. Feature extraction involves transforming raw data into a set of meaningful features that can effectively represent the underlying patterns and relationships in the data. It aims to capture the most relevant information and discard irrelevant or redundant data. This process can be particularly useful when dealing with high-dimensional or unstructured data. Feature selection involves selecting a subset of the most informative and relevant features from the available set of features. It helps in reducing dimensionality, improving model interpretability, and avoiding overfitting.
5. **Select a model**: Choose an appropriate ML model based on the type of problem, data characteristics, and available resources. We have seen which considerations it is necessary to make for the choice of the learning paradigm: *An adequate analysis is also necessary for the choice of the type of algorithm to adopt*. In fact, there is not just one way to go; there are many, and each solution has strengths and weaknesses. Consider algorithms such as decision trees, helper vector machines, neural networks, or ensemble methods. Research and compare different models to identify the most suitable one, that is, the one that best fits the available data and that best allows us to make an adequate inference and extract adequate knowledge.

6. **Train the model**: Now, the process becomes more substantial. At this stage, ML takes center stage as we define the model and proceed with training. The selected model begins the task of extracting valuable insights from the extensive volume of available data, uncovering new knowledge that was previously unknown to us. This marks the beginning of the learning journey, where the model dives deep into the data to uncover patterns, relationships, and hidden information that has yet to be revealed. Train the selected model using the training data. This involves feeding the input features and their corresponding target values to the model and adjusting its internal parameters through an optimization algorithm. The model learns from the training data to make accurate predictions.
7. **Validate and tune the model**: In this phase, we use the knowledge gained from the previous step of the test to determine the effectiveness of the model. Evaluating an algorithm involves assessing how closely the model approximates the real-world system. In supervised learning, we have known values that allow us to evaluate the algorithm's performance. In unsupervised learning, alternative metrics are employed to gauge success. If the results are unsatisfactory, we can revisit the preceding steps, make necessary adjustments, and retest the model until we achieve the desired outcome. This iterative process allows for refinement and improvement as we strive to develop a more accurate and reliable model. Evaluate the model's performance on the validation set. Use suitable evaluation metrics for your specific problem, such as accuracy, precision, recall, or mean squared error. Fine-tune the model by adjusting hyperparameters, such as learning rate, regularization, or number of layers, to optimize its performance.
8. **Evaluate the model**: Now, we have reached the critical juncture where we can put our work into action. It's time to apply the model we have built to real-world data and evaluate its ability to approximate the desired outcomes. The model, having undergone thorough training and testing, is now assessed and valued in this phase. By deploying the model on real data, we can observe its performance, analyze its predictions, and assess how well it aligns with our expectations. This stage allows us to gauge the practicality and effectiveness of the model in real-world scenarios, providing valuable insights for further improvements and optimizations. Once the model is tuned, assess its performance on the testing set, as this offers an impartial estimation of the model's performance on fresh, unseen data. Evaluate the model's metrics and analyze its strengths and weaknesses. Consider additional techniques such as cross-validation for a more robust evaluation.
9. **Deploy the model**: If the model meets the desired performance criteria, deploy it for real-world use. This involves integrating the model into your application or system and ensuring its compatibility with the production environment. Implement mechanisms to monitor and maintain the model's performance over time. The ultimate objective of building an ML application is to solve a problem, and this can only be achieved when the ML model is actively utilized in a production setting. Consequently, ML model deployment holds equal importance to the development phase. Deployment entails transitioning a trained ML model from an offline environment to integration within an existing production system, such as a live application. It represents a critical stage that must be completed to enable the model to fulfill its intended purpose and address the challenges it was designed for. Deployments establish an online

learning framework where the model is continuously updated with new data, enabling ongoing improvement. The precise process of deploying an ML model varies depending on factors such as the system environment, model type, and organizational DevOps practices. Nonetheless, the general deployment process, particularly in a containerized environment, can be summarized into four key steps, which will be elaborated upon later.

10. **Iterate and improve**: At last, we can shift our attention to the final tasks. After confirming the functionality of the model and assessing its performance, we are now prepared to thoroughly analyze the results to identify any areas that can be further enhanced or optimized. This stage involves a comprehensive examination of the model's performance metrics, exploring potential shortcomings, and seeking opportunities for improvement. By delving into the analysis, we can gain valuable insights that will guide us in refining the model, addressing any limitations, and maximizing its efficacy. This meticulous evaluation process is vital for achieving a high-performing and robust ML solution. ML is an iterative process. Continuously gather feedback, monitor the model's performance, and collect new data to retrain and refine the model. As new insights are gained or requirements change, update the model accordingly to ensure it remains effective and accurate.

By following these steps, you can systematically build ML models and iteratively improve their performance to solve complex problems and make accurate predictions or decisions.

Now, our basic understanding of ML and how to choose the learning paradigm and algorithm that best fits our data is adequate. The next step involves an initial exploration of the resources that the MATLAB environment makes available to us to build our ML-based applications.

Exploring MATLAB toolboxes for ML

Up until now, we have acquired knowledge about the functions and capabilities of ML algorithms. We have also gained an understanding of how to identify various types of algorithms, select the appropriate solution for our requirements, and establish an effective workflow. Now, it is time to delve into the process of executing these tasks within the MATLAB environment.

With MATLAB, the process of solving ML problems becomes remarkably straightforward. The comprehensive set of tools and functionalities provided by MATLAB empowers users to leverage various algorithms and techniques effortlessly. Whether you are a beginner or an experienced practitioner, MATLAB equips you with the necessary resources to dive into the world of ML with confidence.

MATLAB is a software platform specifically designed to address scientific problems and facilitate design processes. It offers an integrated environment where calculations, visualizations, and programming seamlessly come together. This user-friendly environment allows problems and their solutions to be expressed using familiar mathematical notation.

The name MATLAB is an acronym for **Matrix Laboratory**, highlighting its initial focus on providing convenient access to matrix-related software. Over the years, MATLAB has evolved through valuable user feedback and input. The MATLAB programming language is centered around matrices, which naturally represent computational mathematics. This matrix-based approach allows for an intuitive and efficient representation of mathematical operations.

The desktop environment of MATLAB encourages experimentation, exploration, and discovery. Its integrated graphics capabilities facilitate clear visualization and enable users to gain a comprehensive understanding of the underlying data. With MATLAB, users are empowered to engage in a wide range of scientific and computational tasks with ease and confidence.

In addition, MATLAB is known for its extensive collection of specialized solutions known as toolboxes. These toolboxes offer specific solutions to application problems and serve as invaluable resources for a wide range of users. They provide the necessary foundations and tools to apply MATLAB's capabilities to various specialized domains and technologies. Each toolbox consists of a curated set of MATLAB functions, commonly referred to as M-files, which extend the functionality of the MATLAB environment. These toolboxes cater to specific problem domains, enabling users to solve complex and specialized challenges with ease, making MATLAB a versatile and comprehensive platform for a diverse range of applications.

So, let's see some toolboxes that offer us important resources to be able to adequately build our applications based on ML.

Statistics and Machine Learning Toolbox

The toolbox encompasses a wide range of functions and applications tailored for data description, analysis, and modeling. It facilitates exploratory data analysis through descriptive statistics, visualizations, and clustering techniques. Probability distributions can be fitted to data, random numbers can be generated for Monte Carlo simulations, and hypothesis tests can be performed. The toolbox includes classification and regression algorithms, allowing for data inference and the construction of predictive models. Both interactive methods, using the Regression Learner and Classification Learner applications, and programmatically through AutoML can be employed.

For feature extraction and analysis of multidimensional data, the toolbox provides techniques such as **principal component analysis** (**PCA**), feature selection techniques, dimensionality reduction methods, and regularization. These methods assist in identifying variables with the most significant predictive power.

The toolbox encompasses a diverse set of unsupervised, supervised, and semi-supervised ML algorithms, including **support vector machines** (**SVMs**), boosted decision trees, shallow neural networks, and k-means clustering. Interpretability techniques, such as partial dependency graphs, Shapley, and LIME values, are available, and C/C++ code can be automatically generated for embedded deployment. Native Simulink building blocks enable the integration of predictive models with model-based design and simulations. Moreover, many algorithms within the toolbox are designed to handle datasets that are too large to be stored entirely in memory, allowing for efficient processing of large-scale data.

This tool offers a comprehensive set of features that cover key areas in the field of ML. Here are the main topics and features it encompasses:

- **Regression techniques**: This includes a wide range of regression models such as linear, generalized linear, nonlinear, robust, regularized, ANOVA, repeated measures, and mixed-effects models.
- **Big data algorithms**: The tool provides algorithms specifically designed for big data analysis, including dimension reduction, descriptive statistics, k-means clustering, linear regression, logistic regression, and discriminant analysis.
- **Probability distributions**: Both univariate and multivariate probability distributions are available, along with random and quasi-random number generators and Markov chain samplers.
- **Hypothesis tests and design of experiments** (**DOE**): The tool offers hypothesis tests for distributions, dispersion, and location. It also provides techniques for designing optimal, factorial, and response surface experiments.
- **Classification Learner app and algorithms**: For supervised ML, the tool provides the Classification Learner app, which allows for interactive exploration and modeling. It also offers algorithms such as SVMs, boosted and bagged decision trees, k-nearest neighbor, Naïve Bayes, discriminant analysis, and Gaussian process regression.
- **Unsupervised ML algorithms**: The tool includes unsupervised ML algorithms, such as k-means clustering, k-medoids, hierarchical clustering, Gaussian mixtures, and hidden Markov models.
- **Bayesian optimization**: It provides Bayesian optimization techniques to search for optimal hyperparameters and fine-tune ML algorithms.

These features collectively make the tool a comprehensive solution for various ML tasks and enable users to effectively explore, analyze, and model data in diverse scenarios.

Deep Learning Toolbox

The Deep Learning Toolbox is a powerful resource for designing and implementing deep neural networks. It provides a range of algorithms, pre-trained models, and applications to facilitate the development of deep learning solutions.

Here are the key features of the Deep Learning Toolbox:

- **Convolutional neural networks** (**ConvNet, CNN**) and **long short-term memory** (**LSTM**) **networks**: These network architectures enable classification and regression tasks on images, time series, and textual data.
- **Network architectures**: The toolbox supports various network architectures, including **generative adversarial networks** (**GANs**) and Siamese networks. These architectures can be built using automatic differentiation, custom training cycles, and shared weights.

- **Deep Network Designer app**: With the Deep Network Designer app, users can design, analyze, and train networks using a graphical interface. This allows intuitive network exploration and customization.
- **Experiment Manager app**: The Experiment Manager app simplifies the oversight of numerous deep learning experiments. It assists in monitoring training metrics, evaluating outcomes, and contrasting code across various experiments, delivering an efficient workflow.
- **Integration with other frameworks**: The Deep Learning Toolbox offers seamless integration with popular deep learning frameworks such as TensorFlow 2, TensorFlow-Keras, PyTorch, **Open Neural Network Exchange** (**ONNX**), and Caffe. This allows for the easy import and export of networks, layer plots, and models.
- **Transfer learning and pre-trained models**: The toolbox supports transfer learning by providing access to pre-trained models such as DarkNet-53, ResNet-50, NASNet, and SqueezeNet. These models can be leveraged to expedite the development process and achieve excellent performance.

Utilizing a pre-trained image classification neural network allows you to leverage its already acquired knowledge of extracting meaningful features from natural images. This serves as a valuable starting point for learning a new task. ImageNet databases are usually used to train pre-trained neural networks; this is data largely employed in the **ImageNet Large-Scale Visual Recognition Challenge** (**ILSVRC**). These algorithms have been trained on a dataset comprising more than a million images and are capable of categorizing images into 1,000 distinct object categories. Adopting a pre-trained neural network and applying transfer learning is generally more efficient and less time-consuming compared to training a neural network from scratch.

The Deep Learning Toolbox empowers users to work efficiently in deep learning projects, offering advanced capabilities, visualization tools, and compatibility with various frameworks and pre-trained models.

Reinforcement Learning Toolbox

The Reinforcement Learning Toolbox offers an application, functions, and a Simulink block designed for training policies using reinforcement learning algorithms. These algorithms can be utilized to develop controllers and decision-making algorithms for complex applications such as resource allocation, robotics, and autonomous systems.

With this toolbox, you can represent policies and value functions using deep neural networks or look-up tables and train them by interacting with environments modeled in MATLAB or Simulink. The reinforcement learning algorithms provided in the toolbox can be evaluated with one or more agents included, or you can create your own custom agents. You have the flexibility to experiment with hyperparameter settings, monitor training progress, and simulate trained agents interactively using the application or programmatically. To enhance training performance, simulations can be executed in parallel across multiple CPUs, GPUs, computer clusters, and the cloud.

Additionally, the ONNX template format allows you to import existing policies from popular deep learning frameworks such as TensorFlow Keras and PyTorch (with the Deep Learning Toolbox). You can also generate optimized C, C++, and CUDA code to deploy trained policies on microcontrollers and GPUs. The toolbox includes reference examples to assist you in getting started on your reinforcement learning journey.

In the context of reinforcement learning, the environment plays a crucial role as it models the dynamics in which the agent operates. The environment follows these key principles:

- **It receives actions from the agent**: The agent selects and sends actions to the environment based on its policy or decision-making strategy.
- **It generates observations**: The environment produces observations that reflect the state of the system or environment after the agent's action. These observations capture the dynamic behavior and provide feedback to the agent.
- **It provides rewards**: The environment assigns rewards to the agent based on the outcomes of its actions. Rewards measure how well the agent's action contributes to achieving the desired task or goal.

In MATLAB, you have the flexibility to create both predefined and custom environments for your reinforcement learning scenarios. This allows you to model the specific dynamics and interactions that your agent will encounter in the learning process. With MATLAB, you can define the rules, constraints, and characteristics of the environment, enabling you to tailor it to your specific problem domain.

Computer Vision Toolbox

The Computer Vision Toolbox in MATLAB offers a wide range of capabilities for designing and testing computer vision, 3D vision, and video processing systems. Here are the key features and functions it provides:

- **Object detection and tracking**: You can use algorithms and functions to detect and track objects in images or videos.
- **Feature detection, extraction, and matching**: The toolbox includes functions for detecting and extracting features from images, as well as matching features across multiple images.
- **Camera calibration**: Automation workflows are available for calibrating single, stereo, and fisheye cameras, ensuring accurate measurements and geometric transformations.
- **3D vision**: The toolbox provides assistance for various tasks, including stereo vision, point cloud processing, structure from motion, and visual **simultaneous localization and mapping** (**SLAM**).
- **Computer vision apps**: The toolbox offers user-friendly apps for automating ground truth labeling and camera calibration workflows.

- **Deep learning-based object detection and segmentation**: This tool allows you to detect objects using deep learning algorithms such as SSD, YOLO, and ACF. Furthermore, it enables semantic and instance segmentation using deep learning algorithms such as Mask R-CNN and U-Net.
- **Large image support**: The toolbox includes algorithms that can handle images too large to fit into memory, enabling analysis of high-resolution or gigapixel images.
- **Performance acceleration**: You can leverage the power of multicore processors and GPUs to accelerate your computer vision algorithms.
- **Code generation**: MATLAB's code generation capabilities allow you to generate C/C++ code for integrating with existing code, prototyping on desktop systems, and deploying algorithms on embedded vision systems.

With the Computer Vision Toolbox you have a comprehensive set of tools and functions to tackle various computer vision tasks, from basic image processing to advanced deep learning-based object detection and segmentation.

Text Analytics Toolbox

The Text Analytics Toolbox in MATLAB provides a range of algorithms and visualizations for preprocessing, analyzing, and shaping text data. It is designed to support various applications, including sentiment analysis, language modeling, machine translation, and topic modeling. Here are the key features of the Text Analytics Toolbox:

- **Text preprocessing**: You can use the toolbox to process raw text data extracted from diverse sources such as equipment logs, news feeds, surveys, operator reports, and social media. The toolbox provides tools for extracting text from popular file formats and performing preprocessing tasks such as cleaning, tokenization, and stopword removal.
- **Text extraction and transformation**: With the toolbox, you can extract single words or n-grams from text data and convert them into numeric representations suitable for analysis. This enables you to transform raw text into structured and meaningful features.
- **Statistical modeling**: The Text Analytics Toolbox offers ML techniques such as **latent semantic analysis** (**LSA**), **latent Dirichlet allocation** (**LDA**), and **word embedding** to uncover patterns, clusters, and topics within large text datasets. These models help in understanding the underlying structure and content of the text data.
- **Integration with other data sources**: Features generated using the Text Analytics Toolbox have the capability to be merged with features obtained from different data sources, including numerical or categorical data. This integration allows you to develop comprehensive ML models that leverage multiple data types for improved performance and insights.

The Text Analytics Toolbox provides a comprehensive set of tools for processing and analyzing text data. By leveraging ML techniques and combining text features with other data sources, you can gain valuable insights and develop powerful predictive models for various applications.

After having explored the tools available in MATLAB to build ML-based applications, we can now analyze examples of applications already available in real life that make use of this technology. In this way, we will be able to have some interesting ideas and immediately start thinking about the possible reuse of the ML to adapt it to our work environment.

ML applications in real life

ML, as a modern innovation, has revolutionized numerous industrial and professional processes, enhancing various aspects of our daily lives. Intelligent systems powered by ML algorithms possess the capability to learn from historical data or past experiences. By leveraging this knowledge, ML applications can generate outcomes and insights.

The fields of study in which ML is used cover many types of problems. The main ones are as follows:

- The representation of knowledge and reasoning that aims to reproduce the way of reasoning of the human brain through the definition of symbolism and languages to create machines capable of performing automatic reasoning
- Planning and coordination dealing with the development of systems that, given an application domain, have the objective of predicting future results and making decisions to achieve these objectives and maximize their benefits
- Robotics, for studies related to the movement of mechanical parts for the creation of a sequence of actions that perform a specific task or that can react to a specific phenomenon, the manipulation or movement of objects, or the localization or construction of maps
- Natural language processing, which concerns the automatic interpretation of natural language, both in written and spoken form
- Artificial vision: the set of processes for processing and analyzing images, pattern recognition, and automatic analysis of the scene, using 2D and 3D technologies

ML has a wide range of real-life applications; here are some examples:

- **Facial recognition**: Facial recognition is used in many applications, such as biometric authentication on smartphones, security monitoring in public places, and identifying individuals for investigative purposes
- **Personalized recommendations**: Platforms such as Netflix, Amazon, and Spotify use ML algorithms to analyze user data and suggest personalized content or products based on their tastes and preferences

- **Virtual assistance**: Virtual assistants such as Apple's Siri, Amazon's Alexa, and Google Assistant rely on ML algorithms to understand and answer users' questions, as well as to provide information and support in daily activities
- **Autonomous vehicles**: ML plays a key role in the development of autonomous vehicles. Autonomous driving systems use ML algorithms to analyze sensor data and make real-time decisions for navigation and road safety
- **Fraud detection**: Financial institutions use ML models to analyze spending patterns and customer transactions to identify suspicious behavior or fraudulent transactions and prevent fraud
- **Medical diagnostics**: ML is used to analyze large amounts of medical data, such as diagnostic images, patient records, and genetic data, to help doctors diagnose and treat diseases
- **Optimizing supply chains**: Companies use ML algorithms to optimize inventory management operations, forecast customer demand, and optimize delivery times, improving efficiency and reducing costs
- **Emotion sensing**: Emotion analysis using ML algorithms is used in several applications, such as analyzing sentiments on social media, evaluating user responses to a product or service, and personalizing user experiences

These are just a few examples of how ML is applied in real life. Its applications are continuously growing, and its impact is increasingly evident in different sectors.

Let's move on to the summary next.

Summary

In this chapter, we embarked on an exciting journey into the world of ML, exploring a range of popular algorithms to find the best fit for our specific needs. We learned the importance of conducting a preliminary analysis to determine the most suitable algorithm and gained insights into the step-by-step process of building ML models.

Furthermore, we delved into the powerful capabilities of MATLAB for ML, including its support for classification, regression, clustering, and deep learning tasks. We discovered the convenience of using MATLAB apps for automated model training and code generation, streamlining our workflow.

We also introduced the Statistics and Machine Learning Toolbox and the Deep Learning Toolbox, which provided us with additional tools and functionalities to solve our specific problems. We recognized the significance of statistics and algebra in the field of ML and understood how MATLAB could assist us in leveraging these concepts effectively.

Looking ahead to the next chapter, we will focus on seamlessly interacting with the MATLAB workspace. We will explore techniques for importing and organizing our data within MATLAB, as well as methods for exporting data from the workspace. Moreover, we will learn how to properly format and organize our data for the subsequent stages of data analysis.

By mastering these essential skills, we will be well equipped to tackle data analysis challenges and make the most of MATLAB's capabilities in the upcoming chapters.

So, in the next chapter, we will learn how to import and export data into MATLAB. We will understand how to work with different types of data and how to perform data wrangling. Additionally, we will understand exploratory statistics and exploratory visualization, data normalization, feature scaling, and data augmentation.

2

Working with Data in MATLAB

Today, the amount of data generated is enormous; computers, smart TVs, smartphones, home appliances, credit cards, sensors, public and private transport, and automation systems: these devices effortlessly produce a plethora of data, and here are just a handful of examples. Various purposes are served by storing and utilizing such data. One notable application is the utilization of **machine learning** (**ML**) algorithms for data analysis. This chapter delves into the process of importing and organizing data in MATLAB. To achieve this, it is crucial to acquaint yourself with the MATLAB workspace to streamline operations. The chapter then proceeds to examine the different data formats available for collected data and goes on to explore various data formats for the gathered data, providing guidance on importing and exporting data to and from MATLAB. Moreover, it delves into data types suitable for managing grouping variables and categorical data. The section wraps up by detailing the process of exporting data from the workspace, encompassing cell arrays, structure arrays, and tabular data, and storing them in file formats supported by MATLAB.

Finally, it offers insights on organizing data in the appropriate format for the subsequent stages of data analysis.

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

- Importing data into MATLAB
- Reading ASCII-delimited files
- Exporting data from MATLAB
- Working with different types of data
- Exploring data wrangling
- Discovering exploratory statistics
- Introducing exploratory visualization
- Understanding advanced data preprocessing techniques in MATLAB

Technical requirements

In this chapter, we will introduce ML basic concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed, and a working knowledge of the MATLAB environment is also required.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `IrisData.csv`
- `matrix.mat`
- `matrix.txt`
- `NumMatrix.txt`
- `ItalianMuseum.xlsx`
- `DataItalianCities.txt`
- `coliseum.jpg`
- `Apollo13.wav`
- `CleaningData.xlsx`
- `GlassIdentificationDataSet.xlsx`

Importing data into MATLAB

The exchange of data between the analysis environment and external devices plays a pivotal role in data analysis. Importing data refers to the process of bringing external data into a software or platform for further analysis, processing, or storage. This flow of operation can be developed in MATLAB through the following steps:

1. **Prepare your data**: Ensure that your data is in a compatible format such as a text file, spreadsheet (CSV or Excel), or a supported file format (MAT or HDF). Make sure the data is organized and structured properly.
2. **Navigate to the Import Tool**: MATLAB provides an interactive tool called the Import Tool that simplifies the data import process. To access it, you can either click on the **Import Data** button on the MATLAB toolbar or use the `importdata()` function in the MATLAB command window and press *Enter*. This function loads data from a file:

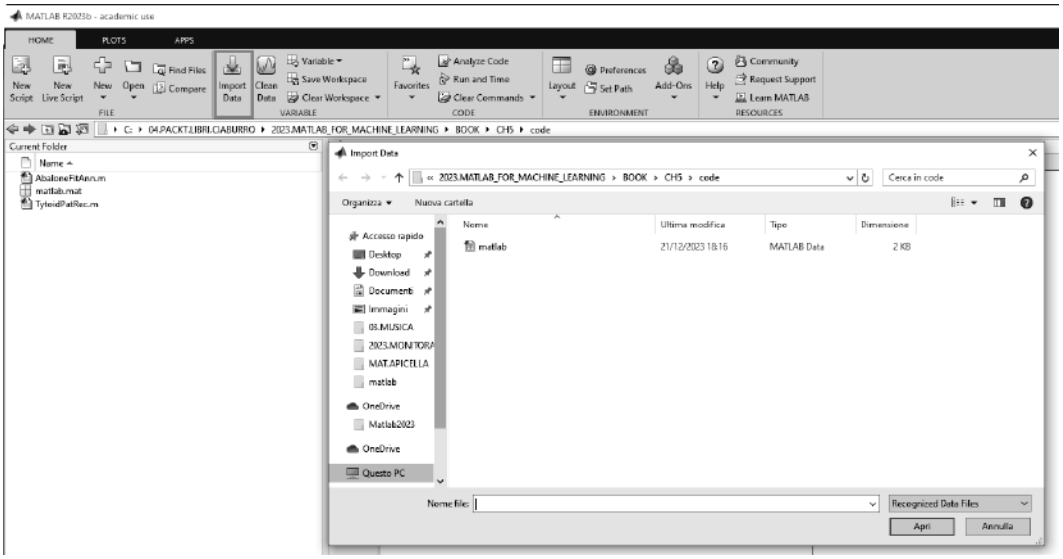


Figure 2.1 – Import Data procedure

3. **Select your data file**: In the Import Tool, browse and select the file you want to import. MATLAB supports various file formats, and you can choose the appropriate file type from the drop-down menu in the tool.
4. **Set import options**: Depending on the file format, the Import Tool will display different import options. You can specify how to import data, such as choosing the range of data to import, handling headers, specifying column formats, or indicating how missing values should be handled. Adjust these options according to your data.
5. **Preview and import the data**: Once you've configured the import options, click on the **Import Selection** button in the Import Tool (*Figure 2.2*). MATLAB will then load and preview the data. You can review the imported data in the tool to ensure it appears as expected:

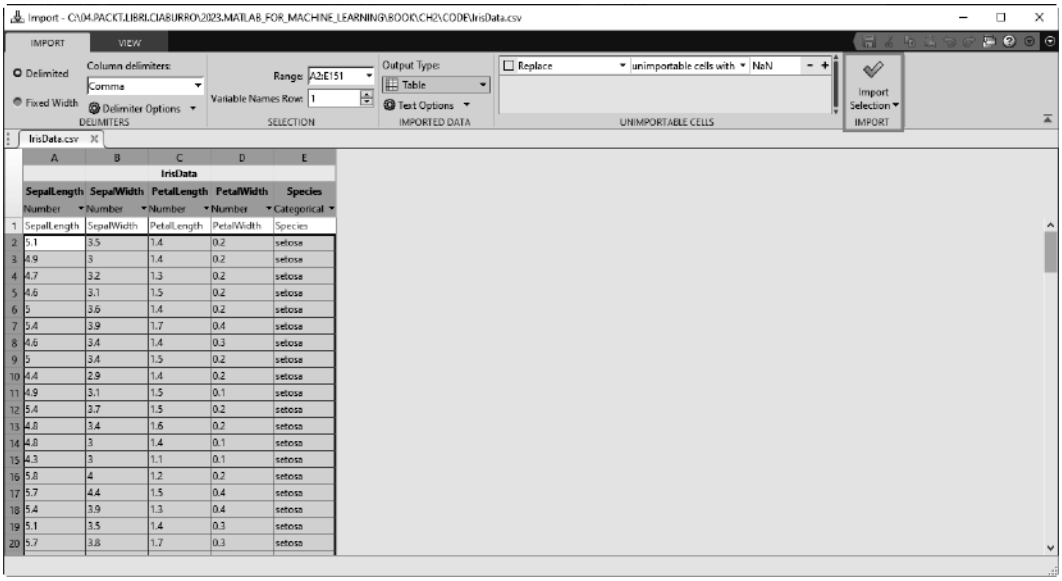


Figure 2.2 – Import window

6. **Store the imported data**: MATLAB will typically import the data into a variable of the appropriate data type. You can choose to save the imported data to a variable in the MATLAB workspace or assign it to a specific variable name for further analysis and manipulation.
7. **Perform data analysis**: With the data successfully imported into MATLAB, you can now utilize its extensive range of built-in functions and tools for data analysis, visualization, and modeling. Apply the appropriate MATLAB commands and functions to process and analyze the imported data as needed.

In addition to the Import Tool, MATLAB also provides various other functions for importing data, such as the `readtable()` function for reading tabular data, the `readmatrix()` function for importing Excel files, `load()` for loading MAT-files, and more.

Exploring the Import Tool

The Import Tool is the simplest and most user-friendly method for importing data into MATLAB. It eliminates the need for you to have prior knowledge about the data format. With the Import Tool, you can easily specify the file that contains the data, and the tool automatically processes the file contents, making the import process hassle-free. To achieve this, we can utilize the import wizard as outlined next:

1. Launch the import wizard using the **Import Data** button (*Figure 2.1*), which opens the import data dialog box.

Important note

It's also possible to use the command line to open the Import Tool, using the `uiimport` command.

2. Select the desired file; in this case, let's choose `IrisData.csv`.
3. The Import Tool window will open, as shown in *Figure 2.3*.
4. The Import Tool window displays a preview of the selected file's data. You can choose to select specific data or import the entire content.
5. In the **Import Data** section (*Figure 2.3*), use the drop-down menu below the **Output Type** label to specify how you want the data to be imported. The available options are the following:

    I. **Table**: Import selected data as a table.

    II. **Column vectors**: Import each column as an individual *m*-by-1 vector.

    III. **Numeric Matrix**: Import selected data as an *m*-by-*n* numeric array.

    IV. **String Array**: Import selected data as a string array with 1-by-*N* character vectors.

    V. **Cell Array**: Import selected data as a cell array capable of storing multiple data types, including numeric and text data.

6. In the **Import Data** window (*Figure 2.3*), in the drop-down menu below the **Output Type** label, select the **Column vectors** option.
7. Click the **Import Selection** button (*Figure 2.3*). This action creates variables in your MATLAB workspace.

To verify the successful import, check the workspace browser. If new variables appear, it indicates that the import was successful. In this case, there should be five new variables: `PetalLength`, `PetalWidth`, `SepalLength`, `SepalWidth`, and `Species`. These variables represent the five columns from the selected file.

To examine the contents of each variable, you can either call them from the MATLAB prompt or double-click on the variable name in the workspace browser. This action opens a variable window displaying the values:

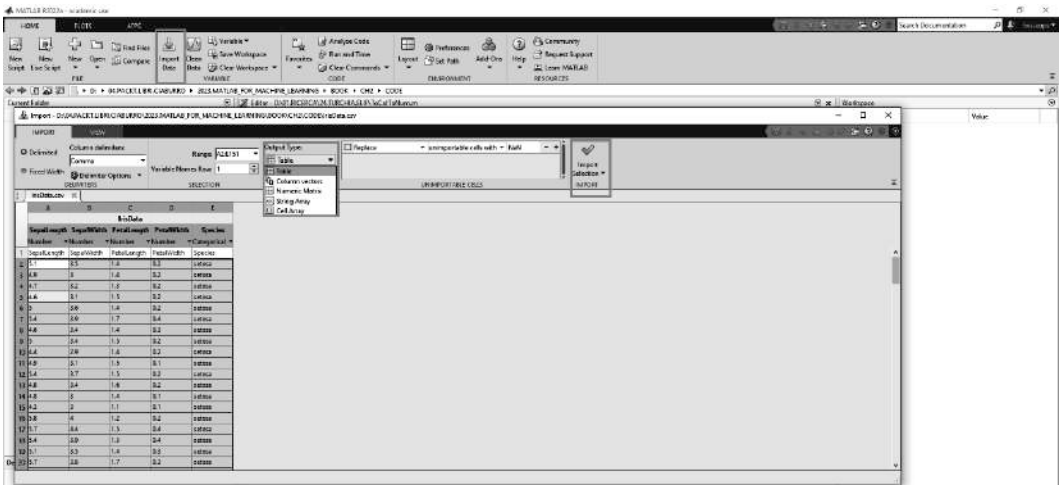


Figure 2.3 – Import Data window

The Import Tool is not the only way to import data into MATLAB, even if it is the simplest way to do this. As already mentioned, it is possible to import files into MATLAB programmatically.

Using the load() function to import files

The `load()` function is a way to import data programmatically: using this function, it is possible to import into MATLAB variables from a file already present in the workspace.

The `load()` function works on MAT-files; MAT-files are a binary file format used by MATLAB to store variables, data, and other MATLAB objects. These files have the `.mat` extension and can contain a wide range of data types, including numeric arrays, structures, cell arrays, and more. Using only the `load` command without any information, we retrieve all variables from the MAT-file named `matlab.mat`, assuming it exists. In case the file does not exist, an error will be returned.

When the `load()` function is used in MATLAB specifying a filename, it behaves differently depending on the type of file specified by the filename parameter:

- If the filename refers to a MAT-file, the `load`(*filename*) function imports the variables stored within the MAT-file and adds them to the MATLAB workspace.
- On the other hand, if the filename corresponds to an ASCII file, executing `load('filename')` creates a double-precision array that holds the data extracted from the file. This array can be accessed and manipulated within the MATLAB environment.

Let's suppose that the `.mat` file contains a single variable named `matrix.mat` that is a 3x3 matrix of integers, using the following command:

```
load('matrix.mat')
```

We can find a new variable in the MATLAB workspace named `matrix` that contains the following 3x3 matrix:

```
matrix
matrix =
    10    20    30
    40    50    60
    70    80    90
```

Using the `load` command is a very effective way to recall data from previous MATLAB sessions.

> **Important note**
>
> Remember to enclose the filename in single quotes; otherwise, MATLAB returns a message such as this: `Unable to resolve the name matrix.mat`.

In addition to loading MAT-files to save workspace contents, let's explore how to load ASCII text files. These files contain plain-text data and can be loaded using different functions based on the delimiter used to separate values within the text.

If numeric entries in the text file are separated by a blank space, comma, semicolon, or tab, the `load()` function can be utilized. Let's proceed with an example to demonstrate its usage. Let's suppose we have the 3x3 matrix already used in the previous example in an ASCII file (`matrix.txt`).

Consider the following code snippet:

```
data = load('matrix.txt')
```

In this example, the `load()` function is used to load the contents of the text file named `matrix.txt`. The function assumes that numeric entries within the file are separated by one of the supported delimiters. The loaded data is then stored in the `data` variable for further processing in MATLAB. Make sure to replace `matrix.txt` with the actual filename, and ensure that the file is located in the current MATLAB folder or provide the full path to the file if it is in a different directory.

The following results are shown:

```
data =
    10    20    30
    40    50    60
    70    80    90
```

It is important to exercise caution when using the `load` command, especially when dealing with MAT-files from untrusted sources. The reason is that the `load` command can potentially execute code contained within the MAT-file during the process of initializing variables.

Reading ASCII-delimited files

The `readmatrix()` function in MATLAB allows you to read the contents of a text file into a matrix. It is a convenient way to load numerical data from a delimited or fixed-width text file. The basic syntax for using `readmatrix` is as follows:

```
NumMatrix = readmatrix('NumMatrix.txt');
```

The function will attempt to infer the delimiter used in the file automatically. You can also specify additional options to customize the behavior of `readmatrix`, such as specifying the range of rows or columns to read, handling missing data, specifying the delimiter explicitly, and more. Here's an example:

```
NumMatrix = readmatrix('NumMatrix.txt', 'Range', 'A1:C3', 'Delimiter',
',');
```

In this example, the `Range` option is used to specify that only the data in the range `A1` to `C3` should be read, and the `Delimiter` option specifies that the data is comma-separated. The following results are returned:

```
NumMatrix =
    10    20    30
    40    50    60
    70    80    90
```

`readmatrix` returns the data from the file as a matrix, where each row corresponds to a line in the file, and each column represents a separate value.

The `readmatrix()` function in MATLAB provides automatic detection of import parameters for your file based on its file extension:

- If the file extension is `.txt`, `.dat`, or `.csv`, `readmatrix` recognizes it as a delimited text file
- If the file extension is `.xls`, `.xlsb`, `.xlsm`, `.xlsx`, `.xltm`, `.xltx`, or `.ods`, `readmatrix` identifies it as a spreadsheet file

Another way to import data into MATLAB is by using the `readtable()` function: this function is used to read the contents of a tabular data file into a MATLAB table. It is a convenient way to handle structured data in various formats such as CSV, delimited text, Excel spreadsheets, and more.

The basic syntax for using `readtable()` function is as follows:

```
TabData = readtable('IrisData.csv');
```

`readtable` automatically detects the file format based on its extension and imports the data into a table. You can also specify additional options to customize the behavior of `readtable`, such as specifying the range of rows or columns to read, handling missing data, specifying the variable names and types, and more. Take a look at this example:

```
TabData2 = readtable('IrisData.csv', 'Range', 'A1:C10', 'MissingRule',
'fill');
```

In this example, the `Range` option is used to specify that only the data in the range `A1` to `C10` should be read (only 3 columns and only 10 rows). The `MissingRule` option is set to fill to handle missing data.

`readtable` returns the data from the file as a MATLAB table, which is a versatile data structure that allows for the easy manipulation, analysis, and visualization of tabular data.

Let me highlight a last way to import data. I'm referring to the `readcell()` function: this is used to read the contents of a cell-based file, such as a CSV or delimited text file, into a cell array. It is specifically designed to handle files where each element is represented as a separate cell. In MATLAB, a cell array is a data structure that can hold elements of different types, such as numbers, strings, or other MATLAB objects. Unlike regular arrays where elements must be of the same type, a cell array allows for flexibility in storing and manipulating heterogeneous data.

The basic syntax for using `readcell` is as follows:

```
CellArray = readcell('ItalianMuseum.xlsx');
```

The `readcell()` function automatically detects the file format based on its extension and imports the data into a cell array. As for the previous function (`readtable`), we can also specify additional options to customize the behavior of `readcell`, such as specifying the range of rows or columns to read, handling missing data, specifying the delimiter explicitly, and more. `readcell` returns the data from the file as a cell array, where each cell represents an element in the file. The size and structure of the resulting cell array depend on the contents of the file.

Let's see another function very useful for reading formatted data: the `textscan()` function is capable of reading both strings and numbers from a single dataset. It operates by extracting data from an open text file and storing the contents in a cell array. To specify the text file, a file identifier (such as `fileID`) is used. In order to work with the file, it is necessary to employ the `fopen()` function, which opens the file and retrieves the file identifier. It is not possible to directly open the file and begin working with it. After reading data from the file, it is important to close the file by using the `fclose()` function and specifying a `fileID` variable. When using the `textscan()` function, it is necessary to provide a format specification that defines how the data should be interpreted. The format specification consists of conversion specifiers that represent different data types.

To specify the data format for reading the `DataItalianCities.txt` file, you need to define the format specification based on the structure of the data in the file. Let's assume that each line in the file represents a list of records; based on this assumption, you can specify the data format as follows:

```
formatSpec = '%s %d %f %d %d';
```

In this format specification, the following applies:

- `%s` corresponds to the city name (a string)
- `%d` corresponds to the population (an integer)
- `%f` corresponds to the surface area (floating-point numbers)
- `%d` corresponds to the population density (an integer)
- `%d` corresponds to the altitude (an integer)

With this format specification, the `textscan()` function will read the data from the file, considering each line to have a string, an integer, and two floating-point numbers in the specified order.

To open the `DataItalianCities.txt` file in read-only mode and assign it to the `fileID` variable, you can use the `fopen()` function as follows:

```
fileID = fopen('DataItalianCities.txt', 'r');
```

The `fopen()` function in MATLAB is used to open files with a specified mode. The `fopen()` function returns a file identifier, represented by the `fileID` variable, which can be used to perform file operations such as reading, writing, or closing a file. Remember to close the file using the `fclose()` function after you have finished working with it.

The first argument, `DataItalianCities.txt`, is the name of the file you want to open. Make sure that the file is located in the same directory as your code or provide the full path if it's located elsewhere. The second argument, `'r'`, indicates that you want to open the file in read-only mode. This allows you to read the contents of the file without modifying it. After executing this line, the `fileID` variable will contain the identifier or handle to the opened file, which you can use to perform read operations on the file.

To read the data from the `DataItalianCities.txt` open file using the `textscan()` function in MATLAB and assign the result to the `DataItalianCities` variable, you can use the following code:

```
DataItalianCities = textscan(fileID, formatSpec);
```

After executing this line of code, the `DataItalianCities` variable will store the extracted data from the file. It will be a cell array where each element corresponds to a column of data based on the specified format.

Finally, we must close the file:

```
fclose(fileID)
```

After importing the data from the file and assigning it to the `DataItalianCities` variable, you can check if everything works as intended by performing various operations on the data. After seeing several ways to import data into the MATLAB environment, we must now learn how to export data processed using MATLAB tools. In this way, it will be possible to exchange data with different platforms and use the data obtained to elaborate reports and technical notes.

Exporting data from MATLAB

Exporting data represents a crucial activity for a data scientist: it allows you to perform advanced data analysis using specialized tools and libraries commonly used in data science. By exporting data from MATLAB to formats compatible with these tools, you can leverage their extensive functionality for **exploratory data analysis** (**EDA**), statistical modeling, ML, and visualization. Exporting data from MATLAB serves several purposes and can be beneficial in various scenarios:

- **Data sharing**: Exporting data allows you to share your results, findings, or processed data with others who may not have direct access to MATLAB
- **External analysis**: Exporting data enables you to analyze your MATLAB data using external software tools or programming languages
- **Documentation and reporting**: Exported data can be used for documentation and reporting purposes
- **Data backup**: Exporting data serves as a form of data backup
- **Integration with other systems**: Exporting data allows you to integrate MATLAB with other systems or workflows

In MATLAB, you can export data to various file formats or external systems using different functions and techniques, depending on the desired output format. Here are some commonly used methods to export data from MATLAB:

- Exporting to a text file:
  - Use the `fprintf` function to write data in a formatted manner to a text file
  - Use the `writematrix` or `writetable` functions to export data to a text file
- Exporting to a spreadsheet file:
  - Use the `writematrix` function to export numerical data to an Excel file (XLSX format)
  - Use the `writetable` function to export tabular data to an Excel file or CSV file

- Exporting to a specific file format:
  - Use functions specific to the desired file format, such as `save as` for saving plots as image files (PNG, JPEG, and so on) or the `audiowrite` function for exporting audio data
- Exporting to a database or an external system:
  - Use database-specific functions or libraries, such as Database Toolbox or third-party libraries, to export data to databases such as MySQL, SQLite, and so on
- Exporting to workspace variables:
  - Use the `save` function to save specific MATLAB workspace variables to a MAT-file for later use

Let's take a look at several examples of how to use MATLAB functions to export data processed into that environment. Correspondingly, many of the functions we have used to import data into MATLAB also have corresponding functions that enable us to export data:

1. We can start using the `save` command to export data in `.mat` format:

   ```
   save MatlabSession.mat
   ```

   The `save` command allows you to save all the contents of the workspace in a compressed file with a `.mat` extension, commonly referred to as a MAT-file. This MAT-file preserves the variables, data, and any other workspace information, enabling you to easily load and retrieve the saved workspace state at a later time.

2. Now, we will create a sample matrix to explain the usage of the `writematrix` function: let me start creating a 5x5 matrix using the `rand()` function. The `rand` function in MATLAB generates uniformly distributed random numbers between `0` and `1`. It is commonly used for generating random numbers or creating random matrices and arrays of a specified size. For example, we can write the following command:

   ```
   MyMatrix1 = rand(5)
   ```

   In this case, `MyMatrix1` will be a 5-by-5 matrix of random numbers between 0 and 1. The following matrix has been generated:

   ```
   MyMatrix1 =
       0.8147    0.0975    0.1576    0.1419    0.6557
       0.9058    0.2785    0.9706    0.4218    0.0357
       0.1270    0.5469    0.9572    0.9157    0.8491
       0.9134    0.9575    0.4854    0.7922    0.9340
       0.6324    0.9649    0.8003    0.9595    0.6787
   ```

Additionally, you can use other functions such as `randi` to generate random integers within a specific range or `randn` to generate random numbers from a standard normal distribution. Remember to set the random number generator's seed using `rng` to ensure reproducibility if you need to generate the same set of random numbers in different runs of the program.

3. Now, we can export the matrix generated simply by using the `writematrix()` function as follows:

```
writematrix(MyMatrix1);
```

4. We can add a specific delimiter to the file using a parameter in the command, as follows:

```
writematrix(MyMatrix1,' MyMatrix1Del.txt', 'Delimiter','tab'));
```

5. Now, we can use the same matrix to write the data to a spreadsheet file. To export the matrix in a spreadsheet file, we can use the `writematrix(9)` function as follows:

```
writematrix(MyMatrix1,' MyMatrix2.xlsx')
```

We can specify the sheet and the range also. An interesting task is to append data to a spreadsheet. To do that, we can create a new matrix:

```
MyMatrix2 = rand(5)
```

6. Then, we append the data to the `MyMatrix2.xlsx` file as follows:

```
writematrix(MyMatrix 2,' MyMatrix2.xlsx ','WriteMode','append')
```

In this way, the second matrix was appended to the first in the same file; this is a good solution to add some information to a single file. Another function for writing data is `writetable()`. This function is used to write data from a table or dataset array to a file. Here's an example of how you can use `writetable()` to write a table named `MyTable` to a CSV file named `MyTable.csv`:

1. To start, we have to generate a table:

```
MyTable = table(['A';'B';'C'],[12524 10253;9952 14251;8521
12547],{'Rome';'Naples';'Milan'},[false;true;true])
```

The following table is generated:

```
Mytable =
  3×4 table
    Var1          Var2             Var3         Var4
    ____    _______________    __________    _____
     A      12524    10253    {'Rome'  }    false
     B       9952    14251    {'Naples'}    true
     C       8521    12547    {'Milan' }    true
```

2. Now, we can export the table simply by using the following command:

```
writetable(MyTable, ' MyTable.csv');
```

After having seen how to import and export files in the MATLAB environment, we will now see how to manipulate specific types of data.

Working with different types of data

Working with different types of data involves understanding their specific formats and applying appropriate techniques for manipulation and analysis. Here are some common types of data and general considerations for working with them.

Working with images

In MATLAB, working with images involves loading, displaying, and performing various operations on image data. Here's a brief overview of how to work with images in MATLAB:

- **Loading an image**: We can load an image into MATLAB using the `imread()` function. It reads the image file and returns a numeric array representing the image data. Here's an example:

```
img = imread('coliseum.jpg');
```

- **Displaying an image**: MATLAB provides the `imshow()` function to display an image. It opens a separate window and shows the image. Here's an example:

```
imshow(img);
```

- **Manipulating an image**: MATLAB offers a wide range of functions to perform various operations on images. For example, you can resize an image using `imresize()`, convert it to grayscale using `rgb2gray()`, apply filters using functions such as `imfilter()`, and perform various other operations.
- **Saving an image**: After modifying an image, you can save it using the `imwrite()` function. It takes the modified image array and writes it to a file. Here's an example:

```
imwrite(img,'new_coliseum.jpg');
```

  To export the data from a variable to a user-specified file in MATLAB, we used the `imwrite()` function. This function writes the data from the variable into a file, with the format of the exported file determined by the specified extension. A new file will be created in the current folder by default. The bit depth of the output image depends on the data type of the variable and the chosen file format.

These are just some basic operations for working with images in MATLAB. There are many advanced techniques and functions available for image processing and analysis, such as edge detection, image segmentation, and object recognition. MATLAB's Image Processing Toolbox provides a comprehensive set of tools for these tasks.

By leveraging MATLAB's capabilities, you can effectively manipulate and analyze image data for various applications, including **computer vision** (**CV**), image recognition, and ML.

Audio data handling

Sound signals are commonly stored digitally in various file formats, with the differences usually stemming from the level of compression applied and the resulting impact on sound quality. MATLAB offers several functions for handling audio files, facilitating tasks such as writing data to an audio file, retrieving information about the file, reading data from an audio file, recording data from an audio input device, and playing audio files.

To import audio files in MATLAB, you can utilize the `audioread()` function. This function supports various file formats, including WAV, OGG, FLAC, AU, MP3, and MPEG-4 AAC. As an example, let's import a short audio file containing the original NASA recording of Apollo 13, featuring the famous message, "*Houston, we have a problem.*"

Here's an example of how you can use `audioread()` to import the audio file:

```
[audio, sampleRate] = audioread('apollo13.wav');
```

In this code, the following applies:

- `apollo13.wav` represents the filename of the audio file you want to import. Make sure the file is located in the current working directory or provide the full file path.
- `audio` is a variable that will store the audio data from the file.
- `sampleRate` is the sample rate of the audio file, which indicates the number of samples per second.

By executing this line of code, MATLAB will import the audio data from the specified file, storing it in the `audio` variable. The sample rate of the audio file will be stored in the `sampleRate` variable. You can then further process or analyze the audio data using MATLAB's signal-processing capabilities. In MATLAB, an alternative method to read audio files is through an interactive approach. You can accomplish this by selecting the **Import Data** button from the **Home**tab or by double-clicking the filename in the **Current Folder** browser. This allows you to read and import audio files directly into your MATLAB environment.

By leveraging the interactive functionality in MATLAB, you can seamlessly read and listen to audio files, enabling a convenient workflow for working with sound signals and performing subsequent analysis or processing tasks, as in this example:

```
sound(audio,sampleRate)
```

To have more control over audio playback in MATLAB, you can utilize the `audioplayer()` function. This function allows you to pause, resume, and define callbacks for audio playback. By creating an `audioplayer` object, you can call its methods to play the audio and access additional playback functionalities.

Here's an example of how you can use the `audioplayer()` function:

```
Apollo13Obj = audioplayer(audio,sampleRate);
play(Apollo13Obj);
```

Now that we have an audio file in the MATLAB workspace, as a matrix (audio), we can export it through the use of the `audiowrite()` function:

```
audiowrite('apollo13.wav',audio,sampleRate)
```

The following command saves the data to a WAV file named `apollo13.wav` in the current folder. It is important to note that this function, `audiowrite()`, is versatile and can also write data to other audio file formats such as OGG, FLAC, and MPEG-4 AAC. Lastly, we utilize the `audioinfo()` function to retrieve information about the WAV file, `apollo13.wav`, that was just created:

```
InfoAudio = audioinfo('apollo13.wav')
```

The `audioinfo()` function is then used to obtain details about the newly created WAV file, such as the audio format, duration, number of channels, and sample rate.

Exploring data wrangling

Data wrangling, also known as data munging or data preprocessing, refers to the process of cleaning, transforming, and preparing raw data for analysis. It involves several tasks, such as handling missing or inconsistent data, removing duplicates, reshaping data formats, and merging multiple datasets. Common techniques used in data wrangling include the following:

- **Data cleaning**: Identifying and handling missing values, outliers, and errors in the dataset. This may involve imputing missing values, removing outliers, or correcting errors.
- **Data transformation**: Modifying the structure or format of the data to make it compatible with the desired analysis or modeling techniques. This can include tasks such as changing variable types, scaling numerical values, or encoding categorical variables.

- **Data integration**: Combining multiple datasets or data sources into a single unified dataset. This may involve joining datasets based on common variables or merging datasets with different structures.
- **Data reshaping**: Transforming the layout or structure of the data to meet specific requirements. This can include tasks such as pivoting, melting, or transposing the data.
- **Data aggregation**: Summarizing or grouping data to obtain higher-level insights. Aggregation may involve calculating summary statistics, creating aggregate variables, or grouping data based on specific criteria.
- **Data reduction**: Reducing the size or dimensionality of the dataset without significant loss of information. This can include techniques such as feature selection or dimensionality reduction.

The first operation on the data refers to identifying, correcting, and removing inconsistencies, errors, and redundancies within a dataset, transforming it into a valuable resource for effective decision-making. Let's take a closer look at it.

Introducing data cleaning

Data cleaning in MATLAB involves performing various operations to handle missing values, outliers, and inconsistencies in the dataset. Here are some common techniques for data cleaning in MATLAB:

- **Handling missing values**: Such techniques allow us, for example, to identify missing values in the dataset using functions such as `isnan()` or `ismissing()`. Replace missing values with appropriate values using techniques such as mean imputation (using `mean()`), median imputation (using `median()`), or interpolation (using `interp1()` or `interp2()`). Remove rows or columns with missing values using functions such as `rmmissing()`.
- **Removing outliers**: Identify outliers using statistical techniques such as the z-score or **interquartile range** (**IQR**) methods. Remove outliers by filtering the data based on predefined thresholds or using functions such as `isoutlier()` and `rmoutliers()`.
- **Handling inconsistent or incorrect data**: Identify and correct inconsistent data values using logical operations, regular expressions, or conditional statements. Standardize data formats, such as converting dates to a consistent format using functions such as `datetime()`.
- **Data type conversion and validation**: Convert data types to the appropriate format using functions such as `str2double()`, `cell2mat()`, or `datetime()`. Validate and ensure the correctness of data values using range checks, logical checks, or custom validation functions.
- **Data validation and error checking**: Validate data against predefined rules or constraints to identify any potential errors or inconsistencies. Perform data integrity checks to ensure consistency and correctness of relationships between variables.
- **Handling duplicates**: Identify and remove duplicate records or observations using functions such as `unique()` or `duplicated()`.

Now, let's proceed with a practical illustration. We will utilize a custom-designed file that holds data for a small sample of observations, specifically the results of a test. We will acquire a file called `CleaningData.xlsx`, which is a spreadsheet encompassing some of the problems we previously mentioned:

```
SampleData = readtable('CleaningData.xlsx');
```

In this file, there are many problems: empty cells, cells containing a period (`.`), cells containing `NA` strings, cells containing `NaN` strings, and cells containing negative numbers (`-19`). The `NaN` code (**Not a Number**) refers to invalid numeric values, such as a numeric value divided by `0`.

For a faster approach to identifying observations with missing values, we can utilize the `ismissing()` function. This function enables us to retrieve a subset of observations that contain at least one missing value. The `ismissing()` function recognizes, by default, only the following missing value indicators; all the others must be specified in the following way:

```
id = {'NA' '' '-19' -19 NaN '.'};
WrongPos = ismissing(SampleData,id);
SampleData(any(WrongPos,2),:)
```

The first line of code specifies the types of data we want to locate in the table, the second identifies the location of missing data, and finally, we print only records with missing data.

In the following step, we will replace the indicators with missing values. We will proceed to clean the data by replacing the missing values indicated by `-19` with the standard missing value matrix in MATLAB, which is `NaN`. To accomplish this, we can employ the `standardizeMissing()` function, which replaces specified values within parentheses with standard missing values in an array or table:

```
SampleData = standardizeMissing(SampleData,-19);
```

Now, let's proceed to create a new table called `SampleDataNew` where we will replace missing values with appropriate values using the `fillmissing()` function. This function allows us to fill in missing entries of an array or table with a specified value and offers several methods to handle missing values. In this scenario, we will fill the missing values with values from the preceding rows of the table:

```
SampleDataNew = fillmissing(SampleData,'previous');
```

After that, there are no missing values; in fact, the missing data has been replaced with those in the previous cell.

To eliminate rows with missing values, we will generate a new table called `SampleDataMinor` that exclusively includes rows without any missing values. We can achieve this by utilizing the `rmmissing()` function, which removes missing entries from an array or table. By applying the

`rmmissing()` function to the table containing the raw data, we can obtain the desired result of a new table, `SampleDataMinor`, which excludes any rows with missing values:

```
SampleDataMinor = rmmissing(SampleData);
```

Thus, the 12-row table has only 7 rows, which are the only rows that have all the correct data.

Another issue that a data scientist has to address regards outliers, which are values that deviate significantly from the rest of the data. They can cause issues in data analysis, particularly in descriptive statistics and correlations, as they have the potential to distort the results. Identifying outliers is an important step in data cleaning, and they can also be addressed in subsequent stages of data analysis. Outliers can be classified as univariate if they exhibit extreme values in a single variable or multivariate if they demonstrate an unusual combination of values across multiple variables. In MATLAB, the process of identifying outliers is straightforward and effective. You can utilize the `isoutlier()` function, which is specifically designed to detect outliers in data. We can apply this function to the previous example, specifically to the `SampleDataNew` table that contains the cleaned data:

```
SampleDataOutlier = isoutlier(SampleDataNew(2:end,3:5));
```

The `isoutlier()` function returns a logical array where the elements are `true` if an outlier is detected in the corresponding item within the table. By default, MATLAB identifies an outlier as a value that is more than three **median absolute deviations** (**MADs**) away from the median. In our case, we applied the function specifically to the three columns that contain numeric data, while omitting the first row that contains the variable names. That function operates on each column individually. Upon comparing the `SampleDataOutlier` table with the `SampleDataNew` table, it becomes evident that we have identified an outlier only in the `age` variable, specifically the value of `100`.

Data wrangling represents a very useful data preprocessing technique, but these methods need to be accompanied by adequate exploratory analysis to identify trends and choose appropriate techniques for subsequent data analysis.

Discovering exploratory statistics

Exploratory statistics refers to the initial phase of data analysis where various statistical techniques are employed to understand the main characteristics of a dataset. There are many techniques available, but the most used one is the following.

EDA

EDA is an approach to analyzing data that focuses on understanding the main characteristics, patterns, and relationships within a dataset. It involves using statistical techniques and visualizations to summarize and explore the data to gain insights and formulate hypotheses. Here are some key steps and techniques involved in EDA:

- **Data summary**: Start by examining the basic summary statistics of the dataset, such as the mean, median, standard deviation, minimum, maximum, and so on. This gives an initial understanding of the central tendency, spread, and distribution of the data.
- **Data distribution**: Examine the distribution of individual variables to identify any skewness or non-normality. Transformations such as logarithmic, exponential, or power transformations can be applied to normalize the data if needed.
- **Correlation analysis**: Calculate correlation coefficients between pairs of variables to understand the strength and direction of their relationships. Scatter plots or correlation matrices can help visualize these relationships.
- **Outlier detection**: Identify outliers, which are data points that deviate significantly from the rest of the data. Outliers can be detected using statistical methods such as z-scores, box plots, or Mahalanobis distance.
- **Multivariate analysis**: Analyze the relationships and interactions between multiple variables simultaneously. Techniques such as **principal component analysis** (**PCA**) or cluster analysis can help uncover underlying structures in the data.
- **Hypothesis testing**: Formulate and test hypotheses based on the observed patterns and relationships in the data. Statistical tests, such as t-tests or chi-square tests, can be used to assess the significance of these findings.

After introducing EDA concepts, let's see some practical cases.

EDA in practice

Let's look at some useful EDA examples. The initial phase of our analysis involves estimating a measure of central tendency for the distribution, which entails determining a representative or central value that effectively describes the data. We will begin by importing the file named `GlassIdentificationDataSet.xlsx` into MATLAB:

```
GlassData = readtable('GlassIdentificationDataSet.xlsx');
```

This is a dataset with 214 records and 11 variables (`id`, `refractive index`, `Na`, `Mg`, `Al`, `Si`, `K`, `Ca`, `Ba`, `Fe`, `type of glass`). The unit of measure of oxide content is the weight percentage in the corresponding oxide.

To initiate our exploratory analysis, we can compute the maximum, mean, and minimum values of the imported table. MATLAB calculates these statistics independently for each column within the table. To accomplish this, we can employ three useful functions – `max()`, `mean()`, and `min()`:

```
Max = max(GlassData{:,3:8});
Mean = mean(GlassData{:,3:8});
Min = min(GlassData{:,3:8});
```

It may be useful to identify records in which `minimum` and `maximum` are found; to get this information, just specify a second output parameter to return the row index, as in the following example:

```
[Max,IndRowMax] = max(GlassData{:,3:8});
```

Additionally, using MATLAB built-in functions, we can calculate the median and mode:

```
Median = median(GlassData{:,3:8});
Mode = mode(GlassData{:,3:8});
```

Let's now take a look at quantiles and percentiles. Quantiles are values that divide the total frequency of a sample or population into a specified number of equal proportions. One specific type of quantile is quartiles, which divide a dataset into four equal groups, with each group containing a quarter of the data. When the data is arranged in ascending order, the first quartile marks the boundary below which one-quarter of the values lie. The second quartile, also known as the median, is located at the center of the distribution, with half of the data above and half below it. The third quartile represents the boundary between the top quarter and the remaining data. The concept of percentiles is much more widespread. A percentile is the percentage of cases occurring at or below a value. The relationship between quartiles and percentiles is represented as follows:

- 1st quartile = 25th percentile
- 2nd quartile = 50th percentile = median
- 3rd quartile = 75th percentile
- 4th quartile = 100th percentile

Let's calculate the first three quartiles of our sample distribution. For space reasons, we will limit the calculation to some columns. We will use the `quantile()` and `prctile()` functions in the following way:

```
Quantile = quantile(GlassData{:,3:8}, [0.25 0.50 0.75]);
Percentiles = prctile(GlassData{:,3:8}, [0.25 0.50 0.75]);
```

Let's move on to how to measure the dispersion of the data. Central tendency measures alone are not enough to fully characterize a distribution. It is essential to complement this information with measures that account for the degree of data dispersion or variability. The `range()` function provides a simple way to calculate the spread or the difference between the maximum and minimum values within a sample. It offers an effortless means to estimate the range of a sample:

```
Range = range(GlassData{:,3:8});
```

Let us now calculate the IQR with the `iqr()` function: The IQR is defined as the difference between the **third quartile** (**Q3**) and the **first quartile** (**Q1**). It represents the range of values that encompass the middle half of the observed data. The IQR serves as a dispersion index, providing a measure of how much the values deviate from a central value within the dataset. It offers valuable insights into the spread or variability of the data while being less sensitive to outliers compared to other dispersion measures such as the range or standard deviation:

```
Iqr = iqr(GlassData{:,3:8});
```

The variance is a widely recognized measure of variability that quantifies the extent to which a set of numbers deviates from its mean. It is determined by calculating the average of the squared deviations of individual values from their arithmetic mean. In MATLAB, you can compute the variance for each column using the `var()` function:

```
Variance = var(GlassData{:,3:8});
```

The standard deviation is the square root of the variance and possesses the advantageous characteristic of being expressed in the same units as the data. To compute the standard deviation for each column, you can use the following code in MATLAB:

```
StDev = std(GlassData{:,3:8});
```

Both the variance and standard deviation, as previously discussed, are susceptible to the influence of outliers. A single outlier can significantly impact the values of these statistics. Even the MAD, although less sensitive to outliers compared to the standard deviation and variance, can still be affected by erroneous data. In MATLAB, you can compute the mean or median absolute deviation using the `mad()` function. This function allows you to calculate the absolute deviation from the mean or median for each data point and then obtain the average or median of these deviations:

```
Mad = mad(GlassData{:,3:8});
```

There are multiple statistical measures available to assess the relationship between two sets of data over time. Two frequently used measures are correlation and covariance. Covariance offers a non-normalized measure of how the two sets of data tend to move together. It is estimated by summing the product of deviations from the mean for each variable in each period. In MATLAB, you can compute the covariance using the `cov()` function.

Let's consider an example where we create a 4x4 matrix of random data:

```
DataA = rand(4);
CovarianceMatrix = cov(DataA);
```

In this example, we generate a 4x4 matrix called `DataA` with random values. We then calculate the covariance matrix using the `cov()` function and store the result in the `CovarianceMatrix` variable.

The covariance matrix represents the pairwise calculation of covariance between each combination of columns in a data matrix. The sign of the covariance indicates the type of relationship between the two variables. A positive sign suggests that they move in the same direction, while a negative sign suggests they move in opposite directions. Moreover, the magnitude of the covariance reflects the strength of the relationship between the variables. However, interpreting the intensity of the relationship solely based on covariance can be challenging, as it is not a standardized measure.

To obtain a standardized measure of the relationship between two variables, correlation is commonly used. Correlation can be calculated from covariance. In MATLAB, you can compute correlation coefficients using the `corrcoef()` function. This function generates a matrix of sample correlation coefficients for a data matrix, where each column represents a separate quantity:

```
CorrelationMatrix = corrcoef (DataA);
```

It's important to note that the correlation coefficient of two random variables is a measure of their linear dependence. Correlation coefficients range from -1 to 1, and their interpretations are as follows: values close to 1 indicate a strong positive linear relationship between the data columns. Values close to -1 indicate a strong negative linear relationship (anticorrelation) between one column of data and another. Values close to or equal to 0 suggest there is little to no linear relationship between the data columns.

However, the EDA must be accompanied by an adequate exploratory visualization analysis, as visualization of the data makes it possible to understand any trends.

Introducing exploratory visualization

Exploratory visualization is a crucial step in the data analysis process, allowing us to gain insights and understand underlying patterns, relationships, and trends within our data. It involves creating visual representations of the data to explore its various attributes and uncover potential patterns or anomalies. The primary goal of exploratory visualization is to visually inspect the data, identify any interesting features, and generate hypotheses for further investigation. By leveraging the power of visual perception, we can better understand complex datasets and make informed decisions. MATLAB provides a variety of functions and tools for exploratory data visualization. Here are some commonly used functions for exploratory visualization:

- `plot()`: This function is used to create line plots, scatter plots, or any custom plot by specifying *x* and *y* coordinates.

- `histogram()`: This function creates histograms to visualize the distribution of a continuous variable. You can customize the number of bins, appearance, and other properties of the histogram.
- `bar()`: This function creates bar charts to compare categorical or grouped data. It allows customization of bar colors, widths, and labels.
- `scatter()`: This function creates scatter plots to visualize the relationship between two continuous variables. You can customize the marker style, color, and size based on additional variables.
- `boxplot()`: This function creates box plots to display the distribution of a continuous variable across different categories or groups. It shows the median, quartiles, and any outliers.
- `heatmap()`: This function creates heatmaps to visualize a matrix of data as a grid of colored cells. Heatmaps are useful for exploring patterns or correlations in multivariate datasets.
- `pairplot()`: This function creates a matrix of scatter plots for multiple variables, showing pairwise relationships. It helps visualize the correlations between variables.
- `line()`: This function creates line plots to visualize the trend or progression of a variable over time or any continuous scale. It is useful for time-series analysis or plotting continuous data.
- `imagesc()`: This function creates a color-coded image representation of a matrix or 2D data. It is useful for visualizing matrices, such as heatmaps or images.
- `contour()`: This function creates contour plots to display the contours of a 3D surface or the boundaries of regions in a 2D space. Contour plots are useful for visualizing continuous functions or spatial data.

To better understand the topic, it is also possible to show some graphs for univariate and bivariate analysis, such as `pairplot()` to show the correlation effects between characteristics and `boxplot()` to show the level of dispersion in the data.

EDA and exploratory visualization analysis are essential tools for data analysis. In the next section, we will see a practical case of the application of these exploratory analysis methodologies.

Understanding advanced data preprocessing techniques in MATLAB

After introducing data preprocessing techniques, in this section, we will analyze some data preprocessing techniques in MATLAB with practical examples. We will talk about min-max scaling and z-score standardization, which are two common techniques used to normalize data in ML. Both techniques aim to rescale numerical data to a common scale, making it easier for ML algorithms to learn from the data.

Data normalization for feature scaling

Data normalization is a preprocessing step used to scale and standardize data to a common range or distribution. It aims to bring different features or variables to a comparable scale, ensuring that no single feature dominates the analysis due to its larger magnitude. Normalizing data can also help improve the performance of certain ML algorithms. There are various techniques for data normalization, including the following:

- **Min-Max scaling**: This method scales the data to a specific range, typically between 0 and 1. The formula for min-max scaling is this:

  $$XNorm = (X - X_min) / (X_max - X_min)$$

  Here, *X* is the original data, *X_min* is the minimum value of the data, and *X_max* is the maximum value of the data. In MATLAB, we can use the `rescale()` function as follows:

```
X=1:8
X =
     1     2     3     4     5     6     7     8
Xnorm=rescale(X)
Xnorm =
         0    0.1429    0.2857    0.4286    0.5714
    0.7143    0.8571    1.0000
```

  The `rescale()` function rescales the values in array `X` to the range `[0, 1]`. The resulting array, `Xnorm`, has the same size as `X`.

- **Z-score standardization**: This technique transforms the data to have a mean of `0` and a standard deviation of `1`. The formula for z-score standardization is this:

  $$Xnorm = (X - mean) / standard_deviation$$

  Here, *X* is the original data, *mean* is the mean of the data, and *standard_deviation* is the standard deviation of the data.

  We can use the same function specifying the range as follows:

```
Xnorm=rescale(X,-1,1)
Xnorm =
   -1.0000   -0.7143   -0.4286   -0.1429    0.1429
    0.4286    0.7143    1.0000
```

The choice of normalization technique depends on the nature of the data and the requirements of the analysis or ML task. It's important to consider the characteristics of the data and select an appropriate normalization method that preserves the meaningfulness of the data.

Introducing correlation analysis in MATLAB

Correlation analysis is a statistical technique used to quantify the strength and direction of the relationship between two or more variables. It helps us understand how changes in one variable are associated with changes in another variable. This analysis allows us to quantify the strength of the relationship between two quantitative variables that are often influenced by other factors and not directly linked in a cause-and-effect manner, but still tend to vary together.

Correlations play a valuable role in describing straightforward associations between data. For instance, consider the wind turbine dataset, where you want to explore whether there is a correlation between the altitude of the turbines and the average wind speed. Each turbine has two corresponding values: altitude and wind speed. Analyzing these variables with a sample correlation reveals a linear relationship, indicating that as altitude rises, wind speed also increases. Consequently, in this scenario, a positive correlation is observed.

The correlation is quantified by a dimensionless value known as the correlation coefficient, denoted by *r*, which ranges from -1 to +1. Its statistical significance is indicated by a p-value. Therefore, correlations are typically represented using two essential numbers: *r* and *p*. Let me explain the major features of correlation and p-values:

- When the value of *r* approaches zero, the linear correlation becomes weaker.
- A positive r-value signifies a positive correlation, indicating that the two variables tend to increase in parallel.
- A negative r-value signifies a negative correlation, implying that one variable tends to increase as the other decreases.
- The p-value provides evidence that, based on the observed sample, we can confidently conclude that the correlation coefficient of the population differs from zero.
- The absence of a unit means that the correlation is measured on its own scale. In our example, the r-value does not correspond to the altitude or temperature scales. Consequently, it differs from other summary statistics such as the mean of altitudes, which falls on the same scale as the corresponding variable.

Once a significant correlation is identified, its strength can be measured. A perfect positive correlation has a value of 1, while a perfect negative correlation is -1. However, in the real world, it is uncommon to obtain a perfect correlation unless one variable is a direct proxy for the other. Conversely, if a perfectly correlated value is obtained, it may indicate an error in the data. For instance, if the recorded data represented the height above sea level instead of temperature, it would exhibit a perfect relationship with elevation.

Another crucial piece of information is the sample size, denoted as N, which represents the number of observations. In most statistical tests, understanding the sample size helps assess its validity and how well it represents the larger population. For example, if altitude and temperature were measured for only 5 campsites, but the relevant area included 2,000, it would be appropriate to increase the sample size for a more representative analysis.

In MATLAB, you can perform correlation analysis using the built-in functions available in Statistics and Machine Learning Toolbox. Here are the steps to conduct correlation analysis in MATLAB:

1. **Load or prepare your data**: Start by loading your data into MATLAB or preparing it in the desired format. Ensure that your data is in a tabular format or as separate vectors/matrices representing variables of interest. For this example, we will use the `IrisData.csv` file. The `Iris` dataset is a widely used dataset in ML and statistics. It is commonly used for classification tasks and exploration of data analysis techniques. The dataset contains measurements of four features (sepal length, sepal width, petal length, and petal width) from three different species of Iris flowers (Setosa, Versicolor, and Virginica). Each sample in the dataset consists of four feature measurements and the corresponding species label. The goal is to predict the species of an Iris flower based on its feature measurements. The Iris dataset is often used to demonstrate various data analysis and ML algorithms, such as clustering, classification, and dimensionality reduction. It serves as a popular benchmark for evaluating the performance of different models and techniques. Overall, the Iris dataset provides a convenient and well-known example for exploring and practicing data analysis and ML concepts. To load the dataset in MATLAB, we will use the `readmatrix()` function as follows:

```
Data = readtable('IrisData.csv');
```

To get some statistics on data imported, we can apply the `summary()` function as follows:

```
summary(Data)
```

The following results are returned:

```
Variables:
    SepalLength: 150×1 double
       Values:
            Min         4.3
            Median      5.8
            Max         7.9
    SepalWidth: 150×1 double
        Values:
            Min           2
            Median        3
            Max         4.4
    PetalLength: 150×1 double
        Values:
            Min           1
```

```
            Median      4.35
            Max          6.9
    PetalWidth: 150×1 double
        Values:
            Min         0.1
            Median      1.3
            Max         2.5
    Species: 150×1 cell array of character vectors
```

Now, we can verify if there are some correlations between the four numeric variables (`SepalLength`, `SepalWidth`, `PetalLength`, and `PetalWidth`).

2. **Calculate correlation coefficients**: MATLAB provides the `corrcoef()` function to calculate correlation coefficients between variables. This function returns a correlation matrix that shows pairwise correlations between variables, as in this example:

```
CorrelationMatrix = corrcoef(Data{:,1:4});
```

The following matrix is returned:

```
CorrelationMatrix =
    1.0000   -0.1176    0.8718    0.8179
   -0.1176    1.0000   -0.4284   -0.3661
    0.8718   -0.4284    1.0000    0.9629
    0.8179   -0.3661    0.9629    1.0000
```

From the analysis of the correlation matrix, we can see that there are several numbers close to `1`. Such numbers indicate a strong correlation. Reading the first column of the correlation matrix, we notice that the first variable (sepal length) is weakly correlated with the second variable (sepal width) while it is strongly correlated with the third variable (petal length) and with the fourth (petal width).

> **Important note**
>
> I leave the analysis of the other columns to you as an exercise. I just want to point out that the main diagonal contains all ones because they represent the correlation of each variable with itself.

3. **Visualize the correlation matrix**: You can visualize the correlation matrix using the `imagesc()` function in MATLAB. This function creates a color-coded image representation of the correlation matrix, making it easier to identify the strength and direction of correlations. Here's an example:

```
imagesc(CorrelationMatrix)
```

To add a color bar to the plot, we can use the following command:

```
colorbar;
```

The result is shown in *Figure 2.4*.

The shadow of blue color indicates a low correlation, while the shadow of yellow indicates a strong correlation.

4. **Calculate p-values**: If you want to determine the statistical significance of the correlations, you can use the `corrcoef()` function with an additional output argument to obtain p-values. Here's an example:

```
[CorrelationMatrix, PValues] = corrcoef(Data{:,1:4});
```

Now, we can analyze the `PValues` matrix:

```
1     0.1518 1.038e-47 2.325e-37
0.1518 1 4.513e-08 4.073e-06
1.038e-47 4.513e-08 1 4.675e-86
2.325e-37 4.073e-06 4.675e-86 1
```

You can filter the correlation matrix based on p-values to focus on statistically significant correlations:

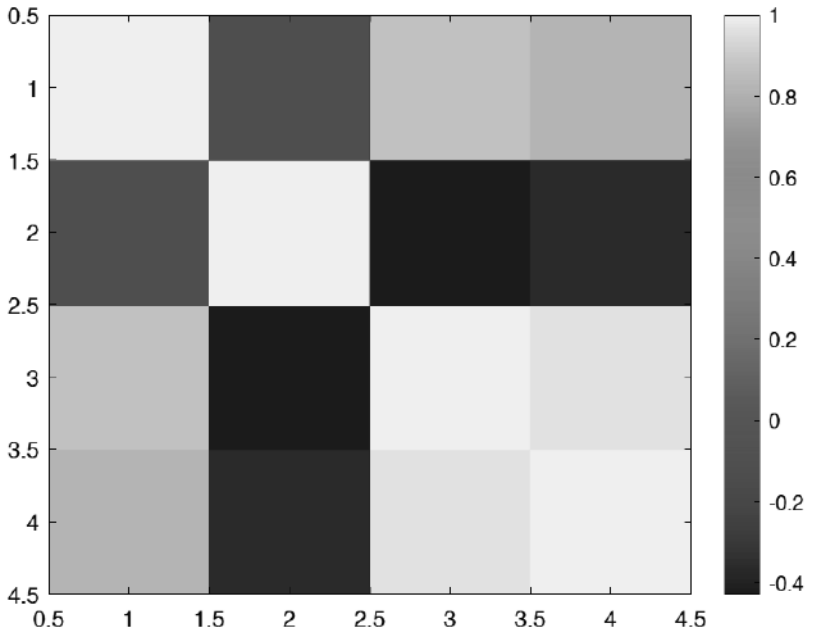


Figure 2.4 – Visualization of a correlation matrix

For example, you can set a threshold (*P-value < 0.05*) and consider correlations below the threshold as significant.

5. **Visualize scatter plots**: If you want to explore the relationship between specific variables in more detail, you can create scatter plots using the `scatter()` function in MATLAB. This can help you visualize the nature and strength of the correlations between two variables. For example, we can analyze the relationship between sepal length and sepal width, which shows a weak correlation:

```
scatter(Data.(1),Data.(2))
```

Then, we can show the correlation between sepal length and petal length, which shows a strong correlation:

```
scatter(Data.(3),Data.(4))
```

Figure 2.5 shows a scatter plot comparison of the features and shows a positive correlation. In fact, we can see that as the abscissa increases, the ordinate also increases. In *Figure 2.4*, we cannot see any appreciable correlation:

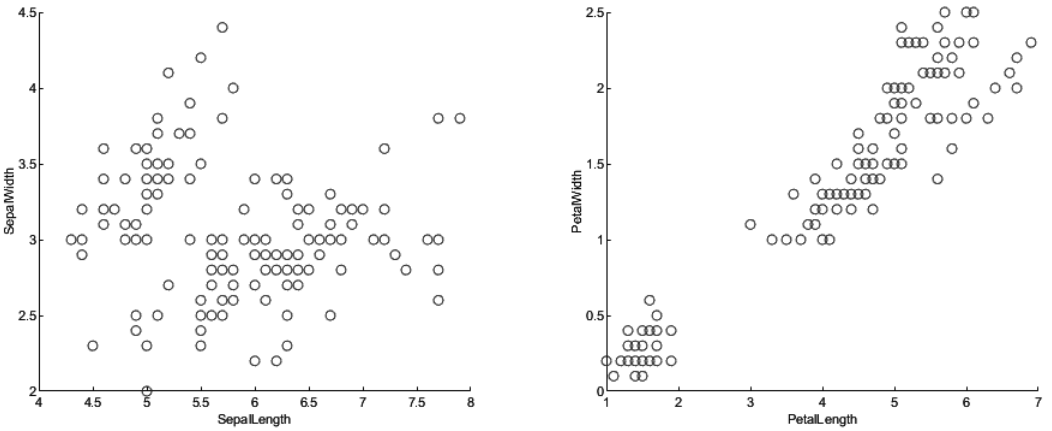


Figure 2.5 – Scatter plot of the features

MATLAB also provides other correlation-related functions, such as `corr()` for calculating the correlation coefficient matrix and `corrplot()` for creating correlation plots with customizable visualizations. These functions offer additional options and flexibility for correlation analysis in MATLAB.

Summary

In this chapter, we began our exploration of the MATLAB desktop and its convenient interaction features. We familiarized ourselves with the MATLAB Toolstrip, which is organized into various tabs. Subsequently, we delved into the importing capabilities of MATLAB, enabling us to read diverse types of data resources. We acquired knowledge on how to import data into MATLAB interactively and programmatically. Moreover, we comprehended the process of exporting data from the workspace and working with media files.

Next, we embarked on the challenging task of data preparation. We learned various techniques, including identifying missing values, modifying data types, replacing missing values, removing incomplete entries, organizing tables, identifying outliers, and consolidating multiple data sources. Following that, we explored exploratory statistics techniques, which enabled us to derive insightful features guiding us in selecting appropriate tools for extracting knowledge from the data. We examined measures of location such as mean, median, mode, quantiles, and percentiles. Additionally, we explored measures of dispersion such as range, IQR, variance, standard deviation, correlation, and covariance.

We delved into the realm of exploratory visualization, allowing us to identify specific trends through visual examination of the data. We acquired the knowledge to create histograms, box plots, and scatter plots. Moreover, we delved into measures of shape, including skewness and kurtosis. Finally, we analyzed two practical cases of preprocessing techniques, data scaling and correlation analysis.

The next chapter will comprehensively cover various classification and regression techniques, emphasizing a foundational understanding and practical implementation within the MATLAB environment. The focus will be on preparing data for accurate results in classification analysis and delving into the specifics of **K-nearest neighbors** (**KNN**) analysis. Additionally, the chapter will provide insights into applying regression methods to user data, exploring the core concepts and algorithms involved. The material will guide you through the preparation of data for regression analysis, ensuring precision and meaningful outcomes.

Part 2: Understanding Machine Learning Algorithms in MATLAB

This part introduces the dynamic world of predictive modeling and advanced machine learning techniques. In this exploration, we will delve into prediction using classification and regression, unraveling the art of foreseeing outcomes and trends through meticulous analysis. Then, we will navigate through the intricate landscape of clustering analysis and dimensionality reduction, where patterns emerge and complexities are distilled, offering insights into data structure. Venturing further, we will introduce the fascinating realm of artificial neural network modeling, a paradigm inspired by the human brain, enhancing our ability to comprehend intricate relationships within data. The journey culminates in the domain of deep learning and convolutional neural networks, where complex features are discerned, allowing us to unravel the hidden nuances of information embedded in vast datasets. Join us as we unravel the threads of these cutting-edge techniques, unlocking the potential to extract valuable knowledge from the intricacies of data.

This part has the following chapters:

- *Chapter 3*, *Prediction Using Classification and Regression*
- *Chapter 4*, *Clustering Analysis and Dimensionality Reduction*
- *Chapter 5*, *Introducing Artificial Neural Network Modeling*
- *Chapter 6*, *Deep Learning and Convolutional Neural Networks*

3

Prediction Using Classification and Regression

Classification algorithms return accurate predictions based on our observations. Starting from a set of predefined class labels, the classifier assigns each piece of input data a class label according to the training model. Classification algorithms learn linear or non-linear associations between independent and categorical dependent variables. For example, a classification algorithm may learn to predict the weather as clear sky, gentle showers or heavy rain, and so on. Regression relates a set of independent variables to a dependent variable, numeric or continuous, for example, predicting rainfall in units of millimeters. Through this technique, it is possible to understand how the value of the dependent variable changes as the independent variable varies. This chapter shows us how to classify an object using nearest neighbors and how to perform an accurate regression analysis in a MATLAB environment. The aim of this chapter is to provide you with an introduction, background information, and a basic knowledge of classification and regression techniques and how to perform them in MATLAB.

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

- Introducing classification methods using MATLAB
- Building an effective and accurate classifier
- Exploring different types of regression
- Making predictions with regression analysis in MATLAB
- Evaluating model performance
- Using advanced techniques for model evaluation and selection in MATLAB

Technical requirements

In this chapter, we will introduce basic concepts relating to machine learning. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. A working knowledge of the MATLAB environment is also required.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `datatraining.txt`
- `VehiclesItaly.xlsx`
- `Employees.xlsx`
- `AirfoilSelfNoise.xlsx`

Introducing classification methods using MATLAB

Classification methods are an essential component of machine learning and data analysis. These methods allow us to categorize data into predefined classes or groups based on specific characteristics or attributes. By utilizing classification algorithms, we can train models to make predictions or assign labels to new, unseen data points. Classification plays a vital role in various domains, including image recognition, spam filtering, sentiment analysis, fraud detection, and medical diagnosis. It enables us to make informed decisions, identify patterns, and gain insights from data.

There are numerous classification algorithms available, each with its own strengths, assumptions, and applications. Some common classification methods include decision trees, **support vector machines** (**SVMs**), random forests, logistic regression, and naive Bayes classifiers. SVM has two variations: SVC for classification and SVR for regression. To effectively employ classification methods, we need to understand the underlying concepts, techniques, and evaluation metrics associated with them. Additionally, data preparation and feature engineering play crucial roles in ensuring accurate and reliable classification results.

Throughout our learning journey, we will explore various classification methods, learn how to implement them in the MATLAB environment, and understand the necessary steps for data preparation and model evaluation. By the end, you will be equipped with the knowledge and skills to apply classification techniques to your own datasets and extract valuable insights. Let's start by exploring decision trees.

Decision trees for decision-making

A **decision tree** serves as a visual representation of a choice or decision being made. It captures the complexity of decision-making processes where the most interesting option may not always be the most useful, and situations may not always offer clear-cut choices. Often, decisions are determined by a series of conditional factors that need to be evaluated. Representing this concept solely through tables and numbers can be challenging, as the justification behind the decision is not immediately apparent.

A decision tree structure helps us convey the same information with enhanced readability by emphasizing the specific branches that lead to the decision or evaluation. The technology of decision trees is valuable in identifying strategies or achieving goals by creating models with probable outcomes.

The graphical representation of a decision tree immediately guides readers to comprehend the result. A visual representation is much more effective than a table filled with numbers. The human mind prefers to see the solution first and then trace back to understand the reasoning behind it, rather than being presented with a series of algebraic descriptions, percentages, and data to describe a result.

A decision tree is composed of the following elements:

- **Nodes**: These contain the names of independent variables or factors involved in the decision-making process.
- **Branches**: These are labeled with the possible values of the independent variables, representing different pathways or choices.
- **Leaf nodes**: These represent the classes or outcomes, where observations are grouped based on the values of an independent variable. Leaf nodes are connected to the nodes through the branches.

Using these tools, we assign labels to data and classes to represent the confidence level of the classification itself. The decision tree provides the probability or likelihood of belonging to a specific class, reflecting the level of confidence in the classification.

The process of classification begins with a set of pre-classified data known as the training set. The goal is to define rules that characterize different classes based on this data. Once the model is built, it is tested using a separate test set. The resulting descriptions or classes are then generalized through inference or induction. These generalized descriptions are then used to classify records whose membership class is unknown.

Decision trees offer a straightforward approach to classifying objects into a finite number of classes. They are constructed by recursively dividing the records into homogeneous subsets based on a target attribute, which must be categorical.

There are two types of classification rules: univariate and multivariate. **Univariate rules** consider a single predictor or target attribute at a time, while **multivariate algorithms** represent the predictor as a linear combination of variables. The process of subdivision results in a hierarchical tree structure, where subsets are referred to as nodes, and the final nodes are called leaf nodes. Nodes are labeled with the attribute name, branches are labeled with possible attribute values, and leaf nodes are labeled with different values representing the membership classes.

To classify an object using a decision tree, we use the following steps:

1. Begin at the root of the tree.
2. Select the instance attribute associated with the current node.
3. Follow the branch corresponding to the attribute value assigned to the instance.
4. If a leaf node is reached, return the label associated with that leaf. Otherwise, repeat from *step 2*, starting from the current node.

By traversing the decision tree, the path taken represents the classification rules or production rules. The branches represent the values assumed by different attributes, and the leaves represent the classification outcomes. Each rule is written along the tree, from the node to the corresponding leaf. All possible paths in the tree represent the different classification rules.

In summary, when classifying an instance using a decision tree, the process involves starting from the root, selecting the appropriate attribute at each node, following the corresponding branch based on the attribute value, and repeating these steps until a leaf node is reached, and then returning the associated label.

Exploring decision trees in MATLAB

In the MATLAB environment, the Statistics and Machine Learning Toolbox provides the necessary tools for constructing a classification tree from raw data. In this section, we will delve into a well-known example featured in many machine learning books—the iris flower dataset. Let's explore how to handle this within the MATLAB environment. We already used this dataset in *Chapter 2, Working with Data in MATLAB.*

Within this dataset, there are 50 samples belonging to each of three iris species: Iris setosa, Iris virginica, and Iris versicolor. Each sample consists of four measured features—sepal length, sepal width, petal length, and petal width—all measured in centimeters. The dataset includes the following variables:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm
- Class: `setosa`, `versicolour`, `virginica`

Our goal is to create a classification tree that can accurately classify flower species based on the size of their sepals and petals. Fortunately, there is no need to connect to the previously mentioned external archive to upload data to MATLAB's workspace. MATLAB already includes a file containing the necessary data within its software distribution:

1. To import this data, simply execute the following command:

   ```
   load fisheriris
   ```

   In MATLAB, two variables, namely `meas` and `species`, have been successfully imported. `meas` contains the measurements for sepal and petal length and width, forming a 150x4 double matrix. On the other hand, `species` represents the classification information and is structured as a 150x1 cell array. Now, let's examine the distribution of the three species within the `species` variable:

   ```
   tabulate(species)
          Value    Count   Percent
   ```

```
      setosa       50     33.33%
  versicolor       50     33.33%
   virginica       50     33.33%
```

After careful analysis, it has been determined that the sample is evenly distributed among the three species. To gain an overview of the listed floral species' features, we can utilize a scatter plot matrix. This showcases the scatter plots of the species' features in a convenient matrix format.

2. To effectively display the characteristics of the species in pairs and distinguish observations within their respective groups, we can leverage the `gplotmatrix()` function. This function is specifically designed to create a matrix of scatter plots:

```
gplotmatrix(meas, meas, species);
```

The variables we want to compare in pairs are contained in the `meas` variable, while the groups are contained in the species variable. Each set of axes in the resulting figure contains a scatter plot of a column of `meas` against a column of `meas`. All graphs are grouped to describe bivariate relationships between combinations of variables.

Figure 3.1 depicts the scatter plot of the floral features for the three iris species.

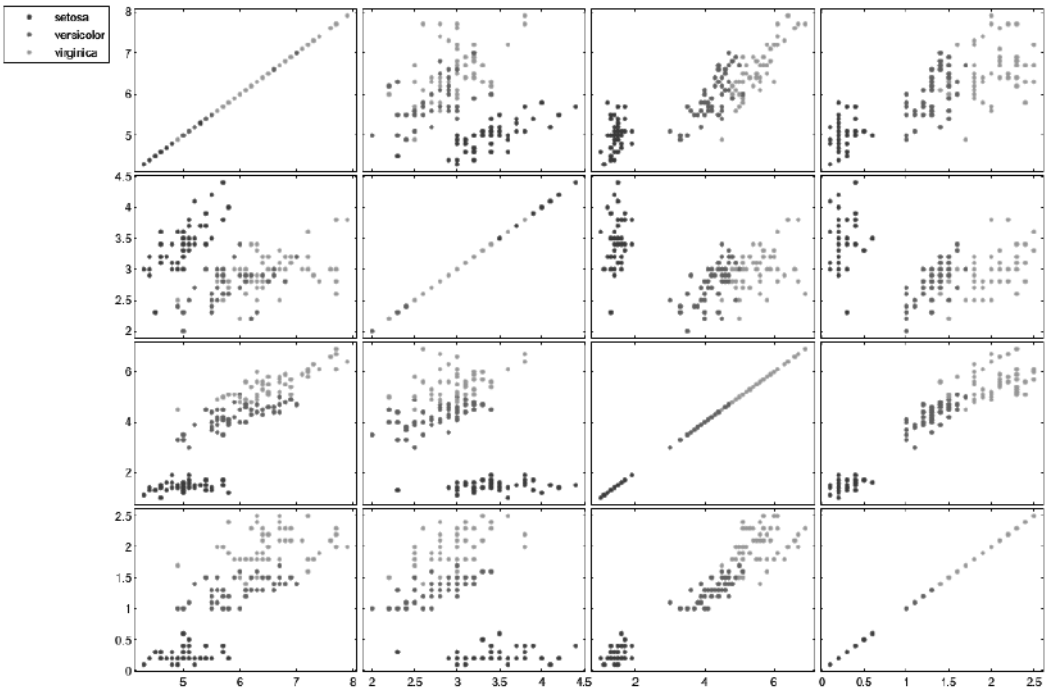


Figure 3.1 – Matrix of scatter plots grouped by species

Upon a preliminary analysis of *Figure 3.1*, it becomes evident that the `setosa` species' values are distinctly separated from the other two species, because the markers are located at various distances from the others. Conversely, the values of the other two species exhibit overlapping patterns across all plots.

Now, let's focus on investigating the variations in petal measurements across different species. To achieve this, we can utilize the two columns containing petal measurements, specifically the third and fourth columns. To visually represent the distribution of these measurements by species, we can employ a modified version of the scatter plot that we have used previously. This can be achieved using the `gscatter()` function, which generates a scatter plot grouped by a specified grouping variable. The function requires two vectors of the same size as arguments. The grouping variable must be provided in the form of a categorical variable, vector, character array, or cell array of character vectors:

```
gscatter(meas(:,3), meas(:,4), species,'rgb','osd');
xlabel('Petal length');
ylabel('Petal width');
```

Figure 3.2 distinctly illustrates the distribution of the three floral species, each occupying distinct regions within the plot.

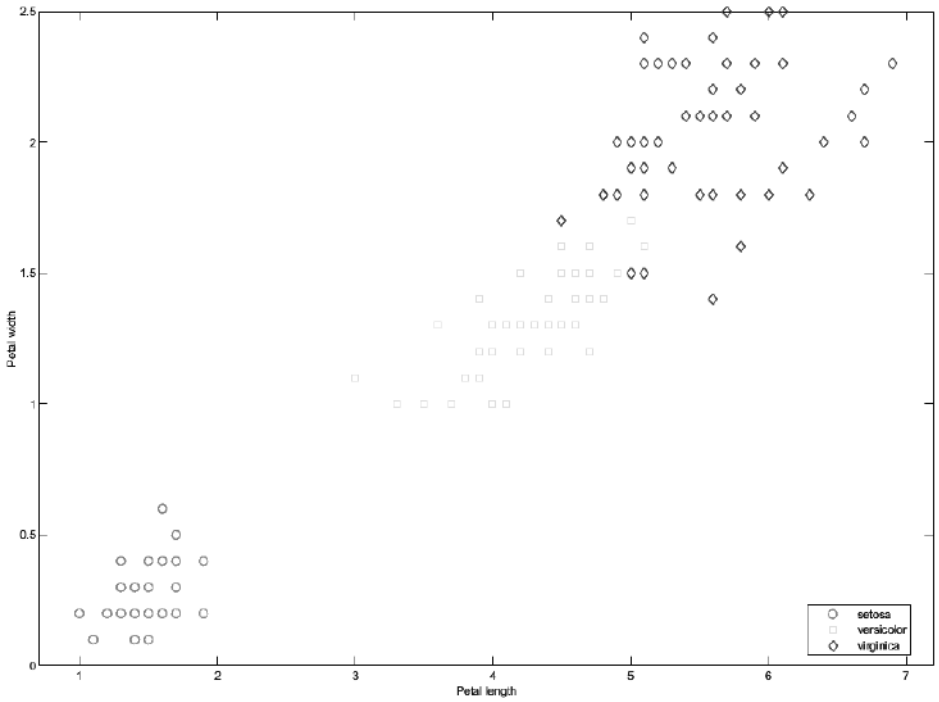


Figure 3.2 – Scatter plot grouped by species

These observations indicate that it is feasible to perform classification based on petal characteristics.

3. To create a classification tree, we can utilize the `fitctree()` function. This function generates a fitted binary decision tree by considering the input and output variables provided as inputs:

```
ClassTree= fitctree(meas,species);
```

The resulting binary tree is constructed by splitting branching nodes according to the values of a specific column within the `meas` matrix. We can visualize the created tree using the `view()` function. There are two available ways to utilize this function:

- Using the `view(ClassTree)` command will provide a text description of the tree
- Using the `view(ClassTree,'mode','graph')` command will generate a graphical representation of the tree

Let's explore both options to gain a comprehensive understanding of the tree:

```
view(ClassTree)
```

The following results are printed:

```
Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else
setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else
versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else
versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else
versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

As observed in the provided code snippet, the classification utilizes only two input variables, `x3` and `x4`, which represent the length and width of the petals, respectively. Now, let's proceed to visualize a graphical description of the tree for better comprehension:

```
view(ClassTree,'mode','graph')
```

Figure 3.3 presents a graphical representation of the tree, depicting its branches and leaf nodes. Each node includes the conditions that must be met to traverse a particular branch.

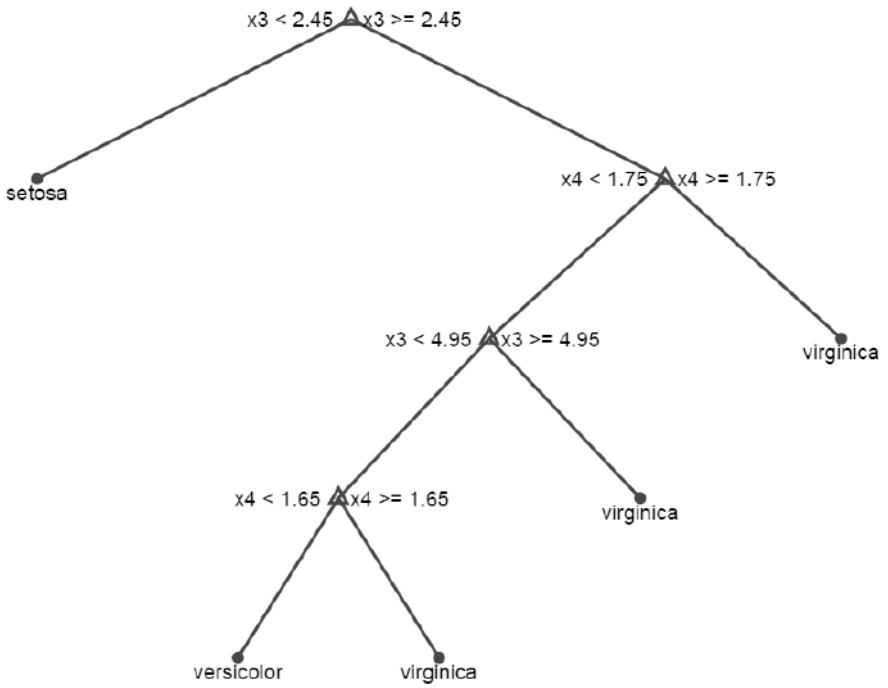


Figure 3.3 – Graphical description of the tree

Figure 3.3 offers valuable insights into the classification of the three floral species, providing immediate information. In many cases, decision tree construction is primarily focused on predicting class labels or responses. Once a tree is constructed, it becomes straightforward to predict responses for new data.

Let's consider a scenario where a new combination of four data points, representing the length and width of the sepal and petal for a specific class of floral species, has been identified. We can use this data to predict the iris species.

4. To predict the classification for new data based on the previously created and trained decision tree named `ClassTree`, use the following command:

```
predict(ClassTree,meas)
ans =
  1×1 cell array
    {'setosa'}
```

The `predict()` function returns a vector comprising the predicted class labels for the predictor data, which can be in the form of a table or matrix. These predictions are based on the trained classification tree. In the given case, only one prediction is made because the input variable contains a single record. If a data matrix with multiple observations were used, the function would produce a series of results equivalent to the number of rows in the data matrix.

Having built a classification tree from our data, the next step is to assess the model's performance in predicting new observations. Various tools are available to measure the quality of the tree. A commonly used method is **resubstitution error**, which calculates the difference between the predicted responses and the actual responses in the training data. It serves as an initial estimate of the model's performance but only provides insight in one direction. A high resubstitution error suggests that the tree's predictions may not be accurate.

To calculate the resubstitution error, you can use the following command:

```
resuberror = resubLoss(ClassTree)
resuberror =
    0.0200
```

5. The low value obtained indicates that the classification tree accurately classifies a significant portion of the data. However, to enhance the assessment of the tree's predictive accuracy, we can perform **cross-validation**. This technique assesses the model's performance by splitting the data into training and testing subsets. The tree is trained on the training data and evaluated on the testing data. **Cross-validation** helps estimate how well the tree generalizes to unseen data and avoids overfitting. MATLAB provides functions such as `crossval()` and `kfoldLoss()` for performing cross-validation.

   During cross-validation, the training data is divided randomly by default into 10 parts. Subsequently, 10 new trees are trained, with each tree utilizing 9 parts of the data for training. The predictive accuracy of each new tree is then evaluated on the remaining data that was not used for training that particular tree. Unlike the resubstitution error, this approach provides a reliable estimate of the predictive accuracy of the resulting tree since it tests the new trees on fresh, unseen data.

   Performing cross-validation helps gauge the tree's generalizability and offers a more robust measure of its predictive performance:

```
cvrtree = crossval(ClassTree)
cvrtree =
  ClassificationPartitionedModel
    CrossValidatedModel: 'Tree'
         PredictorNames: {'x1'  'x2'  'x3'  'x4'}
           ResponseName: 'Y'
        NumObservations: 150
                  KFold: 10
              Partition: [1x1 cvpartition]
             ClassNames: {'setosa'  'versicolor'  'virginica'}
         ScoreTransform: 'none'
```

   We used the `crossval()` function, which performs a loss estimate using cross-validation. A cross-validated classification model is returned. A number of properties are now available in MATLAB's workspace.

To calculate the cross-validation loss, you can utilize the `kfoldLoss()` function in the following manner:

```
cvloss = kfoldLoss(cvrtree)
cvloss =
    0.0533
```

Here, we calculated the classification loss for observations not used for training by using the `kfoldLoss()` function. The low calculated value confirms that the model works well.

The `kfoldLoss()` function calculates the average loss across all folds during cross-validation. It provides a measure of the predictive accuracy of the classification tree on unseen data. By default, `kfoldLoss()` employs 10-fold cross-validation, where the data is randomly divided into 10 parts. Each part is then used as a testing set while the remaining data is used for training the tree.

Using `kfoldLoss()` enables us to obtain a more reliable estimate of the tree's predictive accuracy as it assesses the model's performance on new and unseen data, rather than the training data itself.

At the start of this section, we listed several classification methods. In the following section, we will address SVM techniques and discriminant analysis for classification.

Building an effective and accurate classifier

Classification in machine learning is a supervised learning task that involves categorizing or classifying data into predefined classes or categories. It is one of the fundamental and widely used techniques in machine learning and data mining. The goal of classification is to develop a model or classifier that can accurately assign new, unseen instances to the correct class based on their features or attributes. The classifier learns patterns and relationships from a labeled training dataset, where each instance is associated with a known class label.

We will first discuss **SVMs**.

SVMs explained

SVMs are powerful supervised machine learning algorithms used for classification and regression tasks. They are particularly effective in solving complex problems with a clear margin of separation between classes. SVMs can handle both linearly separable and non-linearly separable data by transforming the input space into a higher-dimensional feature space. The main idea behind SVMs is to find the best possible decision boundary, known as the *hyperplane*, that maximally separates the classes while minimizing classification errors. The hyperplane is determined by a subset of training examples called support vectors, which are the closest points to the decision boundary. SVMs employ a kernel function to map the input data into a higher-dimensional space, where linear separation is more feasible. This allows SVMs to solve non-linear problems by finding non-linear decision boundaries in the transformed feature space.

SVMs excel in tackling intricate problems where there exists a discernible gap between different categories. They possess the capacity to handle datasets that can be separated both linearly and non-linearly through a process involving the transformation of the initial input space into a higher-dimensional feature space. The fundamental concept underpinning SVMs revolves around the quest for the optimal decision boundary, known as the hyperplane. SVMs leverage a kernel function to project the input data into this higher-dimensional space, where achieving linear separation becomes more attainable. This distinctive capability empowers SVMs to address non-linear problems by identifying non-linear decision boundaries within the transformed feature space.

The training of an SVM involves optimizing a cost function that penalizes misclassified examples and maximizes the margin between the support vectors and the decision boundary. There are different variations of SVMs, such as **C-support vector machine** (**C-SVM**) for classification tasks and **epsilon-SVM** for regression tasks. SVMs are widely used in various domains, including image classification, text categorization, bioinformatics, and finance. They are known for their ability to handle high-dimensional data, handle outliers well, and generalize effectively to unseen data.

An SVM-based classification technique enables the classification of both linear and non-linear datasets. In SVMs, the training data instances are represented on a plane with dimensions equal to the number of attributes in each instance. For instance, a three-dimensional plane is used to represent instances with three attributes. The three main components of an SVM classifier are as follows:

- Lines or hyperplanes, which act as boundaries to classify instances into different classes
- Margins, which are the distances between the closest instances of different classes
- Support vectors, which are the instances within the hyperplane boundaries that are challenging to classify

SVMs separate data by finding an optimal hyperplane in a higher-dimensional feature space that maximizes the margin between different classes.

SVMs can handle linearly separable datasets, where instances can be separated by straight lines on the plane, as well as non-linearly separable datasets. In the case of linearly separable instances, the goal is to find the lines or hyperplanes that maximize the margin value. This selection minimizes the classification error.

For non-linearly separable datasets, the classification process is more complex. It involves two phases that build upon the previous approach.

In the first phase, instances are mapped to a higher-dimensional space to achieve linear separability. In the second phase, the previous approach is used to find a line or hyperplane that maximizes the margin, taking advantage of the now linearly separable instances.

To handle datasets that require non-linear functions for separation, **feature spaces** are used in SVMs. This technique involves mapping the initial data to a higher-dimensional space. If the initial data has m dimensions and the feature space has n dimensions, with $m > n$, a mapping function is applied. Suppose we have an input denoted with x and y. The concept of kernel spaces is particularly useful for algorithms that rely on training data solely through scalar products. In this case, instead of explicitly finding $f(x)$ and $f(y)$ in the m-dimensional space, we only need to calculate their scalar product, denoted as $f(x) \cdot f(y)$.

The function f is employed to map the input from the original n-dimensional space to a higher-dimensional m-space. To simplify this computation, especially in large spaces, a kernel function is utilized. The kernel function directly provides the scalar product of the images, eliminating the need for explicit mapping and making the calculation more efficient.

$$K(x, y) = f(x) \cdot f(y)$$

Here, we have the following:

- x and y are input vectors
- f is a transformation function

The dot product operation, denoted by $f(x) \cdot f(y)$, calculates the dot product of the transformed input vectors. It allows us to efficiently compute the scalar product of the mapped input vectors without explicitly calculating the mapping function, $f(x)$ and $f(y)$. This dot product operation is an essential component in various algorithms that utilize kernel methods for classification and regression tasks.

The purpose of the **kernel function** is to transform the input data into a suitable form, particularly when it is not feasible to determine a linearly separable hyperplane, which is often the case. There are several commonly used kernels, including **linear**, **polynomial**, **radial basis function** (**RBF**), and **sigmoid**.

In this methodology, the training phase and the subsequent error evaluation activity play a crucial role. To accomplish this, the data is split into two subsets, named training and testing. The **training set** is utilized for algorithm training and comprises labeled inputs and outputs for supervised learning. Typically, it consists of approximately 80% of the total data. The **testing set**, on the other hand, is used to evaluate the accuracy of the SVM. It contains 20% of the remaining data that was not used in training, and it helps measure the prediction error. This phase also serves to test the algorithm in real-world scenarios, simulating the actual usage of the trained model.

One notable advantage of SVM-based classifiers is their capability to handle complex and non-linear classification problems while ensuring a high level of accuracy. However, for simpler problems, the accuracy may be comparable to that of decision tree-based classification techniques. Some drawbacks of SVMs include the relatively longer time required for model creation, although it is still faster compared to neural networks. The other drawback is the lack of interpretability of the model's inner workings.

Supervised classification using SVM

Supervised classification is a machine learning technique used to classify or categorize data into predefined classes or categories. Supervised classification relies on labeled training data, in which data points are linked to a specific class. The objective of supervised classification is to construct a model capable of making precise predictions regarding the class of previously unseen or fresh data points, utilizing their inherent features as guidance.

To understand how to implement the SVM algorithm in MATLAB, we will use a dataset for binary classification. **Binary classification** is a specific type of supervised classification where the goal is to classify data into one of two mutually exclusive classes or categories. The two classes are often referred to as the positive class and the negative class, or class 1 and class 0. In binary classification, the labeled training data consists of examples where each data point is associated with one of the two classes. The machine learning algorithm is then trained on this labeled data to learn a model that can accurately predict the class of unseen data points.

The dataset consists of features such as `Temperature`, `Humidity`, `Light`, and `CO2`, and the target variable is room occupancy. The ground-truth occupancy information was obtained by capturing time-stamped pictures every minute.

The goal of this experiment is to train a binary classification model that can predict whether the room is occupied or unoccupied based on the given sensor data. The features (`Temperature`, `Humidity`, `Light`, `CO2`, and `Humidity Ratio`) are used as input to the model, and the corresponding occupancy status (`occupied` or `unoccupied`) serves as the target variable for training and evaluation.

> **Important note**
>
> To download the dataset and a short summary of the variables contained within it, please refer to UCI Machine Learning Repository at the following link: `https://doi.org/10.24432/C5X01N`.

By analyzing the relationship between the sensor data and the ground-truth occupancy, the model aims to learn patterns and make accurate predictions on unseen data. The training process involves feeding the labeled data into a binary classification algorithm, using SVM. The algorithm learns from the features and their corresponding occupancy labels to create a model that can classify new instances.

Take the following steps:

1. We start by importing the data into the MATLAB environment. To do that, we will use the `readtable()` function, as we learned in *Chapter 2*, *Working with Data in MATLAB*, in the *Importing data into MATLAB* section. We need to use the `datatraining.txt` file, containing eight variables (`Num`, `date`, `Temperature`, `Humidity`, `Light`, `CO2`, `HumidityRatio`, and `Occupancy`). But for our purpose, only six variables are needed, that is, the last six. So, we have to set some options to use in the function call, as follows:

```
opts = detectImportOptions('datatraining.txt');
```

The `detectImportOptions()` function in MATLAB is used to automatically determine the import options for a specific file. It analyzes the file's format and content to provide a suggested set of options that can be used with other file input functions, such as `readtable()` or `readmatrix()`, to import the data from the file. The function returns an `opts` object, which is an instance of the class. This object contains various properties and methods that allow you to inspect and modify the import options. We can explore the data detected by simply calling the name of the object (`opts`).

The first two columns are not useful for our purpose, so we can modify this property using the `opts.SelectedVariableNames` option, as follows:

```
opts.SelectedVariableNames = [3:8];
```

Only the last six variables have been selected.

2. Now we are ready to import the dataset:

```
DataMatrix = readtable('datatraining.txt',opts);
```

We can see, in the MATLAB workspace, a new variable named `DataMatrix` as a table of size 814x6. This table contains the five predictors (`Temperature`, `Humidity`, `Light`, `CO2`, and `HumidityRatio`) and one dichotomic response (`Occupancy`).

3. To train an SVM classifier for binary classification problems, we can use the `fitcsvm()` function as follows:

```
SVMClassifier =  fitcsvm(DataMatrix(:,1:5),DataMatrix(:,6));
```

The `DataMatrix(:,1:5)` parameter represents the predictor data, which is an 8,143-by-5 matrix, where 8,143 is the matrix's dimension consisting of the number of observations, denoting data points, and 5 signifies the count of predictor variables, or features. Each row within this matrix represents a distinct observation, while each column pertains to an individual predictor variable. The `DataMatrix(:,6)` parameter represents the response data, which is an 8,143-by-1 vector. For binary classification, this vector should be a binary vector of 0s and 1s, where 0 represents the negative class and 1 represents the positive class.

The function returns an `SVMClassifier` object, which represents the trained SVM classifier. This object contains information about the trained model, such as support vectors, coefficients, and kernel parameters. We can print some of this information simply by typing the name of the model into the MATLAB command prompt:

```
SVMClassifier
SVMClassifier =
  ClassificationSVM
           PredictorNames:
{'Temperature'  'Humidity'  'Light'  'CO2'  'HumidityRatio'}
             ResponseName: 'Occupancy'
    CategoricalPredictors: []
               ClassNames: [0 1]
```

```
             ScoreTransform: 'none'
            NumObservations: 8143
                      Alpha: [256×1 double]
                       Bias: -7.1986
          KernelParameters: [1×1 struct]
            BoxConstraints: [8143×1 double]
           ConvergenceInfo: [1×1 struct]
           IsSupportVector: [8143×1 logical]
                    Solver: 'SMO'
```

The `fitcsvm()` function applies the linear kernel by default. It's possible to set a different kernel using the `KernelFunction` parameter. The supported kernels are the following:

- **Linear**: This is a commonly used kernel function. It assumes that the data can be separated by a linear decision boundary or hyperplane. The linear kernel computes the dot product between the input feature vectors, which effectively measures the similarity between the samples in the original feature space.
- **Polynomial**: This can be used to handle non-linearly separable data. It maps the input feature vectors into a higher-dimensional space using polynomial functions, which allows for the learning of non-linear decision boundaries. This kernel requires setting the degree of the polynomial and the scale factor.
- **Gaussian or RBF**: This can handle non-linearly separable data by mapping the input feature vectors into an infinite-dimensional feature space. The RBF kernel is particularly effective when the decision boundary is complex or not well defined. This kernel requires setting the kernel scale or gamma parameter, which determines the influence of each training example.

4. Calculating the resubstitution error helps us gauge the performance of the method employed. As mentioned earlier, this error is determined by measuring the disparity between the predicted and actual responses in the training data. While it offers an initial estimate of the model's performance, it only provides insights in one direction. A high resubstitution error implies that the predictions made by the tree model may not be accurate:

```
ResubError = resubLoss(SVMClassifier);
```

The resubstitution error is typically expressed as a percentage or a decimal value between 0 and 1, where 0 represents a perfect fit (no errors) and 1 represents a complete mismatch (all predictions are incorrect). Let's see the results obtained:

```
ResubError = 0.1459
```

Based on our analysis, we have encountered an error of at least 15%. This suggests that there is room for improvement in the performance of the classifier. It is likely that the data cannot be effectively separated using a linear plane, indicating the need to modify the kernel function.

5. To improve the performance of the model, we can choose an SVM classifier based on the RBF kernel:

```
RbfSVMClassifier = fitcsvm(DataMatrix(:,1:5),DataMatrix-
(:,6),'Standardize',true,'KernelFunction','RBF','KernelS-
cale','auto');
```

Now, we can repeat the resubstitution error calculation to compare the performance of the two models:

```
ResubError = resubLoss(RbfSVMClassifier);
```

The following result is returned:

```
ResubError = 0.0061
```

We initially had an error rate of 15%, but we have successfully reduced it to 0.61%. This significant improvement indicates that the model is robust in handling non-linearity in the data.

Now that we have introduced classification, it is time to explore the world of regression, which allows us to work with continuous numerical values.

Exploring different types of regression

Regression analysis is a statistical method used to examine the connection between a group of independent variables (also known as explanatory variables) and a dependent variable (referred to as the response variable). By employing this technique, it becomes possible to comprehend how the value of the response variable fluctuates when the explanatory variable is altered.

Regression analysis serves a dual purpose: explanatory and predictive. The **explanatory** role helps us understand and assess the impact of independent variables on the dependent variable based on a specific theoretical model. It allows us to quantify the relationship and determine the magnitude and significance of the effects. In the **predictive** role, regression analysis aims to identify the optimal linear combination of independent variables to predict the value of the dependent variable accurately. By utilizing this technique, we can make predictions based on the observed relationships between variables.

Introducing linear regression

To characterize the relationship between variables, we can employ a mathematical function that captures the observed behavior, interpolates the data, and represents its underlying trend while retaining its key information. Linear regression is a method that specifically aims to identify a line that can effectively represent the distribution of points in a two-dimensional plane. When the observed points closely align with the line, it indicates that the chosen model accurately describes the relationship between the variables.

While there are theoretically infinite lines that can interpolate the observations, in practice, only one mathematical model can optimize the data representation. In the case of a linear mathematical relationship, the observations of the variable *y* can be derived from a linear function of the observations of the variable *x*. For each observation, this relationship can be expressed as follows:

$$y = \alpha^* x + \beta$$

In the given formula, *x* represents the explanatory variable, and *y* represents the response variable. The parameters α and β correspond to the slope of the line and the intercept with the *y* axis, respectively. These parameters need to be estimated using the collected observations for both variables included in the model.

Of particular interest is the slope α, which signifies the change in the average response for each unit increase in the explanatory variable. What happens when this coefficient changes? If the slope is positive, the regression line ascends from left to right, indicating that the response increases as the explanatory variable increases. Conversely, if the slope is negative, the line descends from left to right, suggesting that the response decreases with an increase in the explanatory variable. When the slope is 0, it implies that the explanatory variable has no effect on the response value.

However, the significance of the relationship between the variables is not solely determined by the sign of α; its value is also crucial. In the case of a positive slope, the average response is higher when the explanatory variable is higher. Conversely, for a negative slope, the average response is lower when the explanatory variable is higher. The magnitude of the slope plays a vital role in understanding the strength and nature of the relationship between the variables.

In MATLAB, you can perform simple linear regression using the built-in functions and tools available in the Statistics and Machine Learning Toolbox. Another approach to perform simple linear regression in MATLAB is by using the `polyfit()` and `polyval()` functions. These functions allow you to fit data to a pattern that is linear in the coefficients.

Linear regression model in MATLAB

A convenient method to construct a linear regression model in MATLAB is by employing the `fitlm()` function, which is a part of the Statistics and Machine Learning Toolbox. This function offers a straightforward approach to creating and analyzing linear regression models.

The `fitlm()` function generates a `LinearModel` object in MATLAB. This object contains various properties that can be easily examined by simply selecting it. Additionally, the `LinearModel` object offers several methods, such as `plot`, `plotResiduals`, and `plotDiagnostics`, which facilitate creating plots and conducting diagnostic analysis.

A `LinearModel` object encapsulates training data, model description, diagnostic information, and fitted coefficients for a linear regression. By default, when using the `fitlm()` function, the last variable in a table or dataset array is considered the response variable. Alternatively, you can specify the predictors and response variables explicitly using a formula. Moreover, you have the flexibility

to designate a specific column as the response variable by utilizing the `ResponseVar` name-value pair argument. To define a set of columns as predictors, you can employ the `PredictorVars` name-value pair argument. The predictor variables can be numeric or of any grouping variable type, such as logical or categorical. However, the response variable must be numeric or logical in nature.

1. To gain insight into the functionality of the `fitlm()` function, let's consider a dataset that includes information on the number of vehicles registered in Italy and the population of various regions. The dataset includes the following fields:

   - Names of Italian regions (`Region`)
   - Vehicle registrations for each region (`Registrations`)
   - Resident population in each region (`Population`)

   Let's start by importing the data into a table:

   ```
   VehicleData = readtable('VehiclesItaly.xlsx');
   ```

2. To fit a linear regression model with the `Registrations` variable as the response and `Population` as the explanatory variable (predictor), we can use the `fitlm()` function as follows:

   ```
   LRModel = fitlm(VehicleData,'Registrations~Population');
   ```

   The `disp()` command displays the model summary, providing information about the coefficients, p-values, R-squared value, and other statistical metrics:

   ```
   disp(LRModel)
   ```

   The following information is printed:

   ```
   Linear regression model:
       Registrations ~ 1 + Population
   Estimated Coefficients:
               Estimate       SE       tStat       pValue
              ________    ________    ______    __________
   (Intercept) 70549        41016      1.72       0.10258
   Population 0.59212    0.010488    56.458    1.0323e-21
   Number of observations: 20, Error degrees of freedom: 18
   Root Mean Squared Error: 1.16e+05
   R-squared: 0.994, Adjusted R-Squared: 0.994
   F-statistic vs. constant model: 3.19e+03, p-value = 1.03e-21
   ```

The results of the linear regression model include the model formula and estimated coefficients. Each term in the model is represented by a row, and the following columns provide additional information:

- `Estimate`: The estimated coefficient value for each corresponding term in the model.
- `SE`: The standard error of the estimate.
- `tStat`: The t-statistic for each coefficient, which tests the null hypothesis that the coefficient is 0 against the alternative that it is different from 0, given the other predictors in the model.
- `pValue`: The p-value for the F-statistic of the hypothesis test that the coefficient is equal to 0. In our example, a p-value lower than 0.05 indicates that the term is significant at the 5% significance level, given the other terms in the model.
- Interpretation of the intercept adds more meaning to the problem statement, as an average 70,549 registrations are happening in the city.
- We can also explain the slope (0.59212) as with every 1,000-person increase in population, registrations increase by 592, with approximately 6 out of every 10 people having vehicles.

The **t-test** is a parametric test used to determine whether the average value of a distribution significantly differs from a reference value. In addition, the following information is provided about the created model:

- **Number of observations**: The number of records in the data without any NaN values. In our example, there are 20 observations.
- **Error degrees of freedom**: The number of observations minus the number of estimated coefficients, stored as a positive integer value. In our example, with two predictors, the error degrees of freedom are 18.
- **Root mean squared error (RMSE)**: The root mean squared error of the residuals, representing the square root of the mean squared error.
- **R-squared**: The proportion of the total sum of squares explained by the model, also known as the coefficient of determination. In our example, the high R-squared value suggests that the model explains approximately 99% of the variability in the `Registrations` response variable.
- **Adjusted R-squared**: A modified version of R-squared that takes into account the number of predictors in the model.
- **F-statistic versus constant model**: The test statistic for the F-test on the regression model, which tests for a significant linear regression relationship between the response variable and the predictor variables.
- **p-value**: The p-value for the F-test on the model. In our example, the model is significant with a very low p-value of 1.03e-21.

> Let's examine the results of the last MATLAB command. Two values stand out among the others: R-squared and p-value. The calculated R-squared value is very high, equal to 0.994. This indicates that a substantial variation in the response variable can be explained by the predictor variable. On the other hand, the p-value is very small, but understanding its significance requires further exploration. During a statistical significance test, we start by assuming the **null hypothesis**, which states that there is no difference between the groups with respect to the parameters being considered. Under the null hypothesis, any observed differences are attributed to chance.

Now we must make a decision, whether to accept or reject the null hypothesis. To make this determination, we analyze our data using a significance test. If the test *suggests* rejecting the null hypothesis, we declare the observed difference as statistically significant. Conversely, if the test *advises* accepting the null hypothesis, the difference is deemed statistically not significant. The results of a statistical test always come with a certain level of uncertainty and probability. Thus, a decision to reject the null hypothesis is likely correct but may also be incorrect. Assessing the risk of making an error is known as the **significance level** of the test.

This significance level, also known as the p-value, provides a quantitative estimate of the probability that the observed differences are due to chance. The **p-value** is a probability and can only assume values between 0 and 1. A p-value approaching 0 indicates a low probability that the observed difference is due to chance. Researchers typically choose a significance level of 0.05 (5%) or 0.01 (1%). In our case, we calculated a p-value of 1.03e-21, which is far below the chosen significance level. This suggests that the observed difference is statistically significant and not likely due to chance.

After having seen the first example of linear regression that deals with continuous predictors, it is necessary to see how to deal with the case in which the type of predictors is different.

Making predictions with regression analysis in MATLAB

Having explored numerous instances of linear regression, we can confidently assert that we comprehend the underlying mechanisms of this statistical method. Non-linear regression is used to model the relationship between a dependent variable and one or more independent variables when the relationship is not linear. In contrast to linear regression, where the relationship is assumed to be a straight line, non-linear regression allows for more complex and flexible relationships between variables.

Up until now, we have exclusively employed continuous variables as predictors. However, what transpires when the predictors are categorical variables? No need to fret, as the fundamental principles of regression techniques remain unchanged.

Multiple linear regression with categorical predictor

Categorical variables differ from numerical ones as they do not stem from measurement operations but rather from classification and comparison operations. Categorical variables can be categorized into nominal, dichotomous, or ordinal groups.

Now, let's delve into a real-life scenario. Within a company, we have diligently gathered information on employee salaries, which are determined by their years of experience. Our objective is to construct a model that enables us to track the progression of an employee's salary over time. We have categorized the employees into three types: Management, Technical Staff, and General Staff.

Let's now see practically how to process the MATLAB code:

1. To begin our analysis, we will import the relevant data from an Excel worksheet named `employees.xlsx` into MATLAB:

```
EmployeesSalary = readtable('employees.xlsx');
```

We now have a table containing three variables: `YearsExperience`, `Salary`, and `LevelOfEmployee`. The nature and meaning of these variables are readily apparent.

Let's examine the characteristics of these variables:

```
summary(EmployeesSalary)
```

The following results are printed:

```
Variables:
    YearsExperience: 120×1 double
        Values:
            Min            1
            Median      20.5
            Max            40
    Salary: 120×1 double
        Values:
            Min           20
            Median        41
            Max           82
    LevelOfEmployee: 120×1 cell array of character vectors
```

The first two variables, `YearsExperience` and `Salary`, are of the `double` data type, representing numerical values. However, the third variable, `LevelOfEmployee`, consists of character vectors. It is evident that this variable represents a categorical variable since it contains the three types of employees mentioned earlier. Nevertheless, MATLAB is unaware of this categorization, and it is our responsibility to inform it. In practice, we need to transform this variable into a categorical type:

```
EmployeesSalary.LevelOfEmployee=categorical(EmployeesSalary.
LevelOfEmployee);
```

Now we can check the type of the variable:

```
class(EmployeesSalary.LevelOfEmployee)
ans =
    'categorical'
```

So, the variable is corrected and recognized by MATLAB.

2. Now, it is time to create the regression model by applying the `fitlm()` function using `Salary` as the dependent variable, and `YearsExperience` and `LevelOfEmployee` as the independent variables. Given that `LevelOfEmployee` is a categorical variable with three levels (`Management`, `TechnicalStaff`, and `GeneralStaff`), it is represented in the model as two indicator variables. In MATLAB, categorical predictors are typically included as dummy indicator variables. An indicator variable takes on values of 0 or 1. For a categorical variable with *n* categories, it can be represented by *n – 1* indicator variables.

   To account for the distinctions between the different employee types, we can incorporate interaction terms between `YearsExperience` and `LevelOfEmployee`. This allows us to capture the interaction effects between years of experience and the specific employee category:

```
LRModelCat =
fitlm(EmployeesSalary,'Salary~YearsExperience*LevelOfEmployee');
```

   Let's see the characteristics of the model:

```
LRModelCat =
Linear regression model:
    Salary ~ 1 + YearsExperience*LevelOfEmployee
Estimated Coefficients:
                                                Estimate       SE       tStat       pValue
                                                ________    ________    ______    _________

    (Intercept)                                  20.199      0.40884    49.404    8.0624e-79
    YearsExperience                             0.25061     0.017378    14.421    1.8358e-27
    LevelOfEmployee_Management                   30.247      0.57819    52.314    1.5568e-81
    LevelOfEmployee_TechnicalStaff               10.369      0.57819    17.933      5.66e-35
    YearsExperience:LevelOfEmployee_Management  0.48756     0.024576    19.839    9.0452e-39
    YearsExperience:LevelOfEmployee_TechnicalStaff 0.23745  0.024576     9.662    1.7076e-16

Number of observations: 120, Error degrees of freedom: 114
Root Mean Squared Error: 1.27
R-squared: 0.995,  Adjusted R-Squared: 0.995
F-statistic vs. constant model: 4.66e+03, p-value = 4.84e-130
```

Figure 3.4 – Regression model summary

   Based on the results, the model equation is as follows:

```
Salary =
  20.2
+ 0.25 * YearsExperience
+ 30.2 * LevelOfEmployee(Management)
+ 10.4 * LevelOfEmployee(TechnicalStaff)
+ 0.49 * YearsExperience * LevelOfEmployee(Management)
+ 0.24 * YearsExperience * LevelOfEmployee(TechnicalStaff)
```

   In this equation, the term `LevelOfEmployee(GeneralStaff)` is not included because the first level, by default, serves as the reference group. However, the first-order terms for `YearsExperience` and `LevelOfEmployee`, along with all the interactions, are present.

It is apparent that a single equation for the entire system is insufficient to obtain accurate wage estimates. To address this, we need to differentiate between the three categories of employees and create separate models for each. Consequently, we obtain the following three equations to capture the wage dynamics for each employee category:

```
LevelOfEmployee(GeneralStaff):
Salary = 20.2 + 0.25 * YearsExperience
LevelOfEmployee(TechnicalStaff):
Salary = (20.2 + 10.4) + (0.25 + 0.24) * YearsExperience
LevelOfEmployee(Management):
Salary = (20.2 + 30.2) + (0.25 + 0.49) * YearsExperience
```

From a simple comparison between the three equations we have just seen, we can see that the intercept progressively increases with the category of employees (`20.2;20.2 + 10.4; 20.2 + 30.2`); this shifts the regression line upward. Similarly, the slope also progressively increases with the employee category (`0.25 ; 0.25 + 0.24; 0.25 + 0.49`), leading to a greater increase in salary as the number of years of experience increases.

3. To enhance our understanding of the progress made thus far, let's incorporate the following lines into the scatter plot of the data:

```
Xvalues = linspace(min(EmployeesSalary.YearsExperience),max(Em-
ployeesSalary.YearsExperience));
figure()
gscatter(EmployeesSalary.YearsExperience, EmployeesSalary.
Salary, EmployeesSalary.LevelOfEmployee,'bgr','x.o')
title('Salary of  Employees versus Years of the Experience,
Grouped by Level of Employee')
line(Xvalues,feval(LRModelCat,Xvalues,'General-
Staff'),'Color','b','LineWidth',2)
line(Xvalues,feval(LRModelCat,Xvalues,'Technical-
Staff'),'Color','r','LineWidth',2)
line(Xvalues,feval(LRModelCat,Xvalues,'Manage-
ment'),'Color','g','LineWidth',2)
```

The results are shown in *Figure 3.5*.

In the previously suggested code, we first created a linearly spaced vector between the minimum and maximum values of the `YearsExperience` variable using the `linspace()` function. Subsequently, we generated a scatter plot depicting the relationship between employee salaries (`Salary`) and years of experience (`YearsExperience`), with the data points grouped by the level of employee (`LevelOfEmployee`).

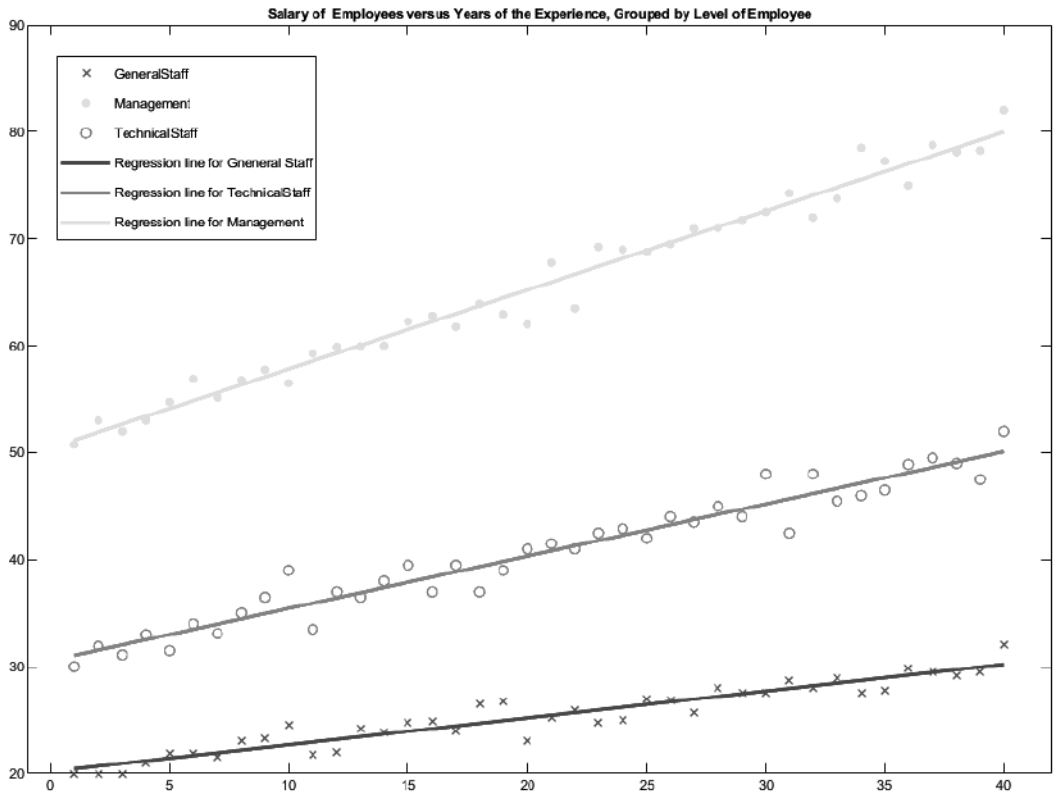


Figure 3.5 – Scatter plot with the three straight lines that fit three data groups

4. Finally, we added the three lines representing the respective trends for each employee category. This was accomplished by utilizing the `feval()` function to evaluate the model at the specified points in the `Xvalues` variable. It is now evident that the three straight lines, corresponding to the three equations, are distinguished by both their intercepts and slopes.

After having analyzed two regression examples in detail, we can see how to evaluate the performance of the models created so far.

Evaluating model performance

Model performance refers to how well a model fits the given data and accurately predicts outcomes. It is important to evaluate model performance to assess its reliability and effectiveness in making predictions or in capturing the underlying patterns in the data. One commonly used metric to evaluate model performance is the R-squared value, also known as the coefficient of determination. R-squared measures the proportion of the variance in the dependent variable that can be explained by the independent variables in the model. A higher R-squared value indicates a better fit, as it means a larger proportion of the variability in the data is accounted for by the model.

However, R-squared alone may not provide a complete picture of model performance. Other metrics, such as **mean squared error** (**MSE**) or **mean absolute error** (**MAE**), can be used to assess the average prediction error of the model. Lower values of MSE or MAE indicate better predictive performance. Furthermore, it is important to consider the context and specific requirements of the problem at hand. For example, if the residuals (the differences between the predicted and actual values) exhibit certain patterns or non-random behavior, it may indicate that the model is missing important factors or violating assumptions.

Additionally, cross-validation techniques can be used to further evaluate the model's performance by assessing its ability to generalize to new, unseen data. This helps to ensure that the model is not overfitting the training data and can make accurate predictions on new observations. In summary, model performance is assessed through various metrics, including R-squared, MSE, MAE, and consideration of the residuals and cross-validation results. Evaluating these aspects provides a comprehensive understanding of how well the model performs and its suitability for the given task.

In the next example, we will exploit the evaluation metrics to improve the performance of the model by identifying the outliers.

Reducing outlier effects

As mentioned earlier, the `fitlm()` function generates a `LinearModel` object that contains valuable information about the linear regression, including training data, model description, diagnostic details, and estimated coefficients. Now, we can utilize some of these properties to gain additional insights from the model. The least-squares method is commonly employed when we have sufficient knowledge about the model's shape and aim to determine its parameters. It is also useful for model exploration.

However, this method requires manual inspection of the data to identify and handle outliers. Therefore, we will now examine whether there are any outliers in the data that should be excluded from the fitting process. Residual plots can aid us in this analysis. The most used plots include the default histogram plot, which displays the range and frequencies of the residuals, and the probability plot, which compares the distribution of the residuals to a normal distribution with a similar variance:

1. We start as usual by importing the data:

   ```
   VehicleData = readtable('VehiclesItaly.xlsx');
   ```

   Then we build the model:

   ```
   LRModel = fitlm(VehicleData,'Registrations~Population');
   ```

   Now, we can work on the evaluation of the model.

2. To start, we extract some different evaluation metrics of the model:

   ```
   LRModel.Rsquared
   ```

The following results are returned:

```
Ordinary: 0.9944
Adjusted: 0.9941
```

Let's extract another metric:

```
LRModel.MSE
```

Let's see the value obtained:

```
1.3407e+10
```

MSE quantifies the mean squared disparity between the model's predicted values and the actual observed values. The MSE is calculated by taking the average of the squared differences between the predicted values and the true values over the entire dataset.

The R-squared and MSE values obtained will be used to compare the performance of the different models.

3. To show how residual plots are useful for identifying outliers, we will refer to the example used in the *Linear regression model in MATLAB* section. We will use the same model (`LRModel`). To visualize the residuals, we will utilize a specific property of the `LinearModel` object:

```
plotResiduals(LRModel)
```

Upon analyzing the histogram obtained, a distinct asymmetry is evident in the negative values. Specifically, the observations near `-2 *105` appear to be potential outliers. To achieve a more accurate fit, we will construct a probability plot. As previously mentioned, the plot in *Figure 3.6* demonstrates the comparison between the distribution of residuals and a normal distribution with similar variance.

The probability plot of residuals also reveals potential outliers, particularly in the lower left region. We can observe three values that significantly deviate from the dotted line. On the other hand, the probability plot appears reasonably linear for the remaining residual values, indicating a reasonable fit to normally distributed residuals.

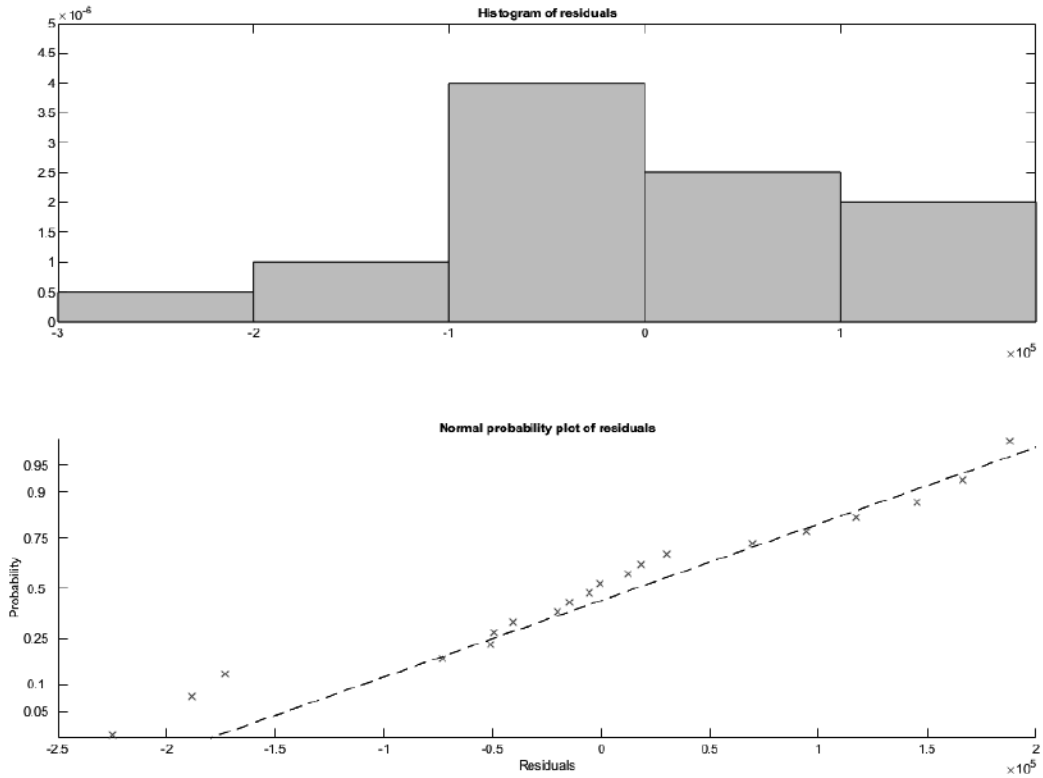


Figure 3.6 – Histogram of residuals and probability plot of residuals for the linear regression model

4. We can identify these outliers and proceed to remove them from the dataset. To locate them, we can utilize the `find()` function, examining values to the left of the abscissa equal to `-1.5*105`:

```
outliers = find(LRModel.Residuals.Raw < -1.5*10^5)
```

The following results are returned:

```
outliers =
     9
    13
    18
```

Currently, we have the outliers detected in the data.

5. Now, at this stage, we can proceed to create the model once again, this time excluding the aforementioned outlier values:

```
LRModel2 =
fitlm(VehicleData,'Registrations~Population','Exclude',outliers)
```

Now we can extract the evaluation metrics to compare the models:

```
LRModel2.Rsquared
```

The following values are printed:

```
Ordinary: 0.9974
Adjusted: 0.9972
```

The model's performance has noticeably improved, as indicated by the increased R-squared value of `0.997` compared to the previous model's R-squared value of `0.994`. This improvement suggests that the new model provides a better fit to the data. Now we will extract the MSE:

```
LRModel2.MSE
ans =
   6.8047e+09
```

From a comparison, we can see an improvement; the MSE decreased from `1.3407e+10` to `6.8047e+09`.

To further evaluate the model's performance, we can assess other metrics, such as the MSE or MAE. Additionally, conducting cross-validation or assessing the residuals' distribution and patterns can provide further insights into the model's effectiveness.

Having discussed the utilization of evaluation metrics to enhance model performance, it is now opportune to delve into these methodologies and strive toward constructing increasingly effective models. By employing these techniques, we aim to improve the overall performance and predictive capabilities of our models.

Using advanced techniques for model evaluation and selection in MATLAB

Model evaluation and selection are crucial steps in machine learning to ensure the chosen model performs well on unseen data and generalizes effectively. When it comes to advanced techniques for model evaluation and selection in MATLAB, there are several approaches you can consider.

In the subsequent sub-section, we will take a look at the most important techniques for model evaluation and selection.

Understanding k-fold cross-validation

K-fold cross-validation is a widely used technique for model evaluation and selection. It involves partitioning the dataset into *k* equally sized subsets or folds. The model undergoes training and assessment in *k* iterations, with each iteration employing a distinct fold as the validation set while using the remaining folds as the training set. The outcomes of each iteration are then averaged to derive a comprehensive performance estimation. This is the essence of how k-fold cross-validation operates:

1. **Splitting the dataset**: Partition the dataset into *k* non-overlapping folds. Generally, the value of *k* falls within the range of 5 to 10, although it can be adjusted based on factors such as dataset size and complexity.
2. **Training and evaluating the model**: This entails a repetitive process repeated *k* times. Within each iteration, the model is trained using *k – 1* folds and subsequently assessed for its performance on the remaining fold.
3. **Performance metric aggregation**: Calculate the performance metric of interest for each iteration. The metrics from all iterations are then averaged to obtain a robust estimation of the model's performance.
4. **Model selection**: Use the average performance metric to compare and select the best model among different algorithms or hyperparameter configurations.

K-fold cross-validation helps address the limitations of single train-test splits by providing a more reliable estimate of a model's performance. It helps assess how well the model generalizes to unseen data and reduces the risk of overfitting or underfitting. MATLAB provides built-in functions, such as `crossval()` and `cvpartition()`, to facilitate k-fold cross-validation. These functions automate the process and allow for easy implementation and evaluation of models using k-fold cross-validation.

Let's see how we can practically approach a k-fold cross-validation:

1. To understand the use of cross-validation, we employ a NASA dataset derived from a sequence of aerodynamic and acoustic experiments conducted in an anechoic wind tunnel. The dataset comprises several fields, including the following:

   - Frequency, in Hertz (named `FreqH`).
   - Angle of attack, in degrees (named `AngleD`).
   - Chord length, in meters (named `ChLenM`).
   - Free-stream velocity, in meters per second (named `FStVelMs`).
   - Suction side displacement thickness, in meters (named `SucSDTM`).
   - Scaled sound pressure level, in decibels (named `SPLdB`).

   As usual, we start importing the dataset into the MATLAB environment:

```
AirfoilSelfNoise = readtable('AirfoilSelfNoise.xlsx');
```

So, data is now available in the MATLAB workspace. To apply cross-validation in our model, we can use the apps available in the MATLAB environment, for example, the Regression Learner app. This app facilitates a streamlined and efficient process for conducting step-by-step regression analysis. With this app, importing and exploring data, selecting features, defining validation schemes, training models, and evaluating results becomes remarkably straightforward and swift. The app offers automated training functionality, enabling the search for the optimal regression model type. It includes options such as linear regression models, regression trees, Gaussian process regression models, support vector regression, and an ensemble of regression trees. Additionally, the trained model can be exported to the workspace for reuse with new data or to generate MATLAB code for programmatic regression. By leveraging the Regression Learner app, users can save time and effort in performing regression analysis, benefiting from its intuitive interface and powerful features.

To open the app, select the **APPS** tab on the MATLAB toolstrip, and click on the **Regression Learner** icon. The Regression Learner app window will open, as shown in the following figure:

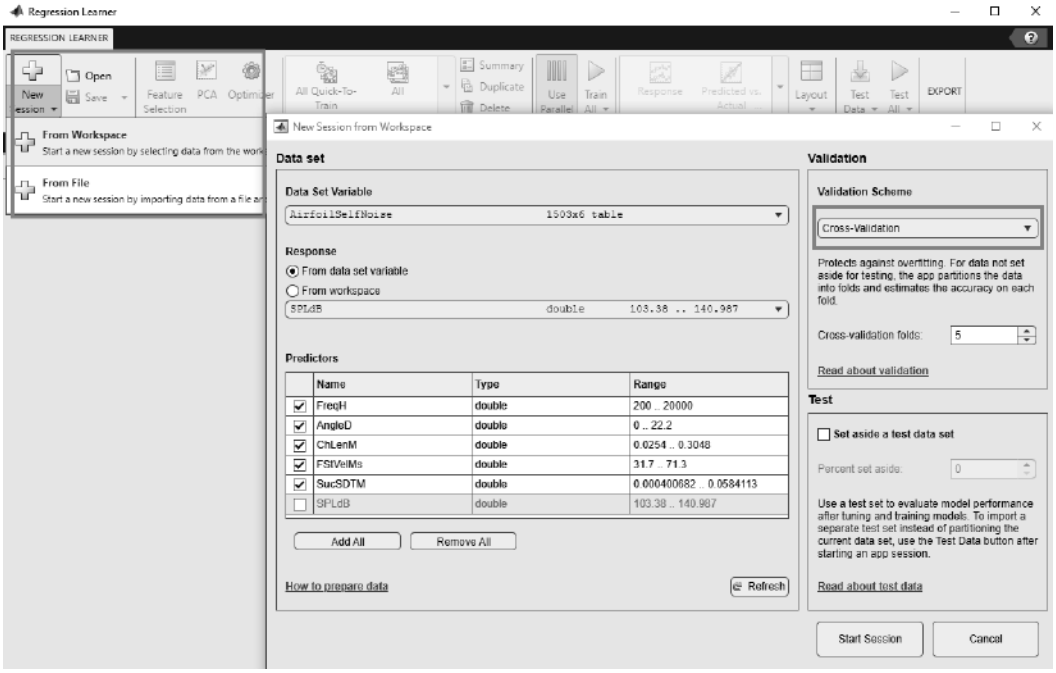


Figure 3.7 – Regression Learner app window

To import data from the MATLAB workspace into the Regression Learner app, follow these steps:

I. Open the Regression Learner app by clicking on the **New Session** button located in the **File** section of the **Regression Learner** tab.

II. The **New Session** dialog box will appear, containing three sections:

i. *Step 1*: Select a table or matrix. In this section, choose the source of your data. You can select a table or a matrix containing the data you want to analyze.

ii. *Step 2*: Select predictors and a response. In this section, specify the variables that will serve as predictors and the variable that will be the response variable. This allows you to define the type of variables involved in the analysis. In this case, we can select **SPLdB** as the response and the other variables as predictors.

iii. *Step 3*: Define a validation method. In this section, you can choose the type of validation method to evaluate the predictive accuracy of the fitted models. Different validation methods, such as cross-validation or holdout validation, can be selected to estimate model performance on new data.

Validation methods play a crucial role in assessing how well the fitted models can make accurate predictions on unseen data. The Regression Learner app provides tools to evaluate and compare models based on their estimated performance, enabling the selection of the best model for the given data.

We will choose **Cross-Validation** as the validation scheme that performs a k-fold cross-validation. We can set the number of folds (`5` by default), then we can press the **Start Session** button.

2. To select the desired model type in the Regression Learner app, follow these steps:

I. Expand the **Model Type** section by clicking on the arrow icon present in the app.

II. A list of available regression models will be displayed, including the following:

- **Linear regression models**
- **Regression trees**
- **SVMs**
- **Gaussian process regression models**
- **Ensembles of trees**

III. To get started, select the **All Quick-To-Train** option. This option allows you to select all the models available for this type of problem, so it will be possible to train the models using default settings quickly.

IV. Click on the **Train** icon to initiate the training process. The app will start training the selected models, and a selection of model types will appear in the **History** section.

V. Once the models finish training, the best RMSE score will be highlighted in a box. This score indicates the model's performance.

VI. To further improve the model's performance, you can try training with all available algorithms. Click on **All** and then click **Train**.

3. By comparing the results in the **History** section, you can observe the RMSE scores for each model. The lower RMSE score represents better performance. In the example provided, the Gaussian process regression model achieved the *lowest RMSE score (RMSE = 1.49)*, while the boosted trees model obtained the *highest RMSE score (RMSE = 6.18)*.

   To assess the improvements obtained, you can compare the predicted versus actual plots for the extreme models. In the model with the lowest RMSE, the data points should closely align with the reference line, indicating a close match between the predicted and actual data.

Overall, the Regression Learner app provides a visual representation of the results, allowing you to easily compare and analyze the performance of different models. Let's see another way to select the data.

Exploring leave-one-out cross-validation

Leave-one-out cross-validation (**LOOCV**) is a specific type of cross-validation technique used for model evaluation. In LOOCV, each data point is sequentially held out as the validation set while the rest of the data is used for training the model. This process is repeated for all data points, and the performance of the model is evaluated by averaging the results across all iterations. The LOOCV technique works in the following way:

1. For each data point in the dataset, do the following:
   - Remove the data point from the training set
   - Train the model using the remaining data points
   - Use the trained model to predict the removed data point
   - Evaluate the performance of the model by comparing the predicted value with the actual value of the removed data point
2. Calculate the performance metric (such as MSE or accuracy) for each iteration.
3. Compute the average performance metric across all iterations to obtain an overall estimation of the model's performance.

LOOCV is particularly useful when the dataset is small or when there is a limited amount of data available. Since each data point serves as a validation set once, LOOCV provides a more reliable estimate of the model's performance compared to other cross-validation techniques. It is worth mentioning that this technique is good for small datasets, as it is difficult to judge its variance error as the data points are almost the same for each fold; it just means one new record in each fold.

In MATLAB, you can use the `cvpartition()` function with the `LeaveOut` option to generate indices for performing LOOCV. These indices can then be used to partition the data for training and validation purposes in a loop.

Now, we will introduce the bootstrap method.

Introducing the bootstrap method

The bootstrap method is a resampling technique used for estimating the sampling distribution of a statistic. This process entails generating multiple bootstrap samples by randomly selecting data points from the original dataset with replacement, from which estimates of the statistic can be obtained. This method allows us to assess the variability and uncertainty associated with the statistic of interest.

The bootstrap algorithm provides the following steps:

1. Start with an original dataset of size *N*.
2. Randomly sample *N* data points from the original dataset with replacement to create a bootstrap sample. This means that each data point in the bootstrap sample is selected independently, and it is possible to select the same data point multiple times.
3. Compute the desired statistic on the bootstrap sample. This could be the mean, median, standard deviation, or any other statistic of interest.
4. Repeat *steps 2* and *3* a number of times equal to *B*, where *B* is the number of bootstrap iterations desired.
5. Collect the computed statistics from each bootstrap sample to create the bootstrap distribution.
6. Calculate the desired confidence intervals or standard errors based on the bootstrap distribution to quantify the uncertainty around the statistic.

The bootstrap method provides a robust approach for estimating the sampling distribution when the underlying distribution of the data is unknown or when analytical methods are not readily available. It can be used for various purposes, such as hypothesis testing, constructing confidence intervals, and assessing model stability.

The bootstrap method holds significance as it enables the estimation of a model's performance attributes, encompassing measures such as central tendency and uncertainty, all without requiring stringent assumptions about the underlying data distribution. This method enhances the reliability and robustness of assessing a model's performance, particularly in scenarios where data is scarce or the data distribution remains poorly defined.

Summary

In this chapter, we have gained valuable insights into performing accurate classification tasks within the MATLAB environment. We began by delving into the realm of decision tree methods, where we familiarized ourselves with key concepts such as nodes, branches, and leaf nodes. By repeatedly dividing records into homogeneous subsets based on the target attribute, we learned how to classify objects into distinct classes effectively. Moreover, we explored the prediction aspect of SVMs, which are particularly effective in solving complex problems with a clear margin of separation between classes. SVMs can handle both linearly separable and non-linearly separable data by transforming the input space into a higher-dimensional feature space.

In the subsequent section, our focus shifted toward conducting precise regression analysis within the MATLAB environment. We commenced by delving into simple linear regression, gaining an understanding of its definition and the process of obtaining ordinary least squares estimation. Additionally, we explored multiple techniques for quantifying the intercept and slope of the linear relationship.

Subsequently, we unearthed the linear regression model builder, a valuable tool for constructing an object encompassing training data, model description, diagnostic information, and fitted coefficients necessary for linear regression. Furthermore, we familiarized ourselves with the correct interpretation of simulation results and grasped the techniques to mitigate the influence of outliers through robust regression.

Then, we understood how to use the tools available in MATLAB to perform an accurate evaluation of the model trained. Finally, we discovered the cross-validation methods to increase the performance of the model, finding the best-performing model.

In the next chapter, we will explore the clustering methodology to find hidden patterns or groupings in a dataset.

4

Clustering Analysis and Dimensionality Reduction

Clustering techniques aim to uncover concealed patterns or groupings within a dataset. These algorithms detect groupings without relying on any predefined labels. Instead, they select clusters based on the similarity between elements. Dimensionality reduction, on the other hand, involves transforming a dataset with numerous variables into one with fewer dimensions while preserving relevant information. Feature selection methods attempt to identify a subset of the original variables, while feature extraction reduces data dimensionality by transforming it into new features. This chapter shows us how to divide data into clusters, or groupings of similar items. We'll also learn how to select features that best represent the set of data.

In this chapter, we will cover the following main topics:

- Understanding clustering – basic concepts and methods
- Understanding hierarchical clustering
- Partitioning-based clustering algorithms with MATLAB
- Grouping data using the similarity measures
- Discovering dimensionality reduction techniques
- Feature selection and feature extraction using MATLAB

Technical requirements

In this chapter, we will introduce basic concepts relating to machine learning. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also need a working knowledge of the MATLAB environment.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `Minerals.xls`
- `PeripheralLocations.xls`
- `YachtHydrodynamics.xlsx`
- `SeedsDataset.xlsx`

Understanding clustering – basic concepts and methods

Clustering is a fundamental concept in data analysis, aiming to identify meaningful groupings or patterns within a dataset. It involves the partitioning of data points into distinct clusters based on their similarity or proximity to each other. In both clustering and classification, our goal is to discover the underlying rules that enable us to assign observations to the correct class. However, clustering differs from classification as it requires identifying a meaningful subdivision of classes as well. In classification, we benefit from the target variable, which provides the classification information in the training set. In contrast, clustering lacks such additional information, necessitating the deduction of classes by analyzing the spatial distribution of the data. Dense areas in the data correspond to groups of similar observations. If we can identify observations that are like each other but distinct from those in another cluster, we can infer that these two clusters represent different conditions. At this stage, three crucial aspects come into play:

- How to measure similarity
- How to find centroids and centers
- How to define a grouping

The notion of distance and the definition of a grouping are the fundamental components that characterize a clustering algorithm.

How to measure similarity

Clustering involves the identification of data groupings based on the measure of proximity between elements. Proximity can refer to either similarity or dissimilarity. Thus, the definition of a data group relies on how we define similarity or dissimilarity. In many approaches, proximity is conceptualized in terms of distance within a multidimensional space. The effectiveness of clustering algorithms heavily depends on the choice of metric and how distance is calculated.

Clustering algorithms group elements based on their mutual distances, where membership in a particular set is determined by the proximity of an element to other members of the same set. Therefore, a set of observations forms a cluster when they tend to be closer to each other compared to observations in other sets. So, what do we mean by similarity and dissimilarity? **Similarity** refers to a numerical measure of how alike two objects are. Higher similarity values indicate greater resemblance between objects. Similarities are typically non-negative and often range from 0 (no similarity) to 1 (complete similarity). In *Figure 4.1*, the difference between intercluster and intracluster distances is shown.

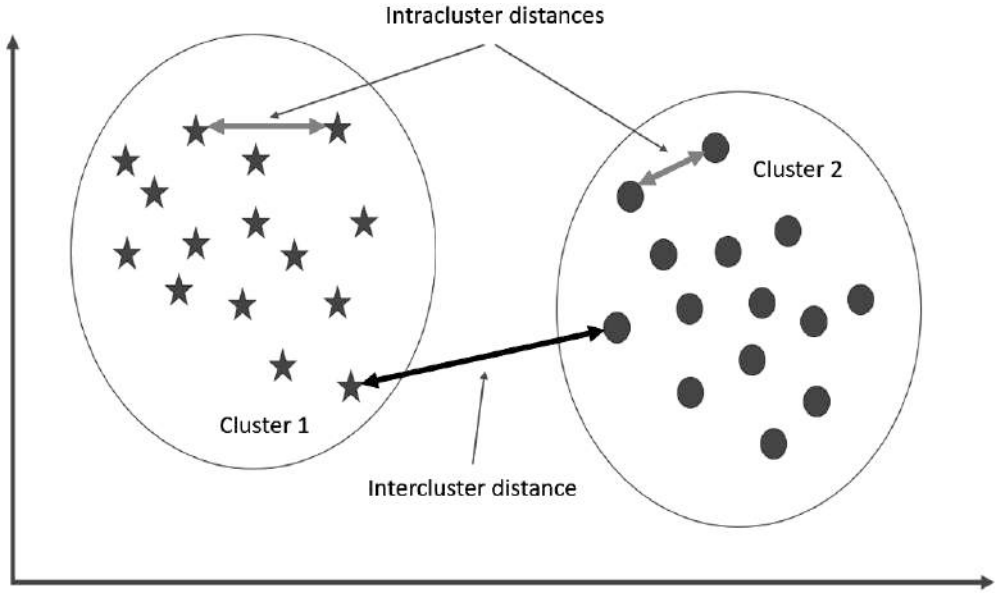


Figure 4.1 – Difference between inter cluster and intra cluster distances

On the contrary, **dissimilarity** represents a numerical measure of how different two objects are. Lower dissimilarity values indicate greater similarity between objects. Dissimilarity is sometimes referred to as distance. Like similarity, dissimilarity values may also fall within the interval *[0,1]*, but it is common for them to range from 0 to ∞.

Dissimilarities between data objects can be quantified using distance metrics. Distances possess specific properties and can be employed to measure dissimilarity. For example, the **Euclidean distance**, denoted as *d*, can be utilized to measure the distance between two points, *x* and *y*, using the following formula:

$$d(x,y) = \sqrt{\sum_{k=1}^{n}(x_i - y_i)^2}$$

In a two-dimensional plane, the Euclidean distance is the shortest distance between two points, represented by a straight line connecting them. This distance is calculated by taking the square root of the sum of the squared differences between the elements of two vectors, as shown in the earlier formula.

However, distance can be measured using various other metrics, such as the Minkowski distance. The **Minkowski distance** metric is a generalization of the Euclidean distance and can be expressed as follows:

$$d(x,y) = \left(\sum_{k=1}^{n}|x_i - y_i|^r\right)^{\frac{1}{r}}$$

Another option is to utilize the **Manhattan distance** metric, which is derived from the Minkowski distance metric by setting $r = 1$. Additionally, there is the **cosine distance**, which incorporates the dot product scaled by the product of the Euclidean distances from the origin. It quantifies the angular distance between two vectors while disregarding their scale. Once we have selected a suitable metric for our specific case, we can proceed.

Choosing an appropriate similarity metric for clustering holds great significance as it directly influences the quality and comprehensibility of your clustering outcomes. The selection of the metric should be in harmony with both the inherent traits of your data and the objectives of your clustering analysis. The default distance used in most clustering tools is Euclidean distance. However, this may be inappropriate in the presence of very noisy data or with a non-Gaussian distribution. In these situations, a robust alternative is the Manhattan distance, although it is important to keep in mind that robustness involves a loss of information, at least to some extent. In any case, you can choose between different dissimilarity measures based on the type of data and research objectives.

So far, we have examined different formulas for calculating distances between two objects. However, what if the objects under analysis are nominal instead of numerical? For nominal data, which can be represented as simple sequences or strings, various distance measures can be employed. One possible distance between two strings is determined by counting the number of symbols that differ between the strings.

How to find centroids and centers

Centroids and centers are essential concepts in clustering that represent the central points or locations within each cluster. The **centroid** is a point within the feature space that represents the average position of all the data points belonging to the associated cluster. It serves as a type of center of gravity for the cluster and, typically, does not coincide with any specific data point in the dataset. For example, in **k-means clustering**, a centroid is the mean or average position of all the data points within a cluster. It serves as a representative point that minimizes the sum of squared distances between the data points and the centroid. The centroid is computed by taking the average of the feature values across all data points within the cluster. During the iterative process of k-means, data points are assigned to the cluster whose centroid they are closest to, and the centroids are updated accordingly until convergence.

The term *centers* is often used in other clustering algorithms, such as hierarchical clustering or density-based clustering. The centers can represent central points within clusters, but the computation method might differ depending on the algorithm. In **hierarchical clustering**, centers can be computed as the mean, median, or another representative point within each cluster. In **density-based clustering**, the center of a cluster can be defined as the data point with the highest density or as the medoid, which is the most centrally located point.

Both centroids and centers provide a way to summarize and represent clusters in clustering analysis. They offer insights into the characteristic properties of each cluster and can be used to classify new data points based on their proximity to these central points.

How to define a grouping

A cluster is a set of objects that are like each other but are dissimilar to objects contained in other clusters; a cluster can therefore be defined in terms of internal cohesion (homogeneity) and external cohesion (separation). The goal is to group – or segment – collections of objects into subsets, called clusters, such that objects in the same cluster have similar characteristics to each other, unlike elements assigned to different clusters. In this sense, the distance between points of the same cluster must be minimized while the distance between points belonging to different clusters must be maximized.

In cluster analysis, the definition of a measure that quantifies the degree of similarity between objects is crucial (see the measures in *Figure 4.1*). The choice of this measurement greatly influences the formation of one object partition over another. When dealing with numeric data, the similarity measure is referred to as a distance function. In this case, the goal is to minimize the distance function because objects that are closer together exhibit similar characteristics and belong to the same cluster. On the other hand, when working with textual data, the similarity measure is used instead. In these cases, the aim is to maximize similarity since texts that are more similar are closer together and belong to the same cluster.

A visual representation can help reveal the cohesion properties of clusters without explicitly and rigorously defining them. It is important to note that there is no universal definition that applies to every situation. Attempts to precisely quantify homogeneity and separation using explicit numerical indices have resulted in numerous and diverse criteria.

Once a distance measure has been selected, the next step is to determine how to form groups or clusters. There are two main families of clustering algorithms:

- **Hierarchical clustering**: This approach constructs a hierarchical structure or taxonomy of the data. It involves creating a tree-like structure, often referred to as a **dendrogram**, which represents the relationships between the data points. Hierarchical clustering can be agglomerative, starting with individual data points and iteratively merging them into clusters, or divisive, beginning with one large cluster and recursively splitting it into smaller clusters.
- **Partitioning clustering**: In this approach, the data space is divided into non-overlapping sub-zones or partitions. Each data point belongs to exactly one partition, and the union of all partitions covers the entire space. Partitioning algorithms aim to optimize a specific criterion, such as minimizing the within-cluster variance or maximizing the separation between clusters. Popular partitioning algorithms include k-means, where data points are assigned to the cluster with the nearest centroid, and **Gaussian mixture models** (**GMMs**), which represent clusters as a mixture of Gaussian distributions.

Both hierarchical clustering and partitioning clustering methods have their own advantages and applications. The choice between the two depends on the nature of the data, the desired interpretation of the results, and the specific goals of the clustering analysis.

Understanding hierarchical clustering

Hierarchical clustering is a method of clustering that creates a hierarchy or tree-like structure of clusters. It iteratively merges or splits clusters based on the similarity or dissimilarity between data points. The resulting structure is often represented as a dendrogram, which visualizes the relationships and similarities among the data points.

There are two main types of hierarchical clustering:

- **Agglomerative hierarchical clustering**: This starts with each data point considered as an individual cluster and progressively merges similar clusters until all data points belong to a single cluster. At the beginning, each data point is treated as a separate cluster, and in each iteration, the two most similar clusters are merged into a larger cluster. This process continues until all data points are in one cluster. The merging process is guided by a distance or similarity measure, such as a Euclidean distance or correlation.
- **Divisive hierarchical clustering**: This starts with all data points in a single cluster and recursively divides the cluster into smaller subclusters until each data point is in its own cluster. In each iteration, a cluster is split into two or more subclusters based on a dissimilarity measure. This process continues until each data point forms its own cluster or until a stopping criterion is met.

Hierarchical clustering has several advantages. It does not require specifying the number of clusters in advance, as the desired number of clusters can be determined by cutting the dendrogram at a particular level. It also provides a visual representation of the clustering structure, allowing for interpretation and exploration of the data. However, hierarchical clustering can be computationally intensive for large datasets, and the choice of distance or similarity measure and the method of merging or splitting clusters can influence the results. Overall, hierarchical clustering is a flexible and widely used technique for exploring and understanding the underlying structure of a dataset.

In MATLAB, hierarchical clustering is performed using several functions to create a cluster tree or dendrogram. The process involves grouping the data and creating a multilevel hierarchy where clusters at one level are joined as clusters at the next level. The *Statistics and Machine Learning Toolbox* provides the necessary functions to perform agglomerative hierarchical clustering.

The following functions are commonly used in the process:

- `pdist`: This function calculates pairwise distances between data points based on a chosen distance metric. It takes the data matrix as input and returns a distance matrix.
- `linkage`: The `linkage` function is used to compute the linkage matrix, which represents the distances between clusters. It takes the distance matrix as input and performs the linkage calculation based on a specified method, such as `single`, `complete`, or `average`. `single` linkage, often referred to as the nearest neighbor method, employs the minimum distance between objects in the two clusters. `complete` linkage, also known as the farthest neighbor method, utilizes the maximum distance between objects in the two clusters. Finally, `average` linkage computes the mean distance between all pairs of objects in any two clusters.
- `cluster`: The `cluster` function is responsible for forming the clusters based on the linkage matrix. It takes the linkage matrix and a cutoff parameter as input and returns the cluster assignments for each data point.
- `dendrogram`: Finally, the `dendrogram` function is used to visualize the cluster tree or dendrogram. It takes the linkage matrix as input and plots the dendrogram, showing the merging of clusters at each level.

By utilizing these functions in sequence, the `clusterdata` function in MATLAB performs agglomerative hierarchical clustering and generates the dendrogram. Understanding the sequence of function calls can be beneficial for comprehending the entire process and the flow of the hierarchical clustering algorithm.

Let's explore how clustering can be performed in MATLAB by using the provided functions. Clustering involves identifying groups within a dataset based on proximity measures, which can refer to similarity or dissimilarity. Here's an example using MATLAB. In MATLAB, the `pdist` function is used to compute the distance between every pair of objects in a dataset. If the dataset has *k* objects, there are *k*(k – 1)/2* pairs in total. The `pdist()` function calculates the Euclidean distance by default for a *k-by-p* data matrix, where rows correspond to observations and columns correspond to variables. It returns a row vector of length *k(k – 1)/2*, representing pairs of observations in the source matrix. The distances are arranged in the order *(2,1), (3,1), ..., (k,1), (3,2), ..., (k,2), ..., (k,k–1))*. To obtain the distance matrix, you can use the `squareform()` function, which we'll cover later.

By default, `pdist()` calculates the Euclidean distance, but there are various distance metrics available. You can specify a different metric by selecting one of the options provided in the function's syntax. Some available choices include `euclidean`, `squaredeuclidean`, `seuclidean`, `cityblock`, `minkowski`, `chebychev`, `mahalanobis`, `cosine`, `correlation`, `spearman`, `hamming`, and `jaccard`. Additionally, you can create a custom distance function.

Let's then analyze the MATLAB code line by line.

1. As an example, let's consider a matrix with six points in a Cartesian plane, represented by pairs of coordinates (*x*, *y*):

```
DataPoints = [100 100;90 90;10 10;10 20;90 70;50 50];
```

2. Now, let's apply the `pdist` function to calculate the distances between pairs of points. The `pdist` function will return this distance information in a vector, where each element represents the distance between a pair of points:

```
DistanceCalc = pdist(DataPoints);
```

   To gain a better understanding of the relationships among the distances, we can reformat the distance vector into a matrix using the `squareform()` function. This matrix will have element *i,j* representing the distance between point *i* and point *j*. In the resulting matrix, element *1,1* will be the distance between point 1 and itself, which is 0. Element *1,2* will represent the distance between point 1 and point 2, and so on:

```
DistanceMatrix = squareform(DistanceCalc);
```

   The following matrix was printed:

```
         0   14.1421  127.2792  120.4159   31.6228   70.7107
   14.1421         0  113.1371  106.3015   20.0000   56.5685
  127.2792  113.1371         0   10.0000  100.0000   56.5685
  120.4159  106.3015   10.0000         0   94.3398   50.0000
   31.6228   20.0000  100.0000   94.3398         0   44.7214
   70.7107   56.5685   56.5685   50.0000   44.7214         0
```

   It is worth noting that the distance matrix is symmetric, indicating that the distance between point *i* and point *j* is the same as the distance between point *j* and point *i*. In certain cases, it is beneficial to normalize the values in the dataset before calculating the distance information. This normalization is important because variables in raw data can be measured on different scales. These variations in measurement scales can distort proximity calculations. To address this, the `zscore()` function can be used to standardize all values in the dataset to a common scale.

3. As previously mentioned, the concepts of distance and how to define groups are the key components of a clustering algorithm. After calculating the proximity between objects in the dataset, the next step is to determine how these objects should be grouped into clusters. This

is where the `linkage` function comes into play. Utilizing the distance measure generated by the `pdist()` function, the `linkage` function links pairs of objects that are close together, forming binary clusters. This process continues as the binary clusters are merged with other objects, creating larger clusters, until all objects in the original dataset are linked together in a hierarchical tree structure:

```
GroupsMatrix = linkage(DistanceCalc)
GroupsMatrix =
3.0000    4.0000   10.0000
1.0000    2.0000   14.1421
5.0000    8.0000   20.0000
6.0000    9.0000   44.7214
7.0000   10.0000   50.0000
```

With the usage of the `linkage` function, we have completed a significant portion of the task. This function has identified potential groupings based on the previously calculated distance measurements. To gain a deeper understanding of how the function operates, let's examine the results in `GroupsMatrix`.

In `GroupsMatrix`, each row represents a newly formed cluster. By analyzing this diagram we can understand how the points are identified to group them into different groups. The first two columns indicate the points that have been linked, particularly in the earlier stages, while the subsequent columns denote the newly created clusters. It is important to note that hierarchical clustering, of the agglomerative type, begins with small clusters and proceeds to incorporate additional elements, generating larger clusters. The third column of the matrix presents the distance between these points, providing further insights into the clustering process. In *Figure 4.2*, we can see the dendrogram of the clustering method. By analyzing this diagram, we can understand how the points are treated to group them into different groups.

The `linkage` function is responsible for utilizing the distances calculated by the `pdist()` function to determine the order in which clusters are formed. Additionally, this function can calculate the distances between newly merged clusters. By default, the `linkage` function employs the single `linkage` method. However, there are several other methods available for use, including `average`, `centroid`, `complete`, `median`, `single`, `ward`, and `weighted`.

4. Finally, to visualize the hierarchical binary cluster tree, we can utilize the `dendrogram()` function. This function generates a dendrogram plot, which consists of U-shaped lines connecting data points in a hierarchical tree structure. The height of each U-shaped line represents the distance between the two connected data points:

```
dendrogram(GroupsMatrix)
```

In *Figure 4.2*, we can see a dendrogram of a hierarchical cluster obtained from a series of points on the Cartesian plane.

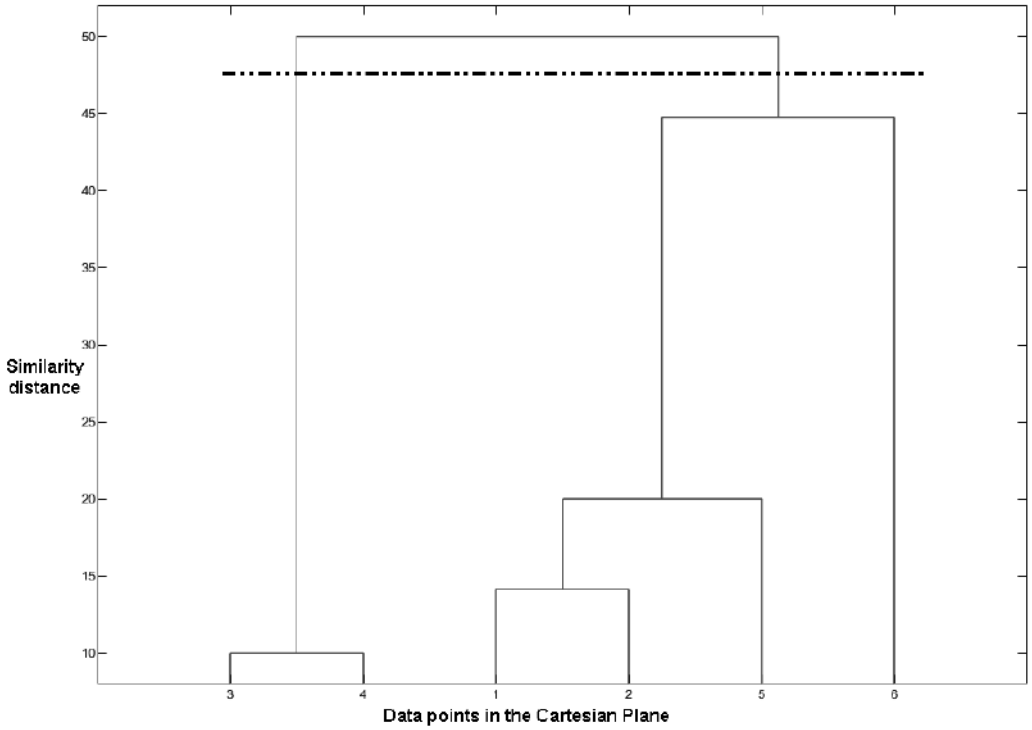


Figure 4.2 – Dendrogram of a hierarchical cluster

A dendrogram is a graphical representation in the form of a branching diagram that illustrates the similarities among a group of entities. The horizontal axis of the dendrogram represents the elements being analyzed, while the vertical axis indicates the level of distance at which the fusion of two elements occurs. The strength of the relationships between two elements is depicted by the distance between the element's corresponding vertical lines and the *x* axis.

To understand the relationship between two elements in a dendrogram, we can trace a path from one element to another by following the tree diagrams and selecting the shortest path. The distance from the starting point to the outermost horizontal line crossed by the path reflects the degree of similarity between the two elements.

After building the dendrogram, we make a horizontal cut of the structure. All resulting subbranches below the horizontal cut represent a single cluster at the top level of the system and determine the corresponding cluster membership for each data sample (*Figure 4.2*).

5. Finally, we can verify how hierarchical clustering is able to group objects. In a dendrogram, the disparity between branches is reflected by their respective heights: the taller the branch, the greater the dissimilarity. This vertical measurement is referred to as the cophenetic distance, representing the dissimilarity between the two objects that form the cluster. To evaluate and compare the cophenetic distances between two objects, the `cophenet()` function can be employed. Verifying

the effectiveness of the dissimilarity measurement can be accomplished by comparing the outcomes of clustering the same dataset using various distance calculation methods or clustering algorithms. This approach allows us to assess and contrast the performance of different techniques. To begin, let's apply this evaluation to the calculations that have just been performed:

```
DistancesCheck = cophenet(GroupsMatrix, DistanceCalc)
```

The following results are returned:

```
DistancesCheck =
    0.8096
```

The validity of a cluster tree is assessed based on how well it reflects the distances between objects as measured by the chosen distance metric. The `cophenet()` function calculates the cophenetic correlation coefficient for the hierarchical cluster tree, which indicates the strength of the correlation between object distances and the linkage of the objects within the tree. A higher cophenetic correlation coefficient (closer to *1*) indicates more accurate hierarchical clustering.

To enhance the model's performance, we can recompute the distances between points using the `pdist()` function, utilizing a different metric this time, while working with the same dataset:

```
NewDistanceCalc = pdist(DataPoints, 'cosine');
```

Now, utilizing the new distance measure generated by the `pdist()` function with the cosine metric, we will form clusters by linking objects that are in close proximity. However, this time we will employ a different algorithm for calculating the distance between clusters. Specifically, we will use the weighted method, which computes the weighted average distance. By doing so, we can establish clusters based on the closeness of objects, considering their respective weights:

```
NewGroupsMatrix = linkage(NewDistanceCalc,'weighted');
```

6. Lastly, we can employ the `cophenet()` function to evaluate the clustering solution:

```
NewDistancesCheck = cophenet(NewGroupsMatrix, NewDistanceCalc)
```

The following results are printed:

```
NewDistancesCheck =
    0.9302
```

The outcome demonstrates an enhancement in the performance of hierarchical clustering by utilizing a different distance metric and linkage method.

Hierarchical clustering is a versatile clustering method with a wide range of applications across various domains. Some of the typical areas where hierarchical clustering is applied include **image processing**, **text analysis**, **market research** and **customer segmentation**, and **ecology and environmental science**.

Having introduced the basic concepts of clustering, we can now move on to a practical example to discover how to use the tools available in the MATLAB environment to correctly perform a partitioning clustering analysis.

Partitioning-based clustering algorithms with MATLAB

Partitioning-based clustering is a type of clustering algorithm that aims to divide a dataset into distinct groups or partitions. In this approach, each data point is assigned to exactly one cluster, and the goal is to minimize the intra-cluster distance while maximizing the inter-cluster distance. The most popular partitioning-based clustering algorithms include k-medoids, fuzzy c-means, and hierarchical k-means. These algorithms vary in their approach and objectives, but they all aim to partition the data into well-separated clusters based on some distance or similarity measure.

Introducing the k-means algorithm

One of the most well-known partitioning-based clustering algorithms is k-means. In k-means clustering, the algorithm attempts to partition the data into *k* clusters, where *k* is a predefined number specified by the user. The algorithm iteratively assigns data points to the nearest cluster centroid and recalculates the centroid positions until convergence is achieved. The result is a set of *k* clusters, each represented by its centroid. The process involves iteratively relocating data instances by transferring them from one cluster to another, beginning with an initial partitioning.

The k-means algorithm, developed by James MacQueen in 1967 (MacQueen, J. (1967) *Some methods for classification and analysis of multivariate observations*, Proceedings of the Fifth Berkeley Symposium On Mathematical Statistics and Probabilities, 1, 281-296.), is a clustering algorithm that partitions a group of objects into *k* clusters based on their attributes. It is a variation of the **expectation-maximization** (**EM**) algorithm, with the objective of determining *k* data groups generated by Gaussian distributions.

The algorithm follows an iterative procedure:

1. Choose the number of clusters, *k*. Choosing the right value for *k*, , is a crucial step as it directly impacts the quality of your clustering outcomes. Numerous techniques are available to ascertain the optimal *k*, including the elbow method, silhouette score, cross-validation, and hierarchical clustering, among others.
2. Initialize *k* partitions and assign each data point randomly or using heuristic information.
3. Calculate the centroid for each cluster, which is the mean of all the points within the cluster.
4. Calculate the distance between each data point and each cluster centroid.
5. Create a new partition by assigning each data point to the cluster with the closest centroid.
6. Recalculate the centroids for the new clusters.
7. Repeat *steps 4* to *6* until the algorithm converges.

The goal is to determine the position of *k* centroids, with each centroid representing a cluster. The initial positions of the centroids significantly impact the results, and it is beneficial to place them as far apart as possible. Each object is then associated with the nearest centroid, resulting in an initial grouping. In subsequent iterations, new centroids are recalculated as the cluster barycenter based on the previous iteration's results. Data points are reassigned to the new closest centroids. This process continues until the centroids no longer move, indicating convergence. We can see how *Figure 4.4* illustrates the positions of *k* centroids in the data distribution.

Using k-means in MATLAB

In MATLAB, the `kmeans()` function is used to perform k-means clustering. This function partitions the data into *k* mutually exclusive clusters and returns the index of the cluster to which each object is assigned. The clusters are defined by the objects and their centroids. The centroid of each cluster is the point that minimizes the sum of distances from all objects in that cluster. The calculation of cluster centroids varies depending on the chosen distance measure, aiming to minimize the sum of the specified measure. Different distance measures and methods for minimizing distances can be specified as input parameters to the `kmeans()` function. Here is a list summarizing the available distance measures:

- `sqeuclidean`: **Squared Euclidean** distance (default). Each centroid is the mean of the points in that cluster.
- `cityblock`: Sum of absolute differences. Each centroid is the component-wise median of the points in that cluster.
- `cosine`: 1 minus the cosine of the included angle between points. Each centroid is the mean of the points in that cluster after normalizing those points to the unit of the Euclidean length.

- `correlation`: 1 minus the sample correlation between points. Each centroid is the component-wise mean of the points in that cluster after centering and normalizing those points with a mean of 0 and a standard deviation of 1.
- `hamming`: Suitable for binary data only. This measures the proportion of differing bits. Each centroid is the component-wise median of the points in that cluster.

By default, the `kmeans()` function uses the k-means++ algorithm for cluster center initialization and the squared Euclidean metric to determine distances.

K-means++ enhances the k-means algorithm by introducing a more sophisticated approach to initializing cluster centroids. The initialization procedure in k-means++ comprises the following steps:

1. Begin by randomly selecting a single data point as the initial centroid.
2. For each subsequent centroid (up to *k*), pick the next centroid from the data points with a probability that is proportional to the square of the distance between each data point and the nearest existing centroid. This probabilistic selection ensures that the initial centroids are distributed in a manner that optimally spaces them apart and reduces sensitivity to the initial centroid choices.
3. After initializing the centroids using the k-means++ method, the algorithm proceeds in the same fashion as standard k-means. It iteratively assigns data points to the closest centroids and updates the centroids until convergence is achieved.

K-means++ exhibits a tendency for swifter and more consistent convergence due to its enhanced initialization method, which diminishes the likelihood of it becoming trapped in local optima. As a consequence, it often yields superior clustering quality by virtue of the well-dispersed initial centroids, ultimately resulting in more advantageous cluster assignments. Let's now analyze a practical application of this algorithm:

1. To illustrate the usage of the `kmeans()` function, let's consider a dataset containing measurements of the specific weight and hardness (Mohs scale) of minerals extracted from different quarries. These measurements are stored in an `xls` file named `Minerals.xls`. We can start by importing the data into the MATLAB workspace:

   ```
   InputData = readmatrix('Minerals.xls');
   ```

   So, let's look at the imported data by plotting a simple scatter plot:

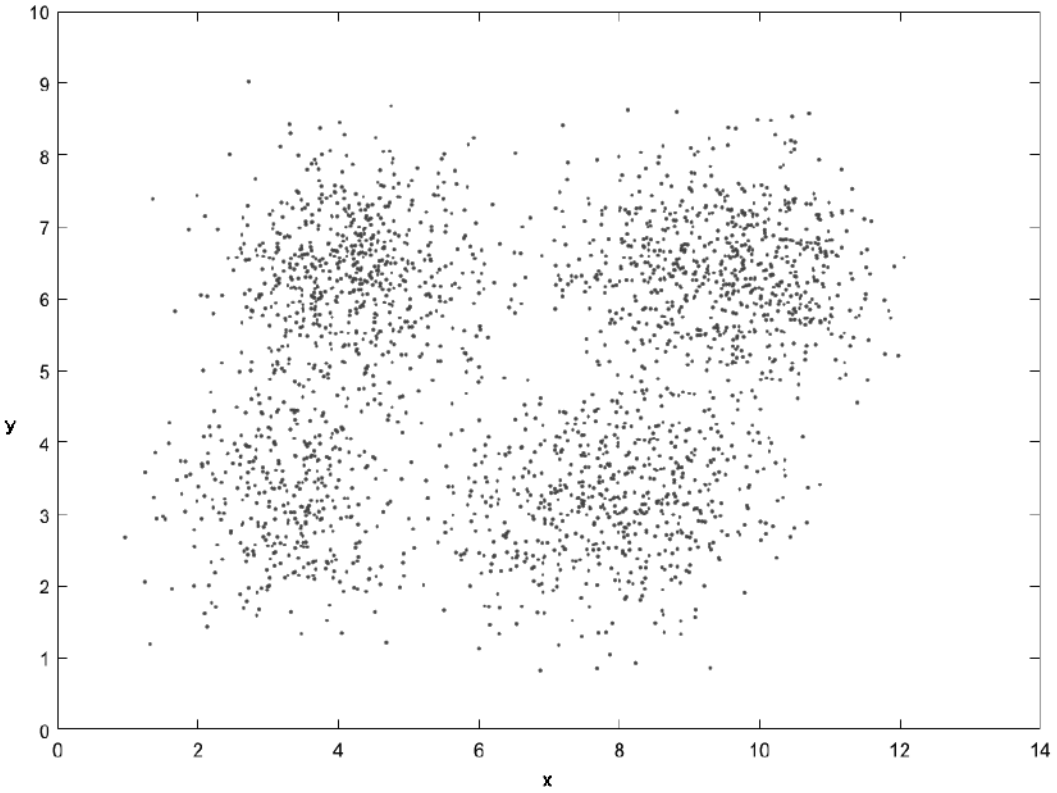


Figure 4.3 – Distribution of dataset (x is the weight and y is the hardness (Mohs scale) of minerals)

By examining *Figure 4.3*, it appears that the data points are concentrated in four distinct regions, each characterized by different values of the two variables (*x,y*). This observation suggests that a cluster analysis with a fixed value of $k = 4$ should be conducted. To ensure reproducibility and obtain consistent results, we can set the following parameters accordingly.

2. We can set the seed to permit the reproducibility of the experiment. The `rgn()` function is responsible for controlling the generation of random numbers in MATLAB. By setting the seed using `rgn()`, the `rand()`, `randi()`, and `randn()` functions will produce a predictable sequence of numbers:

```
rng(5);
```

Now, we can proceed with the application of the `kmeans()` function for our analysis:

```
[IdCluster,Centroid] = kmeans(InputData,4);
```

Two variables have been generated: `IdCluster` and `Centroid`. `IdCluster` is a vector that stores the predicted cluster indices for each observation in the `InputData`. The centroid is a

4-by-2 matrix that represents the final centroid locations for the four clusters. By default, the `kmeans()` function utilizes the k-means++ algorithm for centroid initialization and employs the squared Euclidean distance measure.

Now that we have assigned every data point in the original dataset to one of the four clusters, we can visualize the clusters in a scatter plot:

```
gscatter(InputData(:,1), InputData(:,2), IdClus-
ter,'bgrm','x*o^')
hold on
plot(Centroid(:,1),Centroid(:,2),'x','LineWidth',4,'Mark-
erEdgeColor','k','MarkerSize',25)
```

The plot shown in *Figure 4.4* is printed:

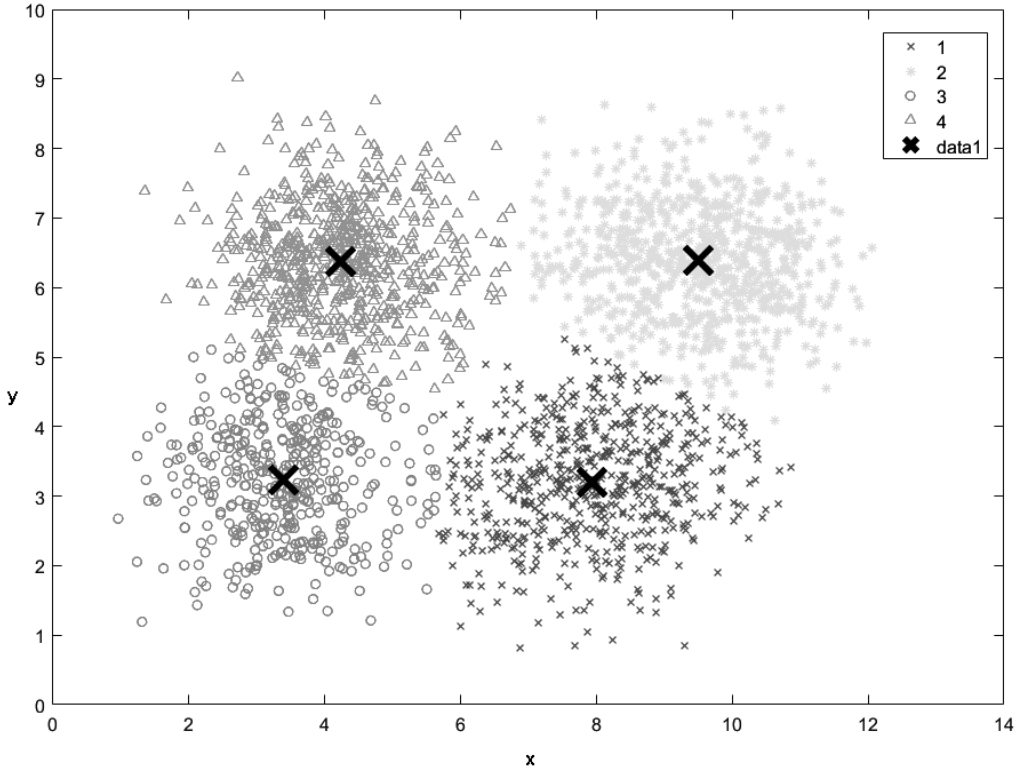


Figure 4.4 – Scatter plot of clusters (x is the weight and y is the hardness (Mohs scale) of minerals)

Upon analyzing *Figure 4.4*, it is evident that the `kmeans()` function has successfully identified distinct clusters with clear separation. Each cluster is automatically visualized using a different color, making it easy to identify them, especially at the boundaries where the data points may overlap. Additionally, the use of different markers in the scatter plot aids in distinguishing the data points.

To highlight the position of each centroid, we have specifically set the marker as `x` and adjusted its appearance by setting the line thickness to `4`, marker color to `black`, and marker size to `25`. The centroids serve as the center of mass for their respective clusters and minimize the sum of squared distances from the other data points, as mentioned earlier.

Upon further examination of the figure, it becomes apparent that cluster boundaries are not always well defined, particularly in this type of chart. Near the boundaries, there are data points that merge together, making it challenging to determine which cluster they belong to.

3. To gain a deeper understanding of the separation and boundaries of the clusters, we can utilize a **silhouette plot**, which provides insights into how well the clusters are separated. The silhouette plot measures the proximity of each data point in one cluster to the data points in the neighboring clusters. The `silhouette()` function is specifically designed to generate cluster silhouettes based on the provided cluster indices from the `kmeans()` function. This function takes a data matrix as input, where each row represents a data point and each column represents its coordinates. The cluster indices can be categorical variables, numeric vectors, character matrices, or cell arrays of character vectors, where each element represents the cluster assignment for a data point.

   When using the `silhouette()` function, any `NaN`or empty character vectors in the cluster indices are treated as missing values, and the corresponding rows in the data matrix are ignored. By default, the `silhouette()` function calculates the squared Euclidean distance between the data points in the input matrix. Using the silhouette plot, we can gain a better understanding of the separation and proximity of the data points within and between clusters, providing valuable insights into the quality of the clustering results:

```
silhouette(InputData, IdCluster)
```

   The following figure shows a silhouette plot for `kmeans` clustering.

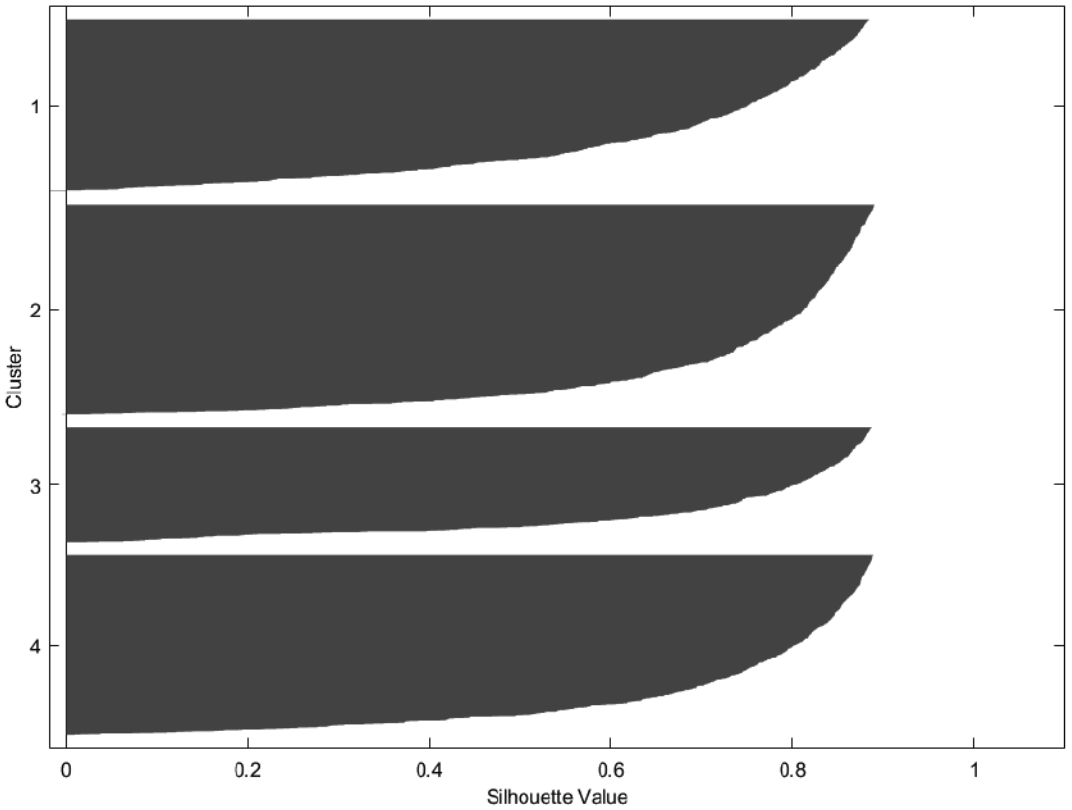


Figure 4.5 – Silhouette plot for k-means clustering

In *Figure 4.5*, the *x* axis represents the silhouette values, which range from `+1` to `-1`. A silhouette value of `+1` indicates that a data point is significantly distant from neighboring clusters, while a value of `0` suggests that the data point does not clearly belong to any specific cluster. A silhouette value of `-1` implies that the data point is likely assigned to the wrong cluster. The silhouette plot returns these values as its first output.

The silhouette plot is useful for assessing the separation distance between the resulting clusters. It provides a visual representation of how close each data point in one cluster is to the points in neighboring clusters. By examining the silhouette plot, we can evaluate the clustering solution and determine whether the chosen parameters, such as the number of clusters, are appropriate.

In a successful clustering solution, we expect to see high silhouette values, indicating that the points are correctly assigned to their respective clusters and have minimal connections to neighboring clusters. If a majority of the data points have high silhouette values, it indicates a good clustering solution. However, if many points have low or negative silhouette values, it suggests that the clustering solution needs to be reviewed.

Based on the analysis of *Figure 4.5*, we can determine whether our assumption of choosing $k = 4$ for the k-means algorithm has produced favorable results. In *Figure 4.5*, there are no clusters with below-average silhouette scores, indicating a good separation between the clusters. Additionally, the silhouette plots do not exhibit significant fluctuations in size. The uniform thickness of the silhouette plot indicates that the clusters are of similar sizes. These observations validate our choice of $k = 4$ for the number of clusters and confirm that it is an appropriate selection.

After analyzing in detail how to easily perform a k-means clustering analysis in MATLAB, we can now see a new clustering methodology: the k-medoids algorithm.

Grouping data using the similarity measures

The k-medoids algorithm is a variation of the k-means algorithm that uses medoids (actual data points) as representatives of each cluster instead of centroids. Unlike the k-means algorithm, which calculates the mean of the data points within each cluster, the k-medoids algorithm selects the most centrally located data point within each cluster as the medoid. This makes k-medoids more robust to outliers and suitable for data with non-Euclidean distances.

Here are some key differences between k-medoids and k-means:

- **Representative points**: In k-medoids, the representatives of each cluster are actual data points from the dataset (medoids), while in k-means, the representatives are the centroids, which are calculated as the mean of the data points.
- **Distance measure**: The distance measure used in k-means is typically the Euclidean distance. On the other hand, k-medoids can handle various distance measures, including non-Euclidean distances. This flexibility allows k-medoids to work with different types of data, such as categorical or ordinal variables.
- **Robustness to outliers**: The k-medoids algorithm is generally more robust to outliers than k-means. Outliers can significantly affect the centroid calculation in k-means, pulling the centroid towards them. In k-medoids, the medoids are actual data points, so the impact of outliers is limited to the specific cluster that contains them.
- **Computational complexity**: K-means has a lower computational complexity compared to k-medoids. The selection of medoids in k-medoids requires the evaluation of pairwise distances between all data points, making it computationally more expensive, especially for large datasets.
- **Cluster shape**: The k-means algorithm tends to produce spherical-shaped clusters due to the use of Euclidean distance and the calculation of mean centroids. K-medoids, on the other hand, can produce clusters of arbitrary shapes, as the medoids can be any data point within the cluster.

When deciding between k-means and k-medoids, consider the nature of your data, the desired robustness to outliers, the availability of appropriate distance measures, and the computational resources. If your data contains outliers or non-Euclidean distances are more appropriate, k-medoids may be a better choice. Otherwise, if the data is well behaved and the Euclidean distance is suitable, k-means can provide efficient clustering results.

Applying k-medoids in MATLAB

In MATLAB, the `kmedoids()` function is used to perform k-medoids clustering. It partitions the observations in a matrix into *k* clusters and returns a vector containing the cluster indices for each observation. The input matrix should have rows corresponding to points and columns corresponding to variables. Similar to the `kmeans()` function, the `kmedoids()` function uses the squared Euclidean distance measure and the k-means++ algorithm for selecting initial cluster medoid positions by default.

Now, let's see how the `kmedoids()` method is applied in a practical example.

Suppose a large distribution company wants to optimize the positioning of its offices to improve the efficiency of transferring goods from sorting hubs to peripheral locations. The company already has the geographic coordinates of its peripheral locations and wants to determine the optimal positions for the sorting hubs. The geographic coordinates have been transformed into relative coordinates for compatibility reasons with the `kmedoids()` function. This information is stored in a file named `PeripheralLocations.xls`:

1. Let's start by importing this data into the MATLAB workspace:

   ```
   DataKMedoids = readmatrix('PeripheralLocations.xls');
   ```

   We now have the recently imported MATLAB matrix called `DataKMedoids` with dimensions of 2,650x2.

   Let's now look at the data by drawing a scatter plot:

   ```
   gscatter(DataKMedoids (:,1), DataKMedoids (:,2));
   ```

   The following plot displays the distribution of the dataset being imported.

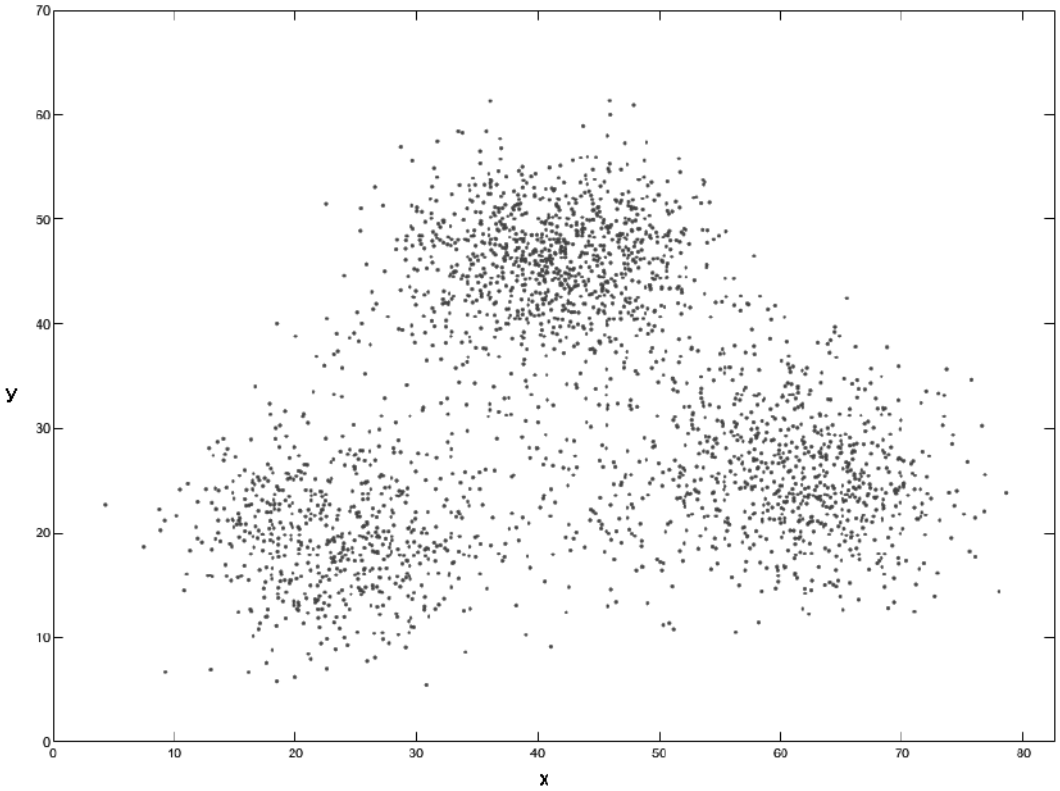


Figure 4.6 – Locations of the peripheral offices (geographic coordinates of the office: x is the latitude and y is the longitude)

Based on an initial visual analysis, three main regions can be identified where the company's peripheral offices are located. This implies the need to designate three current offices as hubs for distributing goods to these peripheral locations. To determine the position of these future hubs and the associated sites for each, a clustering analysis will be conducted.

2. It is evident that each hub's location should serve as the center of its respective cluster. As these positions are concrete values and must correspond to existing offices, utilizing the k-medoids method appears to be a logical decision. K-means and k-medoids differ in their centroid determination and sensitivity to outliers. K-means minimizes the sum of squared distances between data points and cluster centroids, using means as centroids and proving efficient for spherical, equally sized clusters. However, it is sensitive to outliers and influenced by initialization. In contrast, k-medoids minimizes the sum of dissimilarities, using the most centrally located data point (medoid) as the cluster center. This makes k-medoids more robust to outliers and

noise, offering better performance for non-spherical clusters or those with uneven sizes. K-medoids is less sensitive to initialization, but its computational cost, involving dissimilarity calculations and medoid selection, can be higher than k-means, especially for large datasets.

```
[IdCluster,Kmedoid,SumDist,Dist,IdClKm,info] =
kmedoids(DataKMedoids,3);
```

In the previous code, the variables have the following meaning:

- `IdCluster` contains cluster indices of each observation
- `Kmedoid` contains the *k* cluster medoid locations
- `SumDist` contains the within-cluster sums of point-to-medoid distances
- `Dist` contains distances from each point to every medoid
- `IdClKm` contains cluster indices of each medoid
- `info` contains information about the options used by the algorithm when executed

Let's take a look at the information returned by the `kmedoids()` function:

```
>> info
info =
  struct with fields:
        algorithm: 'pam'
            start: 'plus'
         distance: 'sqeuclidean'
       iterations: 3
    bestReplicate: 1
```

The command provides information on the algorithm type, metrics used, and the number of iterations to achieve the best performance. The algorithm used is called **PAM**, which stands for **partitioning around medoids**, and it is a classic approach for solving the k-medoids problem. This algorithm closely resembles k-means; however, it employs medoids, which are the data points positioned at the center of clusters, as cluster representatives rather than centroids. The primary objective of PAM is to minimize the total dissimilarity between data points and their corresponding medoids.

The second parameter denotes the method used to determine the initial positions of the cluster medoids. The chosen metric is the squared Euclidean distance. The command also displays the number of iterations and identifies the best iteration.

To visualize the result, we can generate plots of the clusters and their respective medoids:

```
gscatter(DataKMedoids (:,1), DataKMedoids (:,2), IdClus-
ter,'bgr','xo^')
hold on
plot(Kmedoid(:,1),Kmedoid(:,2),'x','LineWidth',4,'Marker-
EdgeColor','k','MarkerSize',25)
```

The scatter plot displays the clustering results along with the indicated position of each medoid. We have specifically adjusted the markers to highlight the medoids' locations.

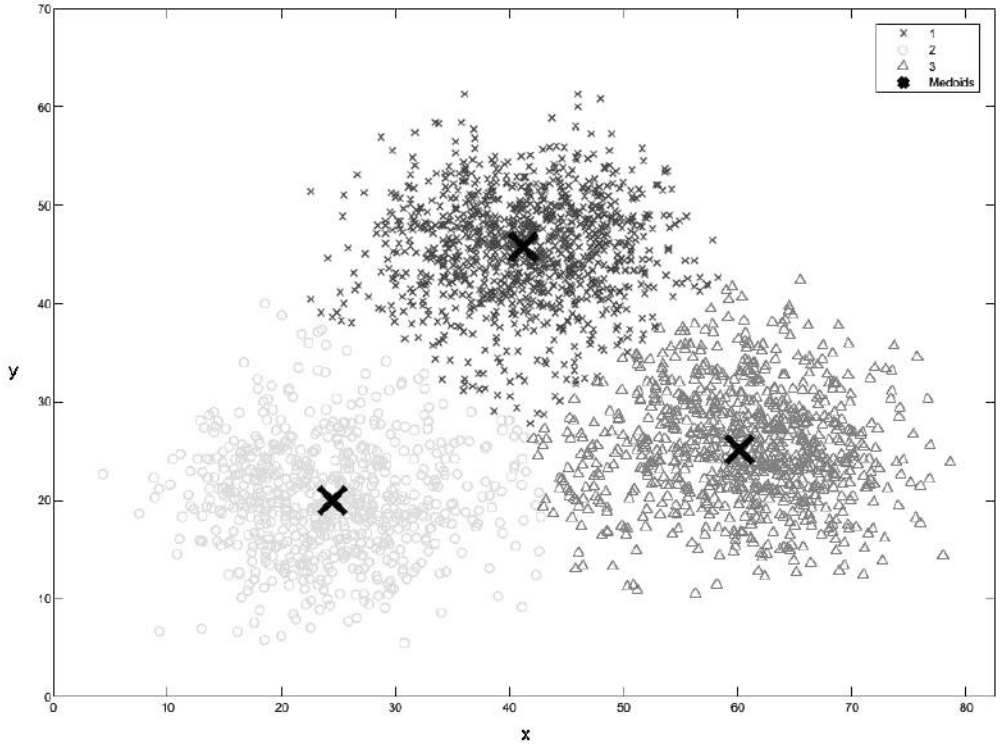


Figure 4.7 – Scatter plot of clustering and the position of each medoid (geographic coordinates of the office: x is the latitude and y is the longitude)

By analyzing *Figure 4.7*, it becomes evident that the peripheral sites are organized into three clusters as determined by the `kmedoids()` function. Similarly, the strategic positioning of the identified hubs is self-explanatory. The distinct colors and markers assigned to each cluster facilitate the quick identification of each site's association with its respective cluster.

After having adequately explored the interesting world of clustering, it is time to move on to analyze how to reduce the size of data in cases with many features.

Discovering dimensionality reduction techniques

Dimensionality reduction is a technique used in machine learning and data analysis to reduce the number of variables or features in a dataset. The goal of dimensionality reduction is to simplify the data while retaining important information, thereby improving the efficiency and effectiveness of subsequent analysis tasks.

High-dimensional datasets can be challenging to work with due to several reasons:

- **Curse of dimensionality**: As the number of features increases, the data becomes more sparse, making it difficult to find meaningful patterns or relationships
- **Computational complexity**: Many algorithms and models become computationally expensive as the dimensionality of the data increases, requiring more time and resources for analysis
- **Overfitting**: High-dimensional data is more susceptible to overfitting, where a model becomes too specialized to the training data and fails to generalize well to new data

Dimensionality reduction methods aim to address these challenges by reducing the number of features while preserving important information. There are two main approaches to dimensionality reduction:

- **Feature selection**: This approach selects a subset of the original features based on certain criteria. It aims to identify the most informative features that contribute significantly to the prediction or analysis task. Common techniques for feature selection include correlation analysis, backward/forward selection, and regularization methods such as L1 regularization (Lasso).
- **Feature extraction**: This approach creates new features by combining or transforming the original features. It aims to capture the underlying structure or patterns in the data. **Principal component analysis** (**PCA**) is a popular feature extraction technique that identifies orthogonal axes in the data that explain the maximum variance. Other methods, such as **singular value decomposition** (**SVD**) and **non-negative matrix factorization** (**NMF**), can also be used for feature extraction.

Both feature selection and feature extraction techniques have their advantages and disadvantages. The choice of method depends on the specific problem, dataset characteristics, and the goals of the analysis.

By reducing the dimensionality of the data, dimensionality reduction methods can lead to benefits such as faster computation, improved model performance, better visualization, and enhanced interpretability of the data. However, it is important to note that dimensionality reduction is not always necessary or beneficial, and it should be applied judiciously after considering the specific requirements and characteristics of the problem at hand.

Introducing feature selection methods

When dealing with high-dimensional datasets, it is often beneficial to reduce the number of features to focus only on the most relevant ones, discarding the rest. This can result in simpler models that generalize more effectively. Feature selection involves the process of identifying the most important features while disregarding others during processing and analysis. It is crucial for creating a functional model that maintains a manageable number of features. In many cases, datasets contain redundant or excessive information, while in other cases, they may include incorrect information. Therefore, feature selection helps to eliminate such issues and improve the overall quality of the model.

Feature selection enhances the efficiency of model creation by reducing the computational load on the CPU and the memory requirements for training algorithms. The selection of features serves several purposes:

- **Preparing clean and interpretable data**: Feature selection helps in selecting the most relevant features, resulting in a cleaner and more understandable dataset. In MATLAB, cleaning a large dataset involves various data preprocessing steps to address missing values, outliers, and inconsistencies.
- **Simplifying models and improving interpretability**: By focusing on a subset of important features, models become simpler and easier to interpret, allowing for better insights into the relationships between variables.
- **Reducing training times**: With fewer features, the training process becomes faster, as there is less data to process and analyze.
- **Mitigating overfitting**: Overfitting occurs when a model is too complex and fits the training data too closely, leading to poor generalization. Feature selection helps in reducing overfitting by decreasing the variance in the model.

Feature selection involves finding a subset of the original variables, typically through an iterative process. By exploring various combinations of variables and comparing prediction errors, the subset that produces the minimum error is selected as the input for the machine learning algorithm.

To perform feature selection, appropriate criteria need to be defined beforehand. These criteria typically involve minimizing a specific predictive error measure for models fitted to different subsets. Feature selection algorithms aim to find a subset of predictors that optimally model the measured responses while considering constraints, such as required or excluded features and the desired subset size.

Feature selection is particularly valuable when the goal is to identify an influential subset, especially in cases involving categorical features where numerical transformations may not be adequate.

Feature selection methodologies can be grouped into three broad categories:

- **Filter methods**: These methods evaluate characteristics independently of the learning model used subsequently. Statistical measures are calculated for each feature, such as the correlation with the output or the relative importance of the features themselves. Features are selected based on these measures, without considering the specific learning model. Examples of filtering methods include correlation analysis, chi-square test, information gain, and mutual information.
- **Wrapper methods**: These methods use a specific learning model to evaluate the quality of different combinations of features. Several subsets of features are created, and a learning model is trained and evaluated for each subset. The goal is to select the subset of features that produce the best performance according to the specified metrics, such as accuracy or mean squared error. However, this approach can be computationally expensive, requiring repeated training and evaluation of many models. Examples of wrapper methods include forward selection, backward selection, and bi-directional elimination.

- **Embedded methods**: These methods perform feature selection within the model training process itself. The learning algorithms used incorporate feature selection mechanisms as part of their optimization process. For example, linear regression algorithms with smoothing (such as ridge regression and Lasso regression) tend to reduce the importance of less relevant features, helping to automatically select the more significant features during training. Examples of embedded methods include Lasso, elastic net, and ridge regression.

Each category of feature selection methods has specific advantages and limitations. The choice of methodology depends on the nature of the problem, the number of features, the availability of data, and the required performance. It is important to experiment and compare different methodologies to find the most suitable for the specific case.

Exploring feature extraction algorithms

When dealing with large datasets, it is often necessary to transform them into a reduced representation of features. This transformation process is known as feature extraction. As covered briefly earlier, feature extraction involves taking an initial set of measured data and creating derivative values that capture the essential information while eliminating redundant or unnecessary data. The goal is to retain the relevant information contained in the original dataset while simplifying it and reducing its dimensionality. By extracting meaningful features, the resulting representation can be more manageable and efficient for subsequent analysis or modeling tasks.

By performing feature extraction, subsequent learning and generalization stages are simplified, and in certain cases, it can result in improved interpretations. It involves deriving new features from the original ones, aiming to reduce the measurement cost, enhance classifier efficiency, and achieve higher classification accuracy. When the extracted features are carefully selected, it is anticipated that the reduced representation will effectively fulfill the desired task instead of using the full-sized input. This enables more efficient and accurate processing as the focus shifts to a subset of features that capture the essential information required for the task at hand.

Feature extraction algorithms are used to derive new features from the original set of features. These algorithms aim to capture the most relevant and informative aspects of the data while reducing dimensionality. Here are a few commonly used feature extraction algorithms:

- **PCA**: PCA is a widely used technique for feature extraction. It identifies the orthogonal axes (principal components) in the data that explain the maximum variance. By projecting the data onto these components, it reduces the dimensionality while preserving the most important information.
- **Linear discriminant analysis (LDA)**: LDA is primarily used for feature extraction in the context of classification tasks. It aims to find a projection of the data that maximizes the separation between different classes while minimizing the variance within each class.

- **Independent component analysis (ICA)**: ICA seeks to identify statistically independent components from a set of observed signals. It assumes that the observed signals are linear combinations of hidden independent components. ICA can be useful for extracting meaningful features in signal processing and blind source separation problems.
- **NMF**: NMF decomposes a non-negative matrix into two lower-rank matrices, where the elements are constrained to be non-negative. It can uncover parts-based representations of the data and is often used for feature extraction in image and text analysis tasks.
- **Autoencoders**: Autoencoders are neural network models that aim to reconstruct the input data from a compressed representation (encoding) layer. The encoding layer represents the extracted features. By training the autoencoder to minimize the reconstruction error, meaningful features can be learned.
- **Wavelet transform**: Wavelet transform decomposes the data into different frequency bands, allowing the extraction of features at various scales. It is commonly used in signal and image processing tasks.

These are just a few examples of feature extraction algorithms. The choice of algorithm depends on the nature of the data, the specific problem, and the desired outcome. It is often beneficial to experiment with multiple algorithms and compare their performance to select the most suitable one for a given task.

Feature extraction works by transforming the original set of features into a new representation that captures the essential information while reducing dimensionality. The process typically involves the following steps:

1. **Data preprocessing**: The input data is preprocessed to handle missing values and outliers, and normalize the features if necessary. This ensures that the data is in a suitable form for feature extraction.
2. **Dimensionality reduction**: This step aims to reduce the number of features while preserving the most important information. Techniques such as PCA, LDA, or NMF are commonly used for dimensionality reduction. These methods identify a lower-dimensional subspace or combination of features that retain the most significant variations in the data.
3. **Feature construction**: In some cases, new features are constructed from the original features. This can involve mathematical operations, transformations, or aggregations. The goal is to create features that capture specific patterns or relationships in the data that may be more informative for the task at hand.

4. **Feature selection**: In addition to dimensionality reduction, feature selection methods may be applied to further filter out irrelevant or redundant features. This helps to focus on the most informative features and reduce the noise in the data. Feature selection can be performed using techniques such as correlation analysis, statistical tests, or wrapper methods that evaluate the impact of different feature subsets on a specific learning algorithm.
5. **Feature representation**: Once the desired set of features is extracted and selected, the data is represented using these features. This transformed representation is often lower-dimensional than the original data and contains the most relevant information for subsequent analysis or modeling tasks. The choice of feature extraction techniques depends on the specific problem, the characteristics of the data, and the goals of the analysis. In the following list, you'll find instances across computer vision applications where a range of feature extraction techniques are put to use:

    - In face recognition, we employ PCA and **local binary patterns** (**LBP**) to extract facial features
    - Object detection benefits from the **histogram of oriented gradients (HOG)**, which captures object shape and texture
    - Image classification employs **convolutional neural networks** (**CNNs**) to engage in hierarchical feature learning

It is important to evaluate the extracted features and assess their impact on the performance of the downstream tasks, such as classification or regression. Feature extraction is an iterative process, and it may require experimentation and fine-tuning to find the most effective combination of techniques for a given task.

After having analyzed in detail all the dimensionality reduction techniques, the time has come to move on to practice by addressing examples of these technologies in the MATLAB environment.

Feature selection and feature extraction using MATLAB

In MATLAB, there are several built-in functions and toolboxes that can be used for dimensionality reduction. In the next section, we will explore some practical examples of the dimensionality reduction algorithm in the MATLAB environment.

Stepwise regression for feature selection

Regression analysis is a valuable approach for understanding the impact of independent variables on a dependent variable. It allows us to identify predictors that hold greater influence over the model's response. Stepwise regression is a variable selection method used to choose a subset of predictors that exhibit the strongest relationship with the dependent variable. There are three common variable selection algorithms:

- **Forward method**: The forward method starts with an empty model, where no predictors are initially selected. In the first step, the variable showing the most significant association at a statistical level is added. In subsequent steps, the remaining variable with the highest statistically significant association is added to the model. This process continues until no more variables demonstrate statistically significant associations with the dependent variable.

- **Backward method**: The backward method begins with a model that includes all variables. It then proceeds step by step to eliminate variables, starting with the one with the least significant association with the dependent variable on the statistical plane.
- **Stepwise method**: The stepwise method alternates between the forward and backward processes. It involves adding and removing variables that gain or lose significance during various model adjustments, including the addition or re-insertion of variables.

These variable selection methods enable the identification of a subset of predictors that best explain the relationship with the dependent variable. The choice of algorithm depends on the specific context and the researcher's goals. Each method has its own strengths and limitations, and it is essential to interpret the selected variables within the context of the overall regression analysis.

In MATLAB, you can create a stepwise regression model using the `stepwiselm()` function. This function returns a linear model by performing stepwise regression, where predictors are added or removed based on their significance. Here's how it works:

1. The `stepwiselm()` function takes a table or dataset array as input and performs stepwise regression on the variables within it.
2. The function uses both forward and backward stepwise regression to determine the final model. It starts with an initial model specified using the `modelspec` attribute.
3. At each step, the function compares the explanatory power of incrementally larger and smaller models. It searches for terms to add or remove based on the value of the `Criterion` argument.
4. The function computes the p-value of an F-statistic to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added. If the evidence is strong enough to reject the null hypothesis, the term is added to the model.
5. Conversely, if a term is already in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.
6. The process continues until no single step improves the model. The function terminates when no terms meet the entrance or exit criteria.

The `stepwiselm()` function allows you to build different models from the same set of potential terms, depending on the initial model and the order in which terms are added or removed. By using this function in MATLAB, you can perform stepwise regression to select the most significant predictors and determine the final model that best explains the relationship between the variables.

To get the data, we utilize the extensive collection of datasets provided by the UCI *Machine Learning Repository* (`https://archive.ics.uci.edu/`). This repository serves as a valuable resource for obtaining a wide range of datasets for various machine learning and data analysis tasks. By leveraging this repository, we can access the necessary dataset required for our analysis or modeling purposes. The UCI Machine Learning Repository offers a diverse collection of datasets contributed by researchers and practitioners, facilitating the exploration and experimentation with different data-driven applications.

The *Yacht Hydrodynamics* dataset is utilized for predicting the hydrodynamic performance of sailing yachts based on their dimensions and velocity. This prediction plays a crucial role in evaluating the overall ship performance and estimating the necessary propulsive power during the initial design phase. The key inputs for this prediction include the fundamental hull dimensions and the velocity of the yacht. Specifically, the inputs consist of hull geometry coefficients and the Froude number, while the output is the residuary resistance per unit weight of displacement. By analyzing these inputs, valuable insights can be gained regarding the performance and efficiency of sailing yachts, aiding in the design and optimization processes.

Let's see how to practically carry out a stepwise regression using MATLAB.

1. To start, we will import the dataset into MATLAB using the `readtable()` function as follows:

```
YHData = readtable('YachtHydrodynamics.xlsx');
```

Let's check the size of the table imported:

```
size(YHData)
ans =
   308     7
```

The table imported contains 308 records of 7 features.

2. With the data now available in the MATLAB workspace in the form of a table, we can proceed to perform stepwise regression:

```
Model1 = stepwiselm(YHData,'constant');
```

In this case we used as a starting model a model with a constant value equal to the intercept, this represents the default setting.

The following text was printed:

```
1. Adding FroudeNumber, FStat = 584.1803, pValue = 6.233076e-73
```

The `stepwiselm()` function in MATLAB creates a linear model using stepwise regression to add or remove predictors from a table or dataset array, starting from a constant model. The response variable used by `stepwiselm()` is the last variable in the table or dataset array. The function employs both forward and backward stepwise regression methods to determine the final model. During each step, `stepwiselm()` searches for terms to add or remove from the model based on the specified criterion value provided through the `Criterion` argument. Terms are evaluated for their significance and contribution to the model's performance. The function iteratively adds or removes predictors that improve the model's fit based on the chosen criterion. By leveraging the `stepwiselm()` function in MATLAB, you can systematically identify and incorporate the most relevant predictors into your linear regression model, ultimately refining and optimizing the model's predictive power.

Stepwise regression allows us to systematically select the most relevant predictors and build a regression model. By executing the stepwise regression algorithm in MATLAB, we can automatically identify the significant predictors that have the strongest relationship with the

dependent variable. This enables us to create an optimized regression model that captures the essential information from the dataset.

Let's print some information about the model:

```
Model1
Model1 =
Linear regression model:
    ResResistance ~ 1 + FroudeNumber
Estimated Coefficients:
            Estimate      SE       tStat       pValue
           ________    ______    _______    __________
(Intercept)    -24.484    1.5336   -15.965  3.6732e-42
FroudeNumber   121.67     5.0339    24.17   6.2331e-73
Number of observations: 308, Error degrees of freedom: 306
Root Mean Squared Error: 8.9
R-squared: 0.656,  Adjusted R-Squared: 0.655
F-statistic vs. constant model: 584, p-value = 6.23e-73
```

We initiated the stepwise regression with a constant model, and the function identified the only variable it deemed statistically significant (`x6 = FroudeNumber`) to include in the model.

3. Let's now try a different approach by starting with a linear model that includes an intercept and linear terms for each predictor. Then, step by step, the function will remove terms that are found to be statistically insignificant. This allows us to iteratively refine the model, retaining only the predictors that have a significant impact on the dependent variable:

```
Model2 = stepwiselm(YHData,'linear');
```

The following text was printed:

```
1. Removing PrismaticCoef, FStat = 0.021132, pValue = 0.88452
2. Removing LongPos, FStat = 0.33493, pValue = 0.5632
3. Removing LengDispRatio, FStat = 0.63427, pValue = 0.42642
4. Removing LengthBeamRatio, FStat = 0.034179, pValue = 0.85345
5. Removing BeamDraughtRatio, FStat = 0.13695, pValue = 0.71159
```

In this way, we can see the order in which the model has removed the features, starting with the ones that are less correlated with the response.

Now, we can print the summary of the model:

```
Model2
Model2 =
Linear regression model:
    ResResistance ~ 1 + FroudeNumber
Estimated Coefficients:
            Estimate      SE       tStat       pValue
           ________    ______    _______    __________
```

```
(Intercept)  -24.484   1.5336    -15.965    3.6732e-42
FroudeNumber  121.67   5.0339      24.17    6.2331e-73
Number of observations: 308, Error degrees of freedom: 306
Root Mean Squared Error: 8.9
R-squared: 0.656,  Adjusted R-Squared: 0.655
F-statistic vs. constant model: 584, p-value = 6.23e-73
```

The result is the same, but the procedure followed is different.

4. To explore the full range of possibilities, we will now create a full quadratic model as the upper bound. We will start with a model that includes an intercept, linear terms, interactions, and squared terms for each predictor. This comprehensive model allows us to capture more complex relationships between the predictors and the dependent variable. However, we will still utilize stepwise regression to iteratively remove terms that lack statistical significance. By doing so, we can refine the model and focus on the predictors that have a substantial impact on the dependent variable, while disregarding those that are not statistically significant:

```
Model3 = stepwiselm(YHData,'quadratic');
```

Lots of information about how the variables were removed will be printed.

Let's see the form of the model obtained:

```
Model3
Model3 =
Linear regression model:
    ResResistance ~ 1 + PrismaticCoef*FroudeNumber +
FroudeNumber^2
Estimated Coefficients:
           Estimate      SE        tStat        pValue
         _________    ______    _________    ___________
(Intercept) -0.34693    17.184 -0.020188       0.98391
PrismaticCoef 69.155    30.254    2.2858      0.022958
FroudeNumber -206.99    58.003   -3.5686    0.00041711
PrismaticCoef:FroudeNumber
             -305.22    99.309   -3.0735     0.0023082
FroudeNumber^2 871.03    25.817   33.739   1.2559e-104
Number of observations: 308, Error degrees of freedom: 303
Root Mean Squared Error: 4.08
R-squared: 0.928,  Adjusted R-Squared: 0.927
F-statistic vs. constant model: 982, p-value = 4.93e-172
```

In this iteration, the resulting model is more intricate and comprehensive. It includes the `FroudeNumber` variable both as a squared term and as part of the interaction with `PrismaticCoef`. This increased complexity better captures the underlying phenomenon, as indicated by the obtained results: an R-squared value of `0.928`, an adjusted R-squared value of `0.927`, and a highly significant p-value of `4.93e-172`.

These metrics suggest that the model provides a highly representative representation of the relationship between the predictors and the dependent variable, explaining a significant portion of the variability observed in the data.

Let's now see how to perform a PCA by analyzing a practical case.

Carrying out PCA

It is a statistical technique used for dimensionality reduction and data analysis. PCA aims to transform a dataset with a potentially large number of variables into a smaller set of uncorrelated variables called principal components. These components are linear combinations of the original variables and are ordered in such a way that the first component captures the maximum amount of variance in the data, the second component captures the next highest amount of variance, and so on.

PCA is often used in various fields, such as machine learning, data visualization, and exploratory data analysis. It helps in identifying patterns and relationships in the data, reducing noise, and simplifying complex datasets. By reducing the dimensionality of the data, PCA can aid in visualizing and interpreting high-dimensional data, as well as improving computational efficiency in subsequent analyses. Additionally, PCA can be used for data preprocessing and feature extraction, allowing for more effective modeling and prediction tasks.

One of the major challenges in multivariate statistical analysis is effectively displaying datasets with numerous variables. Thankfully, in such datasets, it is common for certain variables to be closely interrelated. These variables essentially contain the same information as they measure the same underlying quantity that influences the system's behavior. Consequently, these variables are redundant and do not contribute anything significant to the model we aim to construct. To simplify the problem, we can replace this group of variables with a new variable that encapsulates the relevant information. The following figure illustrates redundant data in a table.

PCA generates a set of new variables, known as principal components, which are uncorrelated with each other. Each principal component is formed as a linear combination of the original variables. The orthogonality between the principal components ensures that there is no redundant information (see *Figure 4.8*). Together, the principal components form an orthogonal basis for the data space. The primary objective of PCA is to explain the maximum variance in the data using the fewest number of principal components. In numerous real-world datasets, particularly those with high dimensionality, redundancy or noise is frequently present. Not all dimensions (features) play an equally vital role in shaping the inherent structure or patterns within the data. By pinpointing the directions (principal components) along which the data showcases the greatest variance, PCA can trim down the data's dimensionality by opting for a subset of these principal components. This simplification streamlines the dataset while conserving its fundamental traits.

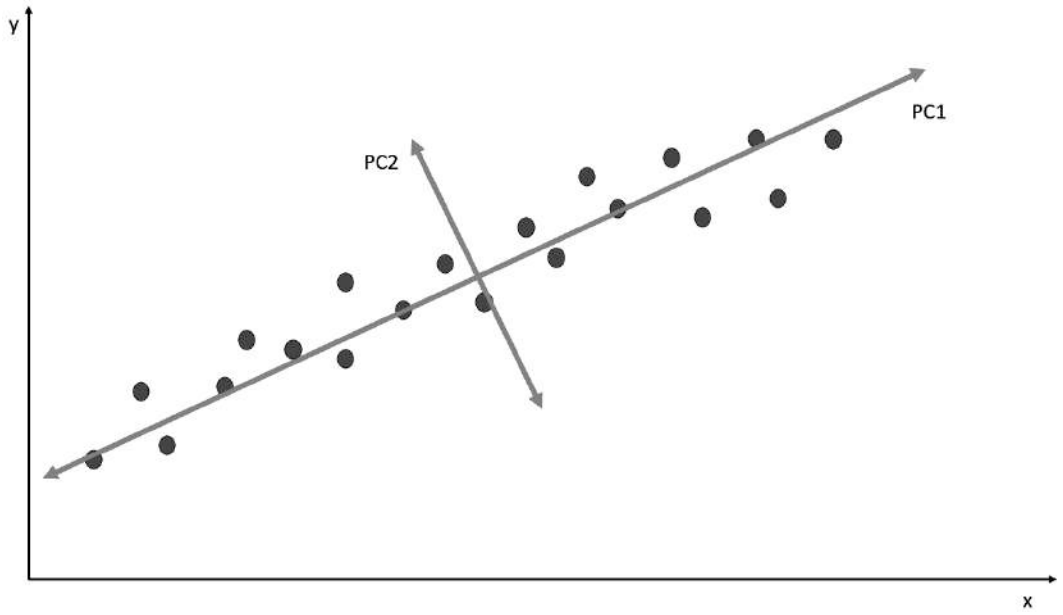


Figure 4.8 – PCA new feature space based on linear combinations of the original feature space

In MATLAB, you can perform PCA using the `pca()` function. This function takes an n-by-p data matrix as input, where each row corresponds to an observation and each column corresponds to a variable. The `pca()` function returns the principal component coefficients, also known as loadings. The coefficient matrix has dimensions p-by-p. Each column of the coefficient matrix contains the coefficients for one principal component, arranged in descending order based on the component variance. By default, the `pca()` function centers the data and utilizes the SVD algorithm.

In this study, we utilized a dataset consisting of measurements of the geometric properties of kernels from three distinct varieties of wheat. The dataset includes kernels from the Kama, Rosa, and Canadian varieties, with a total of 70 samples randomly selected for the experiment. The internal structure of the kernels was visualized using a soft X-ray technique, resulting in high-quality images captured on 13x18 cm X-ray Kodak plates. The wheat grain used in the study was obtained from experimental fields and harvested using a combine harvester. The research was conducted at the Institute of Agrophysics of the Polish Academy of Sciences in Lublin. The data was collected from the UCI Machine Learning Repository:

1. As always, we begin by collecting the data to be analyzed. The `SeedsData -dataset` is multivariate, consisting of 210 instances. Seven geometric parameters of the wheat kernel are used as real-valued attributes organizing an instance. We will first import the dataset into the MATLAB workspace:

```
SeedsData = readtable('SeedsDataset');
```

Before applying the `pca()` function to our data, let's take a preliminary look at the `SeedsData` table. The first seven columns of the table represent the measured variables, while the eighth column indicates the type of seed.

2. To examine the potential relationships between these variables, we can use the `plotmatrix()` function:

```
plotmatrix(SeedsData{:,1:7})
```

The `plotmatrix()` function generates a matrix of subaxes, where the diagonal subaxes display histogram plots of the data in each respective column. The remaining subaxes are scatter plots, representing the relationships between different pairs of columns in the matrix. In the following figure, each subplot in the ith row and jth column represents a scatter plot of the ith column against the jth column. The visualization in *Figure 4.9* provides insights into the data distribution and potential correlations between variables.

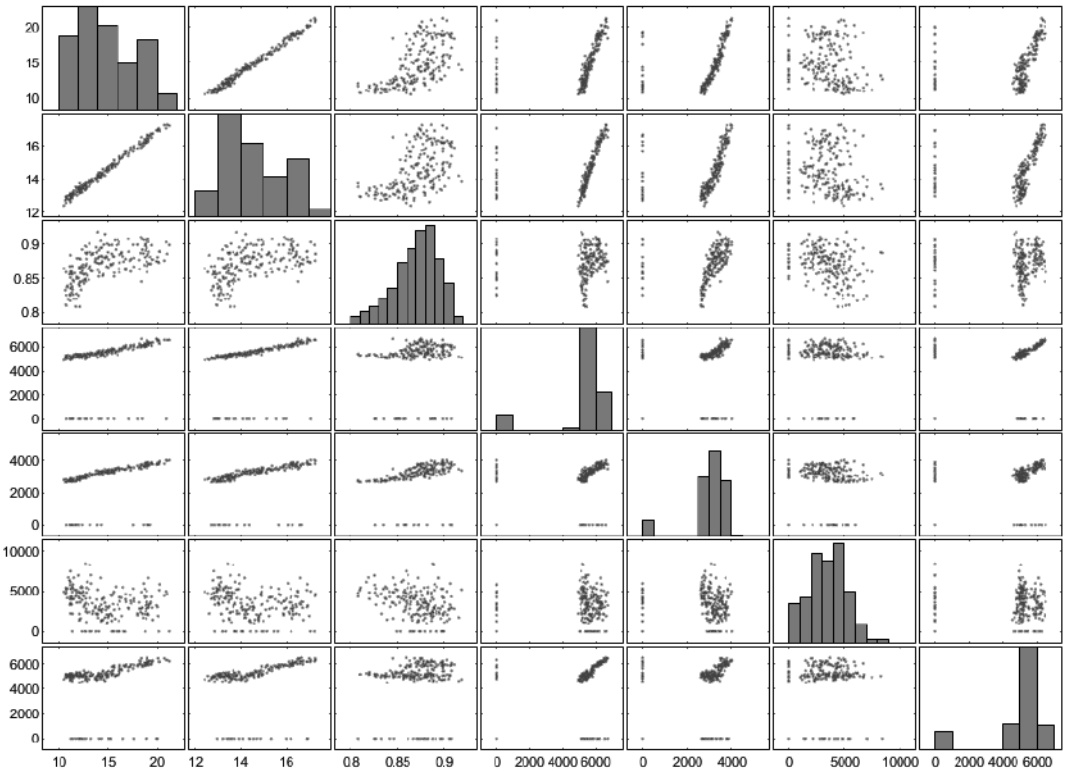


Figure 4.9 – Scatter plot matrix of the measured variables (Area; Perimeter; Compactness; LengthK; WidthK; AsymCoef; LengthKG)

As observed in *Figure 4.9*, scatter plot matrices provide a useful visual tool for identifying potential linear correlations among multiple variables. This aids in pinpointing specific variables that may exhibit mutual correlations, indicating possible redundancy in the data. Additionally, the diagonal of the matrix displays histogram plots, offering insights into the distribution of values for each measured variable. The remaining plots represent scatter plots of the matrix columns, with each plot appearing twice: once in the corresponding row and again in the corresponding column as a mirror image.

Upon analyzing *Figure 4.9*, several plots demonstrate a linear relationship between variables. For instance, the plot showing the relationship between `Area` and `Perimeter`, as well as the one between `Perimeter` and `LengthK`, exhibits such a correlation. However, no correlation can be observed for certain variable pairs. For example, the `LengthKG` variable appears to have no correlation with any other variable, as its data is scattered throughout the plot area.

3. To further validate this initial visual analysis, we can calculate the linear correlation coefficients between the measured variables. The `corr()` function can be employed for this purpose, which returns a matrix containing the pairwise linear correlation coefficient `(r)` between each pair of columns in the user-provided matrix:

```
CorrData=corr(SeedsData{:,1:7})
```

One might question the need to analyze both the scatter plots and the correlation coefficients. The reason is that there are cases where the scatterplots provide information that the correlation coefficients alone cannot convey. If the scatterplot does not indicate a linear relationship between variables, the correlation calculation becomes less meaningful. In such scenarios, two possibilities arise:

- If no relationship exists at all between the variables, calculating the correlation is not appropriate because correlation specifically applies to linear relationships.
- If a strong relationship exists, but it is not linear, the correlation coefficient can be misleading. In certain cases, a strong curved relationship may exist, which cannot be captured accurately by the correlation coefficient.

This emphasizes the critical importance of examining the scatterplots alongside the correlation coefficients.

4. With this understanding, it is now appropriate to proceed with the PCA:

```
[coeff,score,latent,tsquared,explained,mu] =
pca(SeedsData{:,1:7});
```

This code returns the following information:

- `coeff`: The principal component coefficients
- `score`: The principal component scores
- `latent`: The principal component variances

- `tsquared`: The Hotelling's T-squared statistic for each observation
- `explained`: The percentage of the total variance explained by each principal component
- `mu`: The estimated mean of each variable

In situations where the variables are measured in different units or the variance differs significantly across columns, it is often recommended to scale the data or apply weights. The `pca()` function in MATLAB incorporates a default behavior of centering the user-provided matrix by subtracting the column means before conducting either SVD or eigenvalue decomposition.

Additionally, for performing the PCA, the `pca()` function offers three different algorithms:

- SVD (`svd`)
- Eigenvalue decomposition of the covariance matrix (`eig`)
- Alternating least squares algorithm (`als`)

By default, the `pca()` function utilizes the SVD algorithm for the analysis.

Let's analyze the results returned by the `pca()` function. The coefficient matrix, `coeff`, obtained from the PCA, contains the coefficients for the first seven variables present in the `SeedsData` table. The rows of `coeff` represent the variables, while the columns correspond to the principal components. The coefficients within each column determine the linear combination of the original variables that represent the information in the new dimensional space. The columns of `coeff` are arranged in descending order based on the variance of each principal component.

A principal component is a linear combination of the original variables, denoted as `p`, weighted by a vector, `u`. The first principal component is formed by combining the variables with the highest variance. In contrast, the second principal component combines variables with a slightly lower variance, while also ensuring orthogonality to the previous component. This pattern continues for subsequent principal components, each incorporating variables with progressively lower variances while maintaining orthogonality to the preceding components. The number of principal components is equal to the number of observed variables. Each principal component is derived as a linear combination that maximizes the variance while maintaining non-correlation with the preceding components.

The second output, `score`, comprises the coordinates of the original data in the new dimensional space defined by the principal components. It represents how each observation aligns with the principal components and provides a representation of the data in the transformed space.

5. Our goal is to represent the dataset in a new space with reduced dimensions. To achieve this, we plot the first two columns of the `score` matrix, which represent the coordinates of the original data in the new coordinate system defined by the principal components. To make the graph more comprehensible, we group the data by class:

```
gscatter(score(:,1),score(:,2),SeedsData.Seeds,'brg','xo^')
```

The following figure is printed:

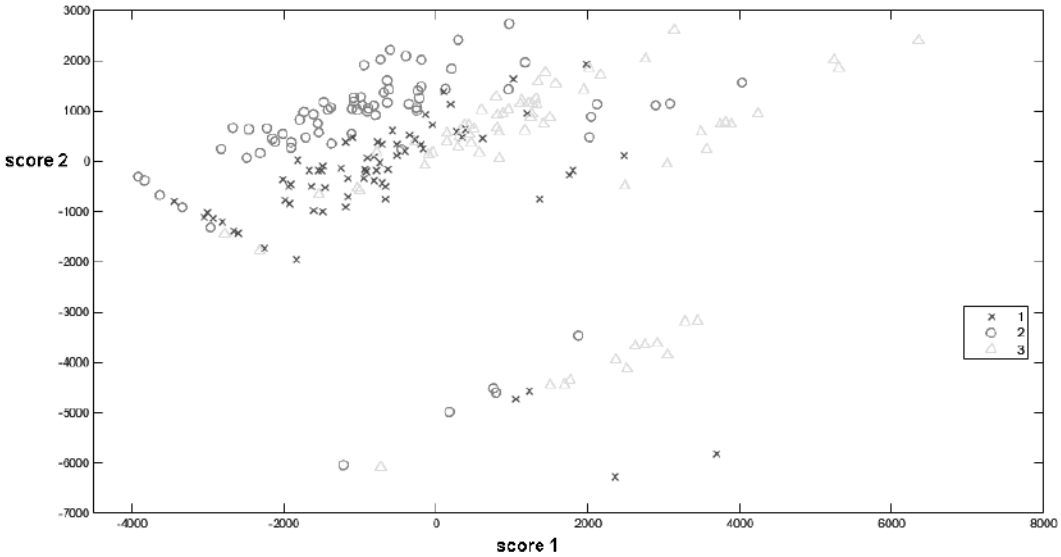


Figure 4.10 – The scatter plot for the first two principal components

The previous figure distinctly classifies the seeds into three distinct classes. The data points are visibly distributed in separate areas of the plot, with minimal uncertainty observed only in the border regions.

Let's revisit the results obtained from the `pca()` function. The third output, `latent`, is a vector that represents the variance explained by each corresponding principal component. The variance of each principal component is reflected in the sample variances of the columns in the `score` matrix, which align with the corresponding rows in `latent`. As previously mentioned, the columns of `latent` are arranged in descending order based on the variance of each principal component.

6. To complete the analysis, we can visualize both the principal component coefficients for each variable and the principal component scores for each observation in a single plot. This type of plot is commonly referred to as a biplot. Biplots serve as exploration plots that enable the simultaneous display of graphical information on both the samples and variables present in a data matrix. In biplots, samples are represented as points, while variables are depicted as vectors:

```
biplot(coeff(:,1:2),'scores',score(:,1:2),'varlabels',SeedsData.
Properties.VariableNames(1:7));
```

The following diagram depicts the principal components coefficients for each variable:

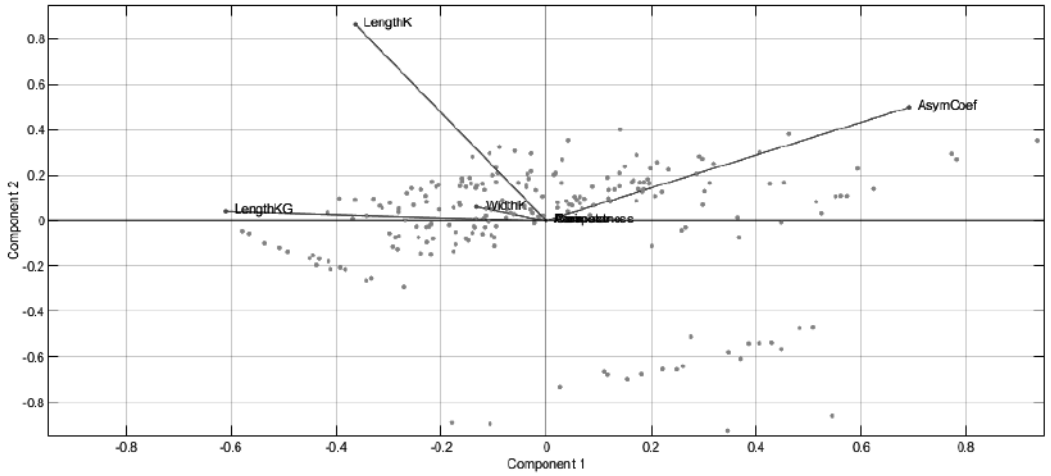


Figure 4.11 – Biplot of the principal component coefficients for each variable and principal component scores for each observation

In the biplot, each of the seven variables is represented by a vector. The direction and length of each vector indicate the contribution of that variable to the two principal components depicted in the plot.

For example, in the first principal component (horizontal axis), four variables have positive coefficients, while three variables have a negative coefficient. This explains why four vectors are directed toward the right half of the plot, while three vectors are directed toward the left half. The largest coefficient in the first principal component corresponds to the `AsymCoef` variable.

Similarly, in the second principal component (vertical axis), all the variables have positive coefficients. By examining the length of the vectors, we can clearly understand the weight of each variable in their respective principal components. It is evident that the `AsymCoef` variable holds a significant weight compared to the others in the first principal component. Similarly, the `LengthK` variable assumes a prominent role in the second principal component.

Summary

In this chapter, we gained knowledge about performing accurate cluster analysis in the MATLAB environment. Our exploration began by understanding the measurement of similarity, including concepts such as element proximity, similarity, and dissimilarity measures. We delved into different methods for grouping objects, namely hierarchical clustering, and partitioning clustering.

Regarding partitioning clustering, we focused on the k-means method. We learned how to iteratively locate *k* centroids, each representing a cluster. We also examined the effectiveness of cluster separation and how to generate a silhouette plot using cluster indices obtained from k-means. The silhouette value for each data point serves as a measure of its similarity to other points within its own cluster,

compared to points in other clusters. Furthermore, we delved into k-medoids clustering, which involves identifying the centers of clusters using medoids instead of centroids. We learned the procedure for locating these medoid centers.

Next, we explored the process of selecting a feature that best represents a given dataset, which is known as dimensionality reduction. We gained an understanding of the fundamental concept behind dimensionality reduction and how it can be achieved through variable transformation.

To perform feature extraction and dimensionality reduction, we learned how to utilize the `stepwiselm()` function. This function enables the creation of a linear model and the automatic addition or removal of variables based on their significance. We also discovered how to construct small models starting from a constant model and large models from models containing numerous terms.

In addition, we discussed techniques for handling missing values within a dataset and explored various methods for extracting features. Among these methods, we specifically analyzed PCA. PCA is a robust quantitative approach that simplifies the data by identifying the most informative components.

In the next chapter, we will delve into the fundamental concepts of artificial neural networks and their implementation in the MATLAB environment. Our focus will be on understanding the basic principles of neural networks and how to apply them effectively. We will explore various aspects of neural network analysis, including data preparation, fitting, pattern recognition, and clustering analysis, all within the MATLAB framework. Additionally, we will delve into the techniques of preprocessing, postprocessing, and network visualization, which play a crucial role in enhancing training efficiency and evaluating network performance. By the end of the chapter, you will have gained practical knowledge on implementing and optimizing artificial neural networks using MATLAB, enabling you to tackle a wide range of data analysis tasks.

5

Introducing Artificial Neural Network Modeling

Artificial neural networks (**ANNs**) include data structures and algorithms for learning and classifying data. Through neural network techniques, a program can learn through examples and create an internal structure of rules to classify different inputs. MATLAB provides algorithms, pre-trained models, and apps to create, train, visualize, and simulate ANNs. In this chapter, we will see how to use MATLAB to build an ANN-based model to predict values and classify data.

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

- Getting started with ANNs
- Training and testing an ANN model in MATLAB
- Understanding data fitting with ANNs
- Discovering pattern recognition using ANNs
- Building a clustering application with an ANN
- Exploring advanced optimization techniques

Technical requirements

In this chapter, we will introduce basic concepts relating to machine learning. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also need a working knowledge of the MATLAB environment.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `ANNFitting.m`
- `ANNPatReg.m`

Getting started with ANNs

Serial computers and their programs exhibit remarkable capabilities in executing tasks that involve repetitive, well-defined operations, prioritizing accuracy, reliability, and speed. While these information-processing systems are highly valuable, they lack true intelligence. The sole element of intelligence lies with the programmer who comprehends the task and formulates the program. To achieve true **artificial intelligence** (**AI**), a system must possess the ability to solve problems that humans consider simple, trivial, and intuitive.

Basic concepts relating to ANNs

ANNs are designed to emulate the intricate workings of biological nervous systems, comprising a vast array of nerve cells or neurons interconnected in a complex network. Typically, each neuron establishes connections with tens of thousands of other neurons, resulting in hundreds of billions of connections. The emergence of intelligent behavior stems from the myriad interactions among these interconnected units.

In this network, specific units serve distinct functions. Some units act as receivers of information from the environment, while others respond to stimuli in the environment. Certain units, known as hidden units, solely communicate within the network, concealed from direct interaction with the external environment. Overall, the neural network's structure involves input units, output units, and hidden units working in concert to process information and exhibit intelligent behavior.

Each unit in the neural network executes a simple operation: it activates when the cumulative received signal surpasses an activation threshold. Once active, the unit transmits a signal to other connected units through communication channels. These connections act as filters, transforming the messages into excitatory or inhibitory signals, and adjusting their intensity based on individual characteristics. Remarkably, the network's input-output link, or transfer function, is not explicitly programmed but rather acquired through a learning process using empirical data, which can be supervised, unsupervised, or reinforcement learning.

ANNs are a type of AI that can learn from experience. This learning is possible thanks to their structure, which is similar to that of the human brain. Like the human brain, ANNs are composed of a large number of nodes, which are connected to each other by connections. Nodes process information, and connections determine how information is transmitted from one node to another. ANNs learn from experience through a process called supervised learning. In this process, ANNs are provided with a dataset of desired input and output examples. ANNs then use this data to learn to map inputs to desired outputs. For example, an ANN that needs to learn to recognize images of dogs is provided with a dataset of images of dogs and images of other animals. The ANN uses these images to learn to distinguish between dogs and other animals.

Neural networks operate in parallel, enabling them to handle multiple data simultaneously, in contrast to serial computers, which process data individually and sequentially. Although individual neurons may be relatively slow, the parallel nature of neural networks accounts for the brain's higher processing speed when tackling tasks requiring the simultaneous handling of numerous data points, such as visual object recognition. This remarkable system exhibits robust noise immunity, akin to a sophisticated statistical model. Even in the event of some unit malfunctions, the overall network performance may experience reductions, but a complete system shutdown is unlikely to occur.

Nevertheless, the latest generation of neural network software demands a solid understanding of statistics. Despite their seemingly immediate usability, users must not be deceived, as they can quickly make predictions or classifications within certain limitations. From an industrial perspective, these networks prove effective when historical data is available for processing with neural algorithms. This capability is particularly valuable in production environments as it facilitates data extraction and model creation.

It's essential to note that while models generated by neural networks are highly efficient, they lack explanations in human symbolic language. The outcomes must be accepted as they are, leading to the characterization of neural networks as black boxes.

Like any modeling algorithm, the efficiency of neural networks relies heavily on the careful selection of predictive variables. These networks require a training phase to establish individual neuron weights, which can be time-consuming when dealing with many records and variables. Unlike conventional models, neural networks lack theorems or definitive guidelines, making the success of a network heavily dependent on the creator's experience.

Neural networks find their utility in scenarios where data may be partially inaccurate or where analytical models are unavailable for problem-solving. They are commonly used in Optical Character Recognition (OCR) software, facial recognition systems, and other applications that handle error-prone or noisy data. Moreover, they are widely employed in data mining analysis and serve as tools for forecasting in financial and meteorological domains. In recent years, their significance has substantially increased in the field of bioinformatics, where they are instrumental in identifying functional and structural patterns in nucleic acids and proteins. By providing a comprehensive set of input data, the network can produce the most probable output.

Understanding how perceptrons work

The fundamental unit of a neural network is the **perceptron**, which emulates the essential functions of a biological neuron. It evaluates the strength of each input, accumulates these inputs, and then compares the sum to a specific threshold. Based on this comparison, the perceptron determines the output value. The neuron's basic structure is well understood, and researchers have identified the primary biochemical reactions that govern its activity. As such, a neuron can be regarded as the elemental computational unit of the brain. Within the human brain, approximately 100 distinct classes of neurons have been identified, each contributing to the intricate neural network responsible for our cognitive processes and abilities.

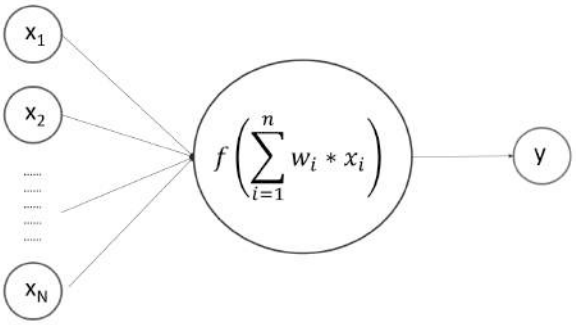


Figure 5.1 – Understanding the perceptron scheme

The primary function of a biological neuron is to generate an electrical potential that travels along its **axon** (neuron output) when the electrical activity at the neuron's cell body surpasses a specific threshold. The neuron's input is received through a set of fibers called **dendrites**, which contact the axons of other neurons, transmitting electrical potentials from them. The point of connection between an axon of one neuron and the dendrite of another neuron is referred to as a **synapse**.

The synapse possesses the ability to regulate the electrical pulse emanating from the axon. The electrical potential generated by a neuron is essentially binary: an on/off state. If the neuron's electrical activity surpasses a specific threshold, an impulse is generated; otherwise, no impulse occurs.

Notably, the intensity of the generated pulse remains consistent across different neurons. As the potential propagates along the axon and reaches the synapse connected to another neuron's dendrite, the post-synaptic potential relies on the biochemical characteristics of the synapse. Despite having the same pre-synaptic potential, two distinct synapses may generate varying post-synaptic potentials. In other words, the synapse modulates and weighs the input potential before transmission.

Post-synaptic potentials continue through the neuron's dendrites and accumulate at the soma level. Only when the sum of these potentials surpasses a certain threshold does the neuron trigger the potential to propagate through its axon. Both biological neurons and artificial neurons receive multiple inputs through dendrites. The artificial neuron aggregates these various input values and computes the result.

If the computed value exceeds a particular threshold, the artificial neuron produces an output signal or potential; otherwise, it remains inactive.

The initial function implemented by the artificial neuron is the algebraic sum of its inputs, which serves to construct the system's response. When simulating a phenomenon, the system may encounter errors, necessitating appropriate corrections. To achieve this, each input is assigned a weight, a numerical value that modulates its impact on the total sum, determining the neuron's potential.

In other words, each input contributes differently to the determination of the threshold value and potential triggering, akin to the characteristic of biological neurons involving synapses between the axon of one neuron and the dendrite of another.

From the perspective of the post-synaptic neuron, the inputs consist of potentials from other neurons whose axons synapse with their dendrites, and these inputs are precisely modulated by the synapses. Some inputs may exert a stronger influence on the total sum, while others could even be inhibitory, reducing the overall sum and thereby lowering the probability of exceeding the threshold and triggering a potential.

This essential property of biological systems is mathematically modeled in connection systems using the concept of weights. Each connection is assigned a numeric value as its weight, which is multiplied by the input value. Consequently, the input's effect on the total sum is determined by the magnitude of its weight.

Activation function to introduce non-linearity

In our previous discussion, we explored the weighted sum function introduced by incorporating the concept of weights. Now, let's delve into another property of the artificial neuron, once again inspired by the behavior of biological neurons. As mentioned earlier, the biological neuron sums up the post-synaptic potentials of its dendrites at the soma level. However, this summation is not a simple algebraic addition of these potentials. Various factors, such as the passive resistance of the neuron membrane, come into play, making the actual summation a function that is typically non-linear.

Similarly, artificial neurons compute the weighted inputs and then modify the result using a specific function. This function is referred to as the activation function, which is applied to the output of the neuron to determine its true potential. The activation function plays a crucial role in shaping the behavior of the neuron and ultimately influences the outcome of the neural network's computation.

The activation function takes the weighted sum of inputs and an additional bias term and applies a specific mathematical operation to produce the neuron's output. This output, often referred to as the activation or the post-activation value, is then used as input to the subsequent layers in the neural network.

Different types of activation functions are utilized in neural networks, each with its unique characteristics and applications. Common activation functions include the following:

- **Step function**: A simple binary function that outputs `1` if the input is greater than or equal to a threshold, and `0` otherwise:

$$f(x) = 0 \; if \; x < 0, \quad 1 \; if \; x \geq 0$$

- **Sigmoid function**: This generates a smooth S-shaped curve, mapping the input to an output between `0` and `1`. It was commonly used in the past but has fallen out of favor in deeper networks due to the vanishing gradient problem:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **Rectified linear unit (ReLU)**: One of the most popular activation functions, ReLU outputs the input value if it is positive; otherwise, it outputs `0`. It helps mitigate the vanishing gradient problem and accelerates convergence in deep networks:

$$f(x) = 0 \; if \; x < 0, \quad x \; if \; x \geq 0$$

- **Hyperbolic tangent (Tanh)**: Similar to the sigmoid function, Tanh produces an S-shaped curve, but it maps the input to an output between `-1` and `1`. It is symmetric around the origin, providing a better range for the gradient compared to the sigmoid function:

$$f(x) = \tanh(x)$$

- **Softmax function**: Primarily used in the output layer for multi-class classification problems, the softmax function converts a vector of real numbers into a probability distribution, ensuring the sum of probabilities equals `1`:

$$f(x)_i = \frac{e^{-x_i}}{\sum_{i=1}^{k} e^{-x_i}}$$

- **Exponential linear unit (ELU)**: ELU is an activation function used in ANNs. It is a smooth, non-saturating function that can handle both positive and negative inputs. ELU is a more recent activation function than ReLU, which is also widely used:

$$f(x) = x \; if \; x > 0$$

$$alpha * (exp(x) - 1) \; if \; x < 0$$

  Here, x is the input to the ELU function and α is a hyperparameter that controls the steepness of the negative slope. The default value of α is `1.0`.

  The ELU function is a smooth, non-saturating function that can handle both positive and negative inputs. This makes it a good choice for tasks that involve both types of inputs, such as image recognition and natural language processing.

The choice of the activation function influences the neural network's performance, training speed, and ability to capture complex relationships in the data. Different activation functions may be used in different layers of the network, depending on the specific problem and architectural considerations.

ANN's architecture explained

Having discussed the properties of an artificial neuron, we will now delve into the architecture of a neural network. This entails both a physical illustration of the network's structure and a determination of the role each neuron plays within this framework. Consider a scenario with multiple inputs and nodes, such that each input is connected to every node. Similarly, each output node receives connections from all inputs. Each output node possesses the characteristics described earlier and carries out its computations in conjunction with the other nodes. Upon introducing an input pattern, the output values are influenced either by the input values themselves or by the network's weights. The weights play a crucial role in the network, determining the extent to which a particular input influences a specific node. The collection of nodes in the structure is commonly referred to as a layer.

Neural networks are organized into layers, each serving a specific purpose in information processing. The primary types of layers in a neural network include the following:

- **Input layer**: The initial layer that receives the input data and passes it on to the subsequent layers.
- **Hidden layers**: Intermediate layers between the input and output layers. These layers process the data and extract relevant features through complex transformations.
- **Output layer**: The final layer that produces the network's output or predictions based on the processed information from the hidden layers.

The number of hidden layers and nodes within them, along with the choice of activation functions and weights, constitute the architecture of the neural network. The architecture (*Figure 5.2*) significantly influences the network's ability to learn from data, generalize to new examples, and perform specific tasks efficiently.

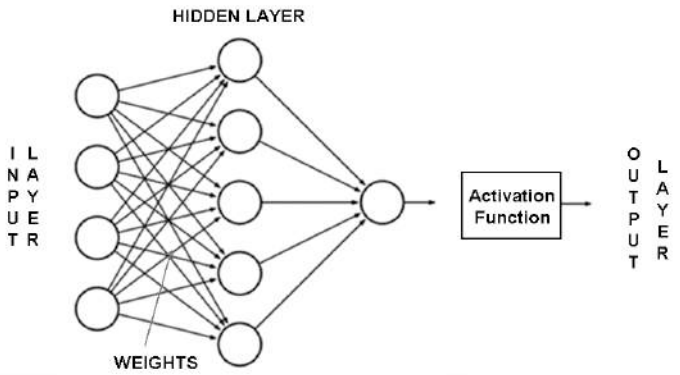


Figure 5.2 – ANN architecture with weights and activation function

Neural networks can take the form of multiple layers, and each layer added enhances the network's computational capacity. Inputs are numerical values that undergo evaluation through the weights of connections with the first layer of nodes, known as the hidden layer. In this hidden layer, each node conducts computations as described earlier, leading to the generation of a potential that then propagates to the nodes of the output layer. The potentials produced by the output nodes collectively represent the final output calculated by the neural network.

The architecture of a neural network refers to the specific way in which the nodes are interconnected. In the case of feedforward neural networks, which are characterized by the architecture shown in the previous figure, the activation of input nodes propagates forward through the hidden layer and further to the output layer. Changing the connections between nodes alters the network's architecture. This not only yields practical consequences in terms of the network's computational capacity but also carries significant theoretical implications related to the concept of learning. The arrangement of nodes in the network affects its ability to learn from data and perform specific tasks, making architecture design a crucial aspect of neural network development.

After analyzing the basic concepts of ANNs, we now need to pay attention to how these algorithms are trained.

Training and testing an ANN model in MATLAB

In the previous section, we saw the architecture of an ANN. It imposes two layers, input and output, which cannot be altered. Consequently, the critical factor lies in the number of hidden layers we consider. The size of a neural network is defined by the number of hidden neurons. Determining the optimal size of the network remains an ongoing challenge, as no analytical solution has been discovered to date. One approach to tackle this problem is to employ a heuristic method: creating various networks with increasing complexity, using a subset of the training data, and monitoring the error on a validation subset simultaneously. After completing the training process, the network with the lowest validation error is chosen as the preferred one.

How to train an ANN

Let's discuss the process of choosing the number of layers. The number of input nodes is fixed based on the number of features in the input data, while the number of output nodes is determined by the number of outcomes to be modeled or the class levels in the outcome. The real challenge lies in determining the appropriate number of neurons for the hidden layer. Unfortunately, there is no analytical method to accomplish this task. The optimal number of neurons depends on various factors, such as the number of input nodes, the volume of training data, and the complexity of the learning algorithm, among others.

Having more neurons in the hidden layer will lead to a model that better fits the training data, but it comes with the risk of overfitting, potentially resulting in poor generalization on future data. Additionally, neural networks with many nodes can be computationally expensive and slow to train.

To address this, a heuristic approach can be adopted, where different configurations are experimented with to find an optimal balance. This trial-and-error method allows us to strike a balance between model complexity, accuracy, and computational efficiency.

ANNs consist of simple elements that operate in parallel. The connections between these elements play a vital role as they dictate the network's functionalities. These connections influence the output through their respective weights, which are adjusted during the neural network's training phase.

During training, the network is fine-tuned by modifying the connection weights, enabling specific inputs to yield desired outputs. For instance, the network can be calibrated by comparing its practical output with the target output we want to achieve. This iterative process continues until the network's output aligns with the desired target.

To obtain dependable results, a substantial number of input/target pairs are required to shape the network appropriately. This training process ensures that the neural network can accurately produce the desired outputs for a variety of inputs, making it a reliable tool for various tasks.

The adjustment of these weights is determined by the specific algorithm we choose to adopt. In the following practical examples, we will discuss and refer to various algorithms that govern the process of weight adjustment.

Introducing the MATLAB Neural Network Toolbox

The **Neural Network Toolbox** offers a range of algorithms, pre-trained models, and apps that enable users to create, train, visualize, and simulate neural networks. It supports both shallow neural networks (with one hidden layer) and deep neural networks (with multiple hidden layers). With these tools, various tasks, such as classification, regression, clustering, dimensionality reduction, time-series forecasting, and dynamic system modeling and control, can be performed.

There are four primary ways to utilize the Neural Network Toolbox:

- **MATLAB graphical user interfaces** (**GUI**): The most user-friendly approach involves using MATLAB GUIs. The main window can be launched using the `nnstart` command, granting access to automatic tasks such as function fitting (`nftool`), pattern recognition (`nprtool`), data clustering (`nctool`), and time-series analysis (`ntstool`).
- **Basic command-line operations**: For greater flexibility, users can utilize command-line operations. While more knowledge is required, this method allows users to have complete control over the process, without relying on menus and icons typically found in the GUI.
- **Customizing the toolbox**: Users can customize the toolbox by creating their own neural networks with arbitrary connections. Existing toolbox training features in the GUI can be used to continue training these custom networks.
- **Modifying toolbox functions**: All computational components in the toolbox are written in MATLAB code and are fully accessible, allowing users to modify and tailor them to specific needs.

This toolbox caters to users of all levels, from beginners to experts. It offers simple tools that guide new users through specific applications and more complex tools that enable experts to customize networks and experiment with new architectures.

Regardless of the approach chosen, a proper analysis using neural networks should encompass the following steps:

1. Data collection
2. Network creation
3. Network configuration
4. Weight and bias initialization
5. Network training
6. Network validation
7. Network testing

By following these steps, users can effectively apply neural networks to various problems and tasks. They are explained in more detail here:

1. The first step in the process involves collecting the data to be analyzed, which is typically done outside the MATLAB environment. This data collection phase is crucial, as the quality of the data will significantly impact the final results and the ability to extract meaningful insights.
2. Next, we proceed to create the neural network using various functions available in the toolbox. These functions allow us to build the network through the chosen algorithm, resulting in the creation of a neural network object. This object stores all the necessary information defining the neural network's properties, such as its architecture, subobject structures, functions, and weight and bias values.
3. The third step is network configuration, where we examine input and output data, set the dimensions of the network to fit the data, and choose appropriate input and output processing settings to enhance network performance. This configuration step is usually performed automatically when the training function is called but can also be done manually using the configuration function.
4. After configuration, the fourth step involves initializing the weights and biases. We set initial values from which the network will begin its training process. This initialization is usually done automatically based on the chosen training algorithm, but users can also set custom values if needed.
5. The fifth step is network training, which is a critical phase of the process. During training, the weights and biases are fine-tuned to optimize the network's performance. This stage is crucial for the network's ability to generalize well with new, unseen data. A portion of the collected data (typically around 70% of available cases) is used for training.

6. Next, in the sixth step, network validation takes place. Here, a fraction of randomly selected data (usually around 15% of available cases) is passed through the network to estimate how well the model has been trained. The results obtained during this phase help determine whether the chosen model adequately reflects the initial expectations or whether adjustments are needed.
7. Finally, in the last step, we utilize the trained network. A portion of the collected data (approximately 15% of available cases) is used to test the network's performance. The trained neural network object can then be saved and employed multiple times with new data as needed. This allows for the reuse of the network to make predictions or analyze various datasets.

The workflow for **neural network design** involves breaking down the collected data into three sets: the **training set**, the **validation set**, and the **test set**. Let's describe each of them in detail:

- **Training set (usually 70% of the available cases)**: The training set is a collection of examples used to train the neural network and adjust its parameters. During the training process, the network learns from the input-output pairs in the training set to optimize its internal weights and biases. The goal is to find the optimal set of parameters that best captures the underlying patterns and relationships in the data. The neural network improves its performance through iterative adjustments during the training phase.
- **Validation set (usually 15% of the available cases)**: The validation set is a separate set of examples used to fine-tune the network's parameters and assess its performance during training. It serves as a validation mechanism to prevent overfitting, a condition where the network performs well on the training data but poorly on new, unseen data. By monitoring the network's performance on the validation set, we can make decisions about the model's complexity, such as determining the optimal number of hidden units or identifying a suitable stopping point for the training algorithm. Adjustments based on the validation set help ensure the network generalizes well to new data.
- **Test set (usually 15% of the available cases)**: The test set is a separate and independent collection of examples used solely to evaluate the performance of a fully trained neural network. After the training and validation stages, the final model is assessed using the test set to estimate its error rate and validate its generalization capabilities. It is crucial to refrain from making any further adjustments to the model based on the test set evaluation to avoid bias or overfitting. The test set provides an unbiased measure of how well the neural network is likely to perform on new, real-world data.

By dividing the data into these three distinct sets, the neural network design workflow ensures that the model is trained, validated, and tested in a controlled and reliable manner, allowing for accurate assessments of its performance and generalization capabilities.

With a clear understanding of this process, we are now ready to proceed with our work on analyzing a practical example of an ANN implementation in MATLAB.

Understanding data fitting with ANNs

Data fitting is the process of constructing a curve or mathematical function that best matches a given set of collected data points. This curve fitting can involve either interpolations, where exact data points are fitted, or smoothing, where a smooth function approximates the data. In the context of regression analysis, curve fitting is closely related to statistical inference, considering uncertainties arising from random errors in observed data.

The approximate curves obtained through data fitting have multiple applications. They can be used to visualize and display the data, predict function values in regions with no available data, and summarize the relationships between multiple variables. This process is valuable for understanding and interpreting complex datasets, making predictions, and gaining insights from the collected information.

Predicting the trend of a particular distribution using mathematical formulas can be challenging, and these formulas may not always accurately represent all the data or cover the entire range of existence. To address such cases, machine learning algorithms come to the rescue. These algorithms can build models without relying on complex mathematical formulas.

ANNs are well suited for data-fitting tasks and trend prediction. They can adapt and learn from a given set of inputs and associated target outputs. **Function fitting** is the process of training a neural network with such input-output pairs, enabling it to form a generalization of the underlying input-output relationship. Once trained, the neural network can generate outputs for inputs it has not encountered during training.

The flexibility of neural networks allows them to capture complex patterns and relationships in the data, making them powerful tools for data-fitting and prediction tasks. By utilizing machine learning algorithms such as neural networks, we can overcome the limitations of traditional mathematical formulas and achieve accurate predictions across various datasets and scenarios.

The performance of algorithms based on machine learning strongly depends on the quality of the data being worked on. The data collection process typically occurs outside the MATLAB environment, which means that you need to have a properly collected data file ready to initiate an analysis in MATLAB. However, if you don't have access to the data yet and are here to learn, there's no need to worry because MATLAB has a solution.

The Neural Network Toolbox software provides several sample datasets that you can use to experiment with the functionality of the toolbox. These sample datasets are readily available and can serve as a starting point for your analysis. To explore the available datasets, you can use the following command:

```
help nndatasets
```

A list of datasets sorted by application type will be returned. With these sample datasets at your disposal, you can begin your learning journey and gain hands-on experience with the Neural Network Toolbox in MATLAB. Now we will work on an example of data fitting in MATLAB:

1. Let's focus our attention on the dataset for data fitting, specifically, `abalone_dataset`, which contains the *abalone shell rings dataset*. To load the dataset into the MATLAB workspace, let's use the following command:

```
load abalone_dataset
```

Executing this command will load the data from the `abaloneInputs` and `abaloneTargets` datasets into the MATLAB workspace. If you prefer to load the input and target arrays with different variable names, you can use the following command:

```
[Input,Target] = abalone_dataset;
```

Executing this command will load the data into the arrays named `Input` and `Target`. The aim of the model is to estimate the age of an abalone, utilizing physical measurements to achieve this prediction.

To obtain a description of a specific dataset, you can use the following command:

```
help abalone_dataset
```

A comprehensive description of the dataset is provided, including details such as the number of attributes, the total number of items, and a list of variables. Additionally, the description offers valuable insights into potential use cases for the dataset.

2. Now that we have the data, we have to choose the training algorithm and set the architecture network. In MATLAB different functions are available for training. To get a list of training algorithms available, we can use the following command:

```
help nntrain
```

For our training, we will use the **Levenberg-Marquardt** (**LM**) **backpropagation**, as follows:

```
TFunc = 'trainlm';
```

LM backpropagation is an optimization algorithm used in training ANNs. It is an extension of the standard backpropagation algorithm that improves convergence and robustness, especially for nonlinear and ill-conditioned problems. In standard backpropagation, the algorithm adjusts the weights of the neural network using the gradient of the error with respect to the weights. However, in some cases, this process can be slow, and the algorithm might get stuck in local minima. LM backpropagation addresses this issue by incorporating the LM optimization method, which is commonly used in nonlinear least squares fitting problems. The LM algorithm combines the ideas of both gradient descent and Gauss-Newton methods.

Here's a basic outline of how LM backpropagation works:

I. Calculate the gradient of the error function with respect to the weights using the standard backpropagation algorithm

II. Calculate the Hessian matrix, which represents the curvature of the error surface with respect to the weights

III. Adjust the weights using a combination of gradient descent and the LM optimization method

The LM backpropagation algorithm adapts the learning rate during training. When the error surface is steep, it behaves more like gradient descent, which helps in avoiding overshooting. When the error surface is relatively flat, it behaves more like the Gauss-Newton method, which speeds up convergence. This combination of techniques makes LM backpropagation an efficient and effective algorithm for training neural networks, particularly in cases where standard backpropagation might face convergence issues or slow learning rates. It is commonly used in various applications, including pattern recognition, function approximation, and nonlinear regression tasks.

After that, we have to set the number of nodes in the hidden layer:

```
HLNodesNum =10;
```

3. Finally, we have to create our ANN using the `fitnet()` function as follows:

```
AbaFitNet = fitnet(HLNodesNum, TFunc);
```

The `fitnet()` function is a built-in MATLAB function used to create and train a feedforward neural network with a single hidden layer for function fitting, pattern recognition, and regression tasks. The following parameters are passed:

- `LnodesNum`: This is a vector that specifies the number of neurons in each hidden layer.
- `Tfunc`: This parameter specifies the training function to be used for training the neural network. It represents the optimization algorithm that updates the network weights during the training process. Some commonly used training functions include `trainlm` (LM), `trainbfg` (BFGS Quasi-Newton), and `traingd` (gradient descent).

4. Training the algorithm is necessary to preprocess the data, as we showed in the *Exploring data wrangling* section, in the *Exploring MATLAB for Machine Learning* chapter. In this case, we can use the neural network processing functions, available in the Neural Network Toolbox. To print all the general data preprocessing functions available, we can use the following command:

```
help nnprocess
```

The following functions are listed:

- `removerows`: Eliminate rows from the matrix based on specified indices
- `mapminmax`: Map the minimum and maximum values of each matrix row to the range [-1, 1]
- `processpca`: Perform principal component analysis on the matrix rows
- `mapstd`: Map the row means and deviations of the matrix to standard values
- `fixunknowns`: Handle matrix rows with unknown values using a specific procedure

The first two represent the default functions applied to feedforward multilayer networks and are therefore the ones we will apply to our case:

```
AbaFitNet.input.processFcns =
                    {'removeconstantrows','mapminmax'};
AbaFitNet.output.processFcns =
                    {'removeconstantrows','mapminmax'};
```

As already mentioned, these are the default processing functions applied to both input and output. The first removes the constant records, as they do not bring any content for data adaptation, and the second instead maps the elements of a matrix or vector from their original range to a specified target range.

5. Then the preprocessing of data is necessary to operate **data splitting**. This is a common technique used in machine learning and data analysis to divide a dataset into separate subsets for different purposes. The main reason for data splitting is to have distinct portions of data for training, validation, and testing, which allows for the evaluation and improvement of machine learning models. There are several techniques to split the data; in this case, the dataset is split into three parts: a training set, a validation set, and a testing set. The training set is used for model training, the validation set is used to tune hyperparameters and optimize the model, and the testing set is used for final evaluation:

```
AbaFitNet.divideFcn = 'dividerand';
```

The `dividerand` property randomly assigns each sample to one of the three subsets based on the specified ratios. It returns three index vectors: `trainInd`, `valInd`, and `testInd`, which indicate the samples' positions in the original dataset for the training, validation, and testing sets, respectively. This attribute specifies the data splitting method employed during the training of the network with a supervised algorithm such as backpropagation. You can assign the name of the desired data division function to this attribute. When this attribute is modified, the network's adaptation parameters are automatically updated to include the parameters and default values associated with the newly selected function:

```
AbaFitNet.divideMode = 'sample';
```

This attribute determines the dimensions of the target data to be partitioned when invoking the data division function. The default value is `sample` for static networks and `time` for dynamic networks. Alternatively, it can be set to `sampletime` to divide targets by both sample and timestep, `all` to partition targets at each scalar value, or `none` to keep the data undivided (meaning all data is used for training and none for validation or testing).

Let's now set the percentage of data to use for the different phases:

```
AbaFitNet.divideParam.trainRatio = 70/100;
AbaFitNet.divideParam.valRatio = 15/100;
AbaFitNet.divideParam.testRatio = 15/100;
```

This attribute contains the parameters and corresponding values of the currently employed data division function. The choice of data splitting method depends on the specific problem, dataset size, and available resources. To find out the splitting functions available, we can use the following command:

```
help nndivision
```

It is essential to perform data splitting carefully to avoid issues such as data leakage and ensure reliable model evaluation.

6. Before training the ANN, it is necessary to choose something. To start with, we must choose the evaluation metrics to check the performance of the model that we are setting. Evaluation metrics are quantitative measures used to assess the performance and effectiveness of a model, algorithm, system, or process. In various fields, such as machine learning, data science, and information retrieval, evaluation metrics are essential for comparing different methods, tuning parameters, and understanding the strengths and weaknesses of a particular solution. The choice of evaluation metric depends on the specific problem being addressed and the goals of the analysis. The selection of appropriate metrics depends on the nature of the problem and the objectives of the analysis. It's essential to choose evaluation metrics that align with the specific goals and requirements of the task at hand. We can use a command such as this:

```
AbaFitNet.performFcn = 'mse';
```

In MATLAB, there are some commonly used evaluation metrics available. In this case, we have used the **mean squared error** (**MSE**), which measures the average squared difference between predicted values and actual values. In other words, it measures how well a model's predictions align with the actual data points. Larger errors result in a higher MSE, while smaller errors result in a lower MSE. The MSE quantifies the deviation between a regression line and a set of data points. It serves as a risk metric that aligns with the expected value of the squared error loss. The MSE is calculated by averaging the squared differences between the observed and predicted values over the entire dataset.

To obtain a comprehensive list of performance functions, please enter the following command in the command line:

```
help nnperformance
```

A list of the evaluation metrics with a short summary will be printed.

7. Finally, we must choose the plot functions to get a visual representation of the results. To do that, we can use the `plotFcns()` function, which contains a one-dimensional cell array of strings that defines the plot functions associated with a network. The neural network training window, accessible through the `train()` function, displays a button for each plot function. Simply click on the respective button during or after the training process to open the desired plot:

```
AbaFitNet.plotFcns = {'plottrainstate','plotperform',
                      'ploterrhist',   'plotregression'};
```

Four plots were selected:

- `plottrainstate`: Reports the training state values
- `plotperform`: Reports the network performance
- `ploterrhist`: Reports the error histogram
- `plotregression`: Reports the linear regression

To obtain a comprehensive list of plot functions, please enter the following command in the command line:

```
help nnplot
```

A list of the plots available for an ANN with a short summary will be returned.

8. Now we can train the ANN already set; for that, we can use the `train()` function as follows:

```
[AbaFitNet,Trs] = train(AbaFitNet,Input,Target);
```

This function trains a shallow neural network, and three arguments are passed:

- `AbaFitNet`: The neural network model object.
- `Input`: The input data used for training the neural network. It should be a matrix where each row represents a single input pattern. The abalone dataset is a collection of features describing the physical dimensions of an abalone. This dataset consists of eight features: sex, length, diameter, height, whole weight, shucked weight, viscera weight, and shell weight.
- `Target`: The target data corresponding to the input patterns. It should be a matrix with the same number of rows as the input matrix, where each row contains the corresponding target values. The target is the age of an abalone, measured in the form of the number of rings.

The `train()` function performs the training process using the specified input data and target data. The type of training algorithm used depends on the specific neural network model and its settings. After training, the object returned will contain the trained neural network, which can be used for making predictions on new data using the tasks related to the trained network.

During the training phase, a new window will be opened (as shown in *Figure 5.3*).

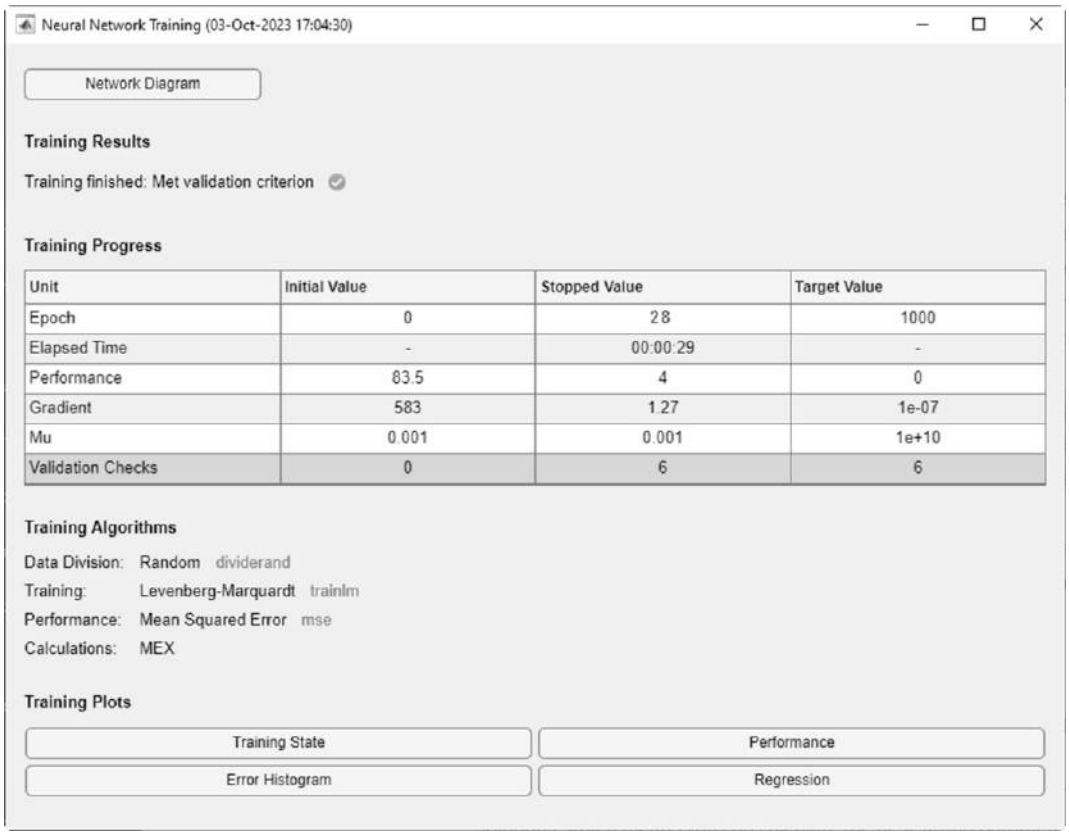


Figure 5.3 – Training results window

In *Figure 5.3*, we can check the training progress, which shows information such as **Epoch**, **Elapsed Time**, **Performance**, **Gradient**, **Mu**, and **Validation Checks**. Furthermore, the properties set in the previous steps are summarized. Finally, several buttons related to the plot set are available.

By clicking on the buttons at the bottom of this window, we can draw the specific plot.

9. After the ANN is trained, it is time to test the network. Testing a neural network involves evaluating its performance on a separate test dataset that the network has not seen during the training phase. This step helps assess the generalization ability of the trained network and ensures it can make accurate predictions on unseen data. To test the ANN that we trained, we can apply the network to the unseen data, as follows:

```
SimTarget = AbaFitNet(Input);
Diff = gsubtract(Target, SimTarget);
Performance = perform(AbaFitNet, Target, SimTarget)
```

In this piece of code, we start evaluating the network on all the data, obtaining a first value of performance to compare with other ones.

The following performance is returned:

```
Performance =
    4.1539
```

10. Now, we have to recalculate training, validation, and testing performance as follows:

```
trainTargets =Target.* Trs.trainMask{1};
valTargets =Target.* Trs.valMask{1};
testTargets =TargetB.* Trs.testMask{1};
trainPerformance = perform(AbaFitNet,trainTargets, SimTarget)
valPerformance = perform(AbaFitNet,valTargets, SimTarget)
testPerformance = perform(AbaFitNet,testTargets, SimTarget)
```

The following performance values are returned:

```
trainPerformance =     3.9470
valPerformance =     5.1255
testPerformance =     4.1468
```

To show the network architecture, we can use the following command:

```
view(AbaFitNet)
```

Finally, to show the plots, we can use the following commands:

```
figure,plotperform(Trs)
```

The `plotperform()` function shows the network performance by using the training record returned by the specific training function.

```
figure,plottrainstate(Trs)
```

The `plottrainstate()` function plots the training states returned by the `train()` function.

```
figure,ploterrhist(Diff)
```

A good way to evaluate the error made in data simulation is to represent it using a histogram. To do this, we will use the `ploterrhist()` function, which plots the error histogram. An error histogram is a graphical representation that illustrates the distribution of errors in a dataset. Errors, in this context, typically refer to the differences between predicted values and actual (observed) values. In practice, examining an error histogram can provide valuable insights into the performance of a predictive model. It can help identify patterns in the errors, assess the model's bias, and detect the presence of outliers or other issues that may need attention during model development and evaluation.

```
figure,plotregression(Target, SimTarget)
```

Finally, to evaluate the network's ability to estimate the model target, we used the `plotregression()` function. This function plots the linear regression of targets relative to outputs.

The following plots will be printed:

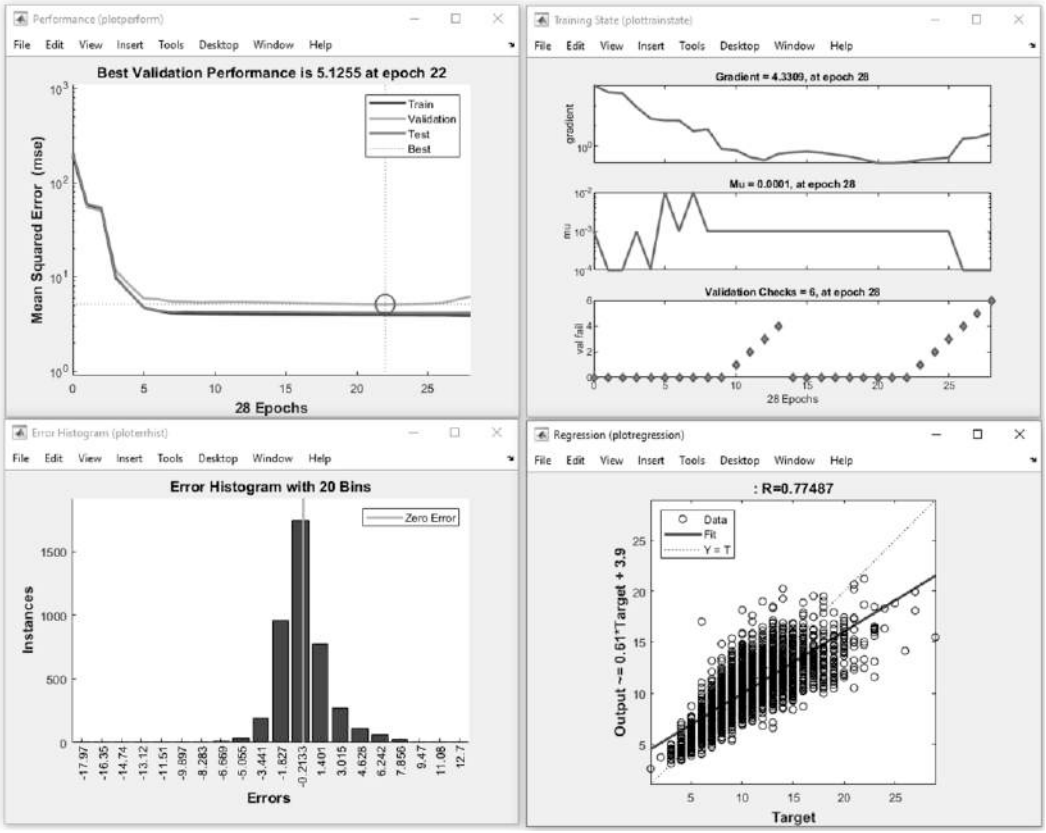


Figure 5.4 – ANN plots

In this way, we will be able to evaluate the trend of the training parameters throughout the process period by period. Furthermore, we will have a statistic of the distribution of errors and, finally, an indication of the position of the forecasts compared to the real values through the regression graph.

In *Figure 5.4*, we can analyze the regression line of the model. The regression line is a key element in statistical data analysis, providing valuable information about a model's ability to predict the data. This straight line represents the mathematical relationship between the independent and dependent variables, trying to minimize the difference between the observed values and those predicted by the model. The slope of the line reflects the average change in the dependent variable for each unit change in the independent variable. If the slope is close to `1`, the model accurately predicts the data. Additionally, the intercept indicates the value of the dependent variable when the independent variable is 0.

The goodness of fit of the model is measured through the correlation coefficient *R*, which measures the strength of the association between two variables. A high *R* indicates good predictive ability. It is important to note that the regression line may have limitations in its predictive ability if the data has complex or nonlinear patterns. In such cases, more advanced models may be necessary.

After having analyzed an example of data fitting in detail, we will now see how to tackle a classification problem.

Discovering pattern recognition using ANNs

Pattern recognition is a branch of machine learning and AI that focuses on the identification of patterns or regularities in data. It involves the automatic discovery and extraction of meaningful information from datasets, with the aim of categorizing or classifying data into different classes or groups. Overall, pattern recognition plays a crucial role in automating the process of identifying patterns and making decisions based on data, making it a fundamental component of many modern AI systems.

Pattern recognition offers numerous benefits in automating decision-making and handling complex data. However, it also poses challenges related to data quality, interpretability, and computational requirements. To leverage its advantages effectively, practitioners need to carefully design and train models while being aware of potential limitations and biases in the data and algorithms.

The classical approach to pattern recognition uses data collected by sensors as input to a classification system. This data generally represents typical measurements such as kinematic and kinetic data from a motion analysis system. The measurements are usually subjected to a preprocessing phase. This is done to improve the signal properties. Subsequent feature extraction provides a feature vector for subsequent classification. This vector describes the input measurements in feature space. In supervised classification, labeled feature vectors are presented to a classifier for training. The vectors used to train the classifier form the training set. These labels assign a feature vector to one of several possible classes. In the recognition phase, the trained classifier uses this decision rule and automatically assigns a feature vector to a class. Different classifiers can be used, using different learning strategies.

Let's now analyze a practical example of pattern recognition using MATLAB. In this section, our objective is to develop a classification model that can categorize thyroid disease based on various patient data. The steps involved are as follows:

1. To initiate the process, we acquire the data for analysis. For this study, we will utilize an existing dataset that comes pre-packaged with MATLAB. As mentioned earlier, MATLAB offers several readily available databases that can be easily imported into the workspace using the `load` command, followed by the specific database name. In this instance, we will work with the `thyroid_dataset` as our chosen dataset:

   ```
   [Input,Target] = thyroid_dataset;
   ```

   In the MATLAB workspace, we now have two variables:

- **Input**: This is a 21x7,200 matrix representing data from 7,200 patients. Each patient is characterized by 15 binary attributes and 6 continuous attributes.
- **Target**: This is a 3x7,200 matrix containing associated class vectors for each input. It indicates which of the three classes each patient belongs to. In the `Target` variable, classes are represented by a `1` in the first, second, or third row. A `1` in the first row indicates the patient is classified as normal (not hyperthyroid), a `1` in the second row indicates hyperfunction (hyperthyroidism), and a `1` in the third row indicates subnormal functioning (hypothyroidism).

The specific problem at hand is to determine whether a patient referred to the clinic is hypothyroid. It is important to note that due to many patients not being hyperthyroid (92%), a successful classifier must achieve a significantly higher accuracy than 92% to be considered effective.

2. Now, it is time to select the appropriate training function for the neural network. To set a specific training algorithm for the network, you can use the `trainFcn` property and assign the name of the desired function. Among several available algorithms, we will opt for the **scaled conjugate gradient** (**SCG**) backpropagation method. This method is an optimization algorithm commonly used in training ANNs, particularly for solving the problem of weight updates during the training process. It is an alternative to other optimization techniques, such as gradient descent, **stochastic gradient descent** (**SGD**), and various flavors of backpropagation. SCG is known for its efficiency and speed in converging to a minimum of the loss function. The advantages of the SCG algorithm include its ability to converge quickly and efficiently in many cases, making it a good choice for training neural networks with relatively small to moderate-sized datasets. However, it may not always outperform other optimization methods on large datasets or in more complex network architectures. The choice of optimization algorithm often depends on the specific problem, dataset size, and computational resources available:

```
trainFcn = 'trainscg'
```

3. After selecting the training algorithm, the next step is to construct the neural network. To achieve this, we need to determine the number of nodes in the hidden layer. In this case, we have decided to build a function-fitting neural network with one hidden layer consisting of 10 nodes:

```
hiddenLayerSize = 10;
```

4. We can construct a pattern recognition network using the `patternnet()` function. Pattern recognition networks are feedforward networks designed for classifying inputs based on target classes. The target data for these networks should consist of vectors with all `0` values except for a `1` in the element corresponding to the class they represent.

   The `patternnet()` function accepts the following arguments:

   - `hiddenSizes`: A row vector specifying one or more hidden layer sizes. The default is `10` if a value is not provided.

- `trainFcn`: The training function to be used. The default is `trainscg` (SCG backpropagation).
- `performFcn`: The performance function used during training. The default is `crossentropy`.

The function returns a pattern recognition neural network with the specified architecture and settings:

```
TyroidPatNet = patternnet(hiddenLayerSize, trainFcn);
```

5. Once the network has been constructed and the data has been preprocessed, the next step is to partition the data into separate sets for training, validation, and testing.

   To achieve this, we typically divide the available dataset into three different subsets:

   - **Training set**: This subset is used to train the neural network. The network learns from the input data and corresponding target outputs during the training process.
   - **Validation set**: The validation set is used to fine-tune the network's hyperparameters and prevent overfitting. It helps in optimizing the network's performance by evaluating its performance on data it has not seen during training.
   - **Testing set**: This set is used to evaluate the final performance of the trained network. It provides an unbiased estimate of the network's generalization ability on unseen data.

   The division of data can be achieved using various techniques, such as random sampling, stratified sampling, or time-based splitting, depending on the nature of the data and the specific problem:

```
TyroidPatNet.divideFcn = 'dividerand';
TyroidPatNet.divideMode = 'sample';
TyroidPatNet.divideParam.trainRatio = 70/100;
TyroidPatNet.divideParam.valRatio = 15/100;
TyroidPatNet.divideParam.testRatio = 15/100;
```

   The significance of operations and the types of functions used in the network construction and data preprocessing steps was extensively covered previously. In case of any uncertainties, it is recommended that you review that paragraph for a more detailed understanding.

6. Regarding measuring network performance, we will select the **cross-entropy performance** function. This specific performance function is well suited for classification and pattern recognition tasks. It quantifies the network's performance by calculating the cross-entropy, which measures the difference between estimated and actual class memberships.

   By utilizing the cross-entropy performance function, we can effectively evaluate how well the neural network performs in classifying inputs and make informed decisions about its effectiveness for the given pattern recognition problem.

By dividing the data into these separate sets, we can ensure that the neural network's performance is not only measured on the training data but also validated on unseen data, making the evaluation more reliable and indicative of the network's true capabilities:

```
TyroidPatNet.performFcn = 'crossentropy';
```

7. Finally, we can now set plot functions to visualize the results of the simulation:

```
TyroidPatNet.plotFcns =
{'plotperform','plottrainstate','ploterrhist', 'plotconfusion',
'plotroc'};
```

8. Now, we are ready to initiate the training process for the network using the `train()` function:

```
[TyroidPatNet,Trs] = train(TyroidPatNet,Input,Target);
```

While training the neural network, the **Neural Network Training** window will be displayed. This window comprises four sections, each offering valuable information throughout the training process:

- **Neural network**: This section provides a summary of the neural network's architecture and configuration, including the number of layers, nodes in each layer, and the chosen training algorithm.
- **Algorithms**: In this area, details about the training algorithm being used, such as the specific optimization technique and convergence criteria, are presented.
- **Progress**: The progress section shows real-time updates on the training process, such as the current epoch, training error, and validation performance. It allows monitoring of the network's performance as it improves over successive epochs.
- **Plots**: This section displays various plots, such as training and validation errors over epochs, enabling a visual assessment of the network's learning progress and potential overfitting.

The **Neural Network Training** window (*Figure 5.5*) offers a comprehensive view of the training procedure, providing insights into the network's behavior and performance at different stages. These insights help fine-tune the network and make informed decisions to optimize its performance for the pattern recognition task at hand.

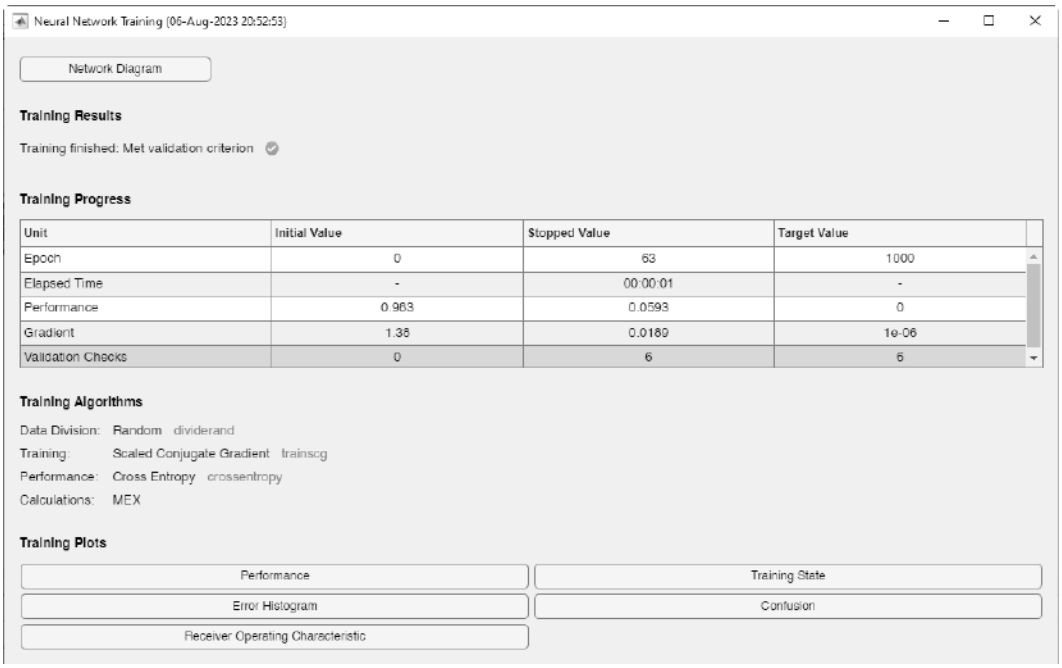


Figure 5.5 – Neural Network Training window for pattern recognition

9. Once the network training is complete, we can employ the trained model to test its performance on the same input data used during the training phase. By doing so, we can obtain the results and utilize them for evaluation purposes.

   Testing the network on the same data helps us assess how well the model generalizes to familiar inputs and provides insights into its effectiveness in handling real-world scenarios. The evaluation results obtained from this process aid in understanding the network's accuracy, precision, recall, and other relevant metrics, allowing us to make informed decisions about its overall performance and suitability for the intended pattern recognition task:

```
SimData = TyroidPatNet(Input);
Diff = gsubtract(Target, SimData);
performance = perform(TyroidPatNet, Target, SimData);
TargetInd = vec2ind(Target);
SimDataInd = vec2ind(SimData);
percentErrors = sum(TargetInd ~=
                    SimDataInd)/numel(TargetInd);
```

The `vec2ind()` function is used to convert vectors to indices. It allows indices to be represented either directly as themselves or as vectors with a `1` in the row corresponding to the index they represent. In the context of this problem, `TargetInd` and `SimDataInd` are vectors containing values `1`, `2`, or `3`, representing the classes to which the targets and outputs belong. The last row of these vectors contains the percentage of error occurrences.

The following value is obtained:

```
percentErrors = 0.0742
```

10. Now, let's proceed with the evaluation of the network. The following commands extract the outputs and targets that pertain to the training, validation, and test subsets. This extracted data will be used in the subsequent step to construct the confusion matrix, which will aid in further assessing the performance of the network on each of these subsets:

```
trOut = SimData(:,Trs.trainInd);
vOut = SimData (:,Trs.valInd);
tsOut = SimData (:,Trs.testInd);
trTarg = Target(:,Trs.trainInd);
vTarg = Target (:,Trs.valInd);
tsTarg = Target (:,Trs.testInd);
```

The subsequent command generates separate plots for the confusion matrix during each phase of the evaluation (training, validation, and testing) and an additional plot representing the overall confusion matrix for the entire evaluation process. These plots will provide a visual representation of the network's performance in classifying the data across different phases and help in understanding the overall classification accuracy and potential misclassifications:

```
figure, plotconfusion(trTarg, trOut, 'Train', vTarg, vOut,
'Validation', tsTarg, tsOut, 'Testing', Target,SimData,'All')
```

A **confusion matrix** is a valuable tool that allows us to compare the classification results of our model to the real data. It provides insights into the nature and quantity of classification errors. The matrix consists of cells where the diagonal elements represent the number of cases that were correctly classified, while the off-diagonal elements show the misclassified cases.

In an ideal situation, a machine learning algorithm should perfectly discriminate between two populations (such as healthy and diseased) that are not overlapping (mutually exclusive). However, in real-world scenarios, the two populations often overlap to some extent, leading to the algorithm making some false positive and false negative predictions.

Figure 5.6 displays the confusion matrix for the training, testing, and validation phases, as well as a combined matrix that considers all three sets of data together.

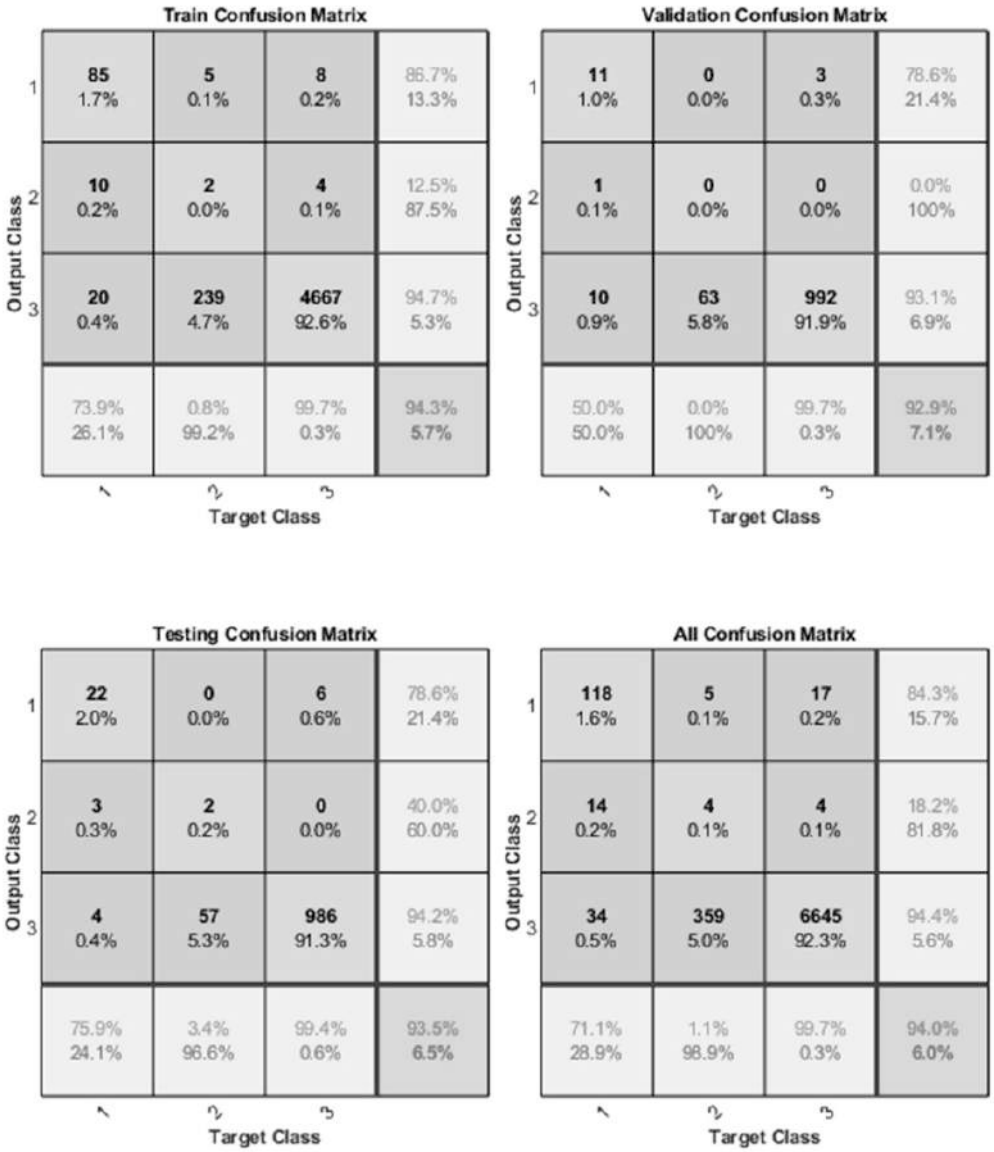


Figure 5.6 – The confusion matrix for training, testing, and validation, and the three kinds of data combined

This visual representation will help us understand how well the model performed across different phases and its overall classification performance considering all data subsets. The confusion matrix has been computed for the training, testing, and validation phases, as well as for the combination of all three data subsets. This comprehensive matrix provides a detailed overview of the model's classification performance across various evaluation stages and gives a complete picture of its effectiveness in handling different datasets.

In *Figure 5.6*, the blue cell located in the bottom-right corner represents the total percentage of correctly classified cases, which are depicted by green cells positioned diagonally. The red cells in the matrix signify the total percentage of misclassified cases located in other cells. The confusion matrix organizes data with rows representing the actual values and columns representing the predicted values. For instance, in the top-left plot of the figure, the first row indicates that 85 cases are correctly classified as `1` (normal), 5 cases are incorrectly classified as `2` (hyperfunction), and 8 cases are incorrectly classified as `3` (subnormal). By observing the blue cell in the bottom right of each plot in the confusion matrix, we can deduce that the classification accuracy is consistently high, exceeding 92%. These results demonstrate excellent recognition capabilities. If higher accuracy is required, retraining the data could be considered. It should be noted that the starting dataset is not correctly balanced across all classes. This can result in a high-performance value for one class and a low-performance value for another. It is therefore advisable to always work on datasets that are correctly balanced across all classes.

Another method of evaluating network performance is through the **receiver operating characteristic** (**ROC**). The ROC curve is a valuable tool for assessing the model's performance in terms of sensitivity and specificity across various classification thresholds. The subsequent command generates plots for the ROC during each evaluation phase and for the entire process:

```
figure, plotroc(trTarg, trOut, 'Train', vTarg, vOut,
'Validation', tsTarg, tsOut, 'Testing', Target,SimData,'All')
```

The ROC is a metric used to evaluate the performance of classifiers. It assesses the quality of a classifier for each class by applying threshold values across the interval `[0, 1]` to its outputs.

In the following figure, ROC plots are displayed for the training, testing, and validation phases, as well as for the combined data from all three subsets. These ROC curves allow us to analyze the classifier's sensitivity and specificity across different classification thresholds for each class, providing valuable insights into its discrimination capabilities and overall performance.

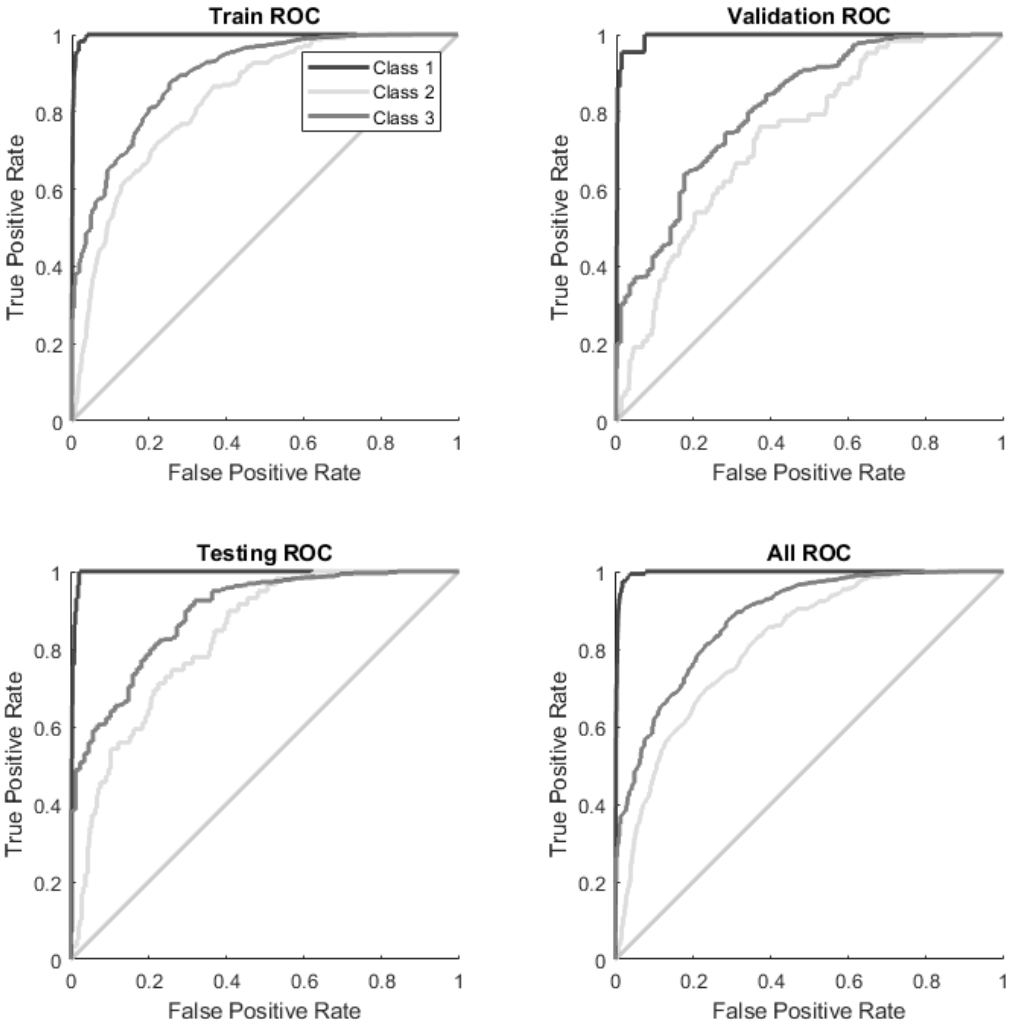


Figure 5.7 – ROC curves for training, testing, and validation, and the three kinds of data combined

In the graph, the colored lines on each axis represent the ROC curves. The ROC curve illustrates the relationship between the **true positive rate** (**TPR, or sensitivity**) and the **false positive rate** (**FPR**) as the classification threshold is varied. TPR measures the proportion of actual positive cases that are correctly identified as positive by the classifier.

$$TPR = TP / (TP + FN)$$

Here, we have the following:

- *TP* = True positives
- *FN* = False negatives

In other words, it calculates the ability of the model to detect all positive instances in the dataset. FPR measures the proportion of actual negative cases that are incorrectly classified as positive by the classifier. It quantifies the rate of false alarms or false positives made by the model.

$$FPR = FP / (FP + TN)$$

Here, we have the following:

- *FP* is the number of false positives, which are the number of negative examples that are incorrectly classified as positive
- *TN* is the number of true negatives, which are the number of negative examples that are correctly classified as negative

An ideal test would exhibit points located in the upper-left corner, indicating 100% sensitivity and 100% specificity. The closer the lines approach the upper-left corner, the better the network's performance, indicating its ability to achieve higher sensitivity while keeping the FPR low, leading to improved classification accuracy.

After tackling a pattern recognition problem using the tools available in MATLAB, in the next practical example, we will see how to tackle a clustering problem with the help of the Neural Network tool.

Building a clustering application with an ANN

Clustering is a popular unsupervised machine learning technique used for grouping similar data points together in a dataset. The goal of clustering is to partition data into clusters in such a way that data points within the same cluster are more like each other than those in other clusters. We examined this topic in depth in *Chapter 4, Clustering Analysis and Dimensionality Reduction*.

In this section, we will see how to address a clustering problem using an ANN in the MATLAB environment. So far, to train a neural network in the MATLAB environment, we have used the commands available from the command line, or much more simply to be implemented in a script with the `.m` extension to reproduce the algorithm whenever we like. But MATLAB has out-of-the-box apps that let us use a wizard to train an ANN. We will do this to address a clustering problem as follows:

1. To get an overview of the apps made available by the Neural Network tool, we can type the following on the command line:

   ```
   nnstart
   ```

   The following window will open:

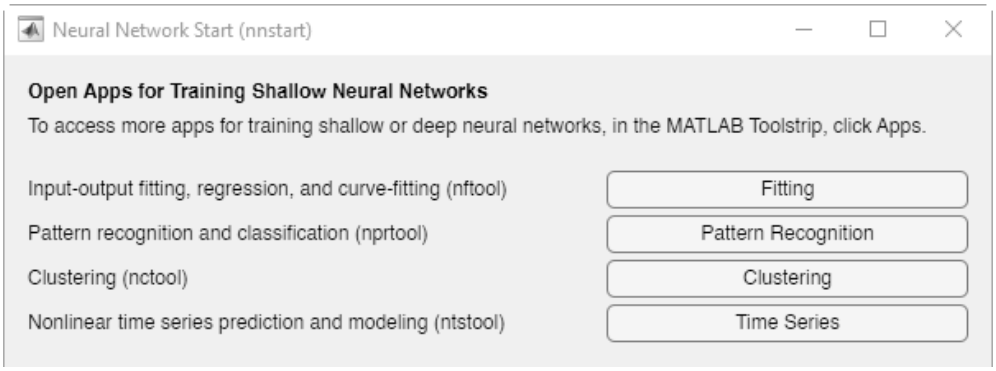


Figure 5.8 – Neural Network start window

Four apps are available: **Fitting**, **Pattern Recognition**, **Clustering**, and **Time Series** (**Nonlinear time series prediction and modeling (ntstool)**). The first two tasks were already addressed in this chapter in the *Discovering pattern recognition using ANNs* section. To do this, we used the command line, but we can also use the app available in the MATLAB environment. Specifically, we can use the `nftool` command to start the **Fitting** app, and the `nprtool` command to open the **Pattern Recognition** app.

In this example, we will use the **Clustering** app, which is started using the following command:

```
nctool
```

The following window will open:

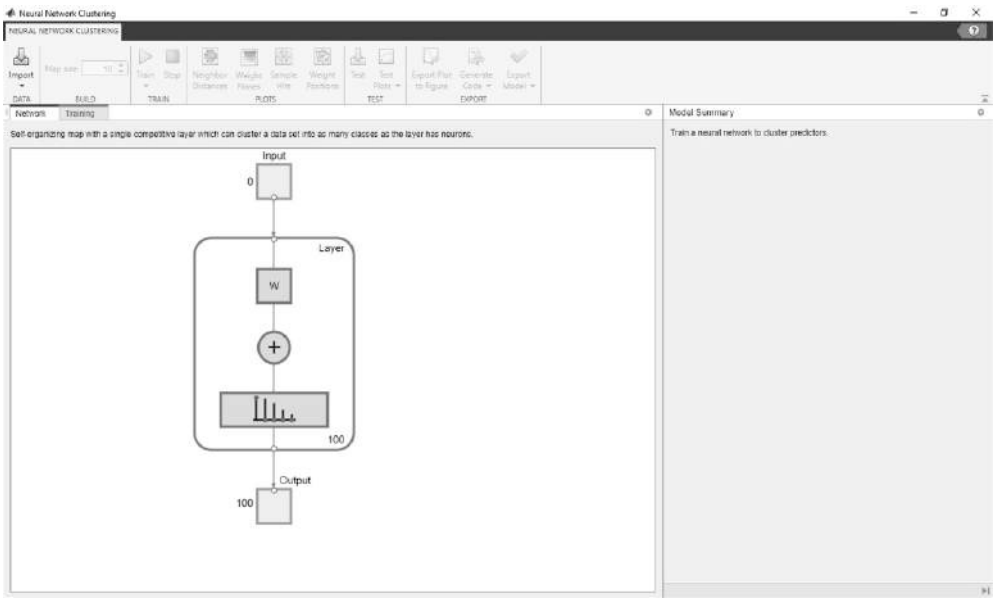


Figure 5.9 – MATLAB app for clustering using ANN

In this app, we can use a wizard for the training of the ANN. Let's see how.

2. To start, we have to import the data. For this example, we will use a dataset already available in the MATLAB environment to explain the clustering application using an ANN. To import that dataset in the MATLAB workspace, we use the following command:

```
[Input,Target] = simplecluster_dataset;
```

Now, we have two matrices in the MATLAB workspace (`Input` and `Target`). We will use only the first (wx1,000 double) for a clustering problem, which is the `Input` matrix with two variables and 1,000 records. By examining the structure of the `Target` (4x1,000 double) matrix, we realize that four classes are available, so the data has `4` groups of data, which will be useful in justifying the results.

Now, we have to import the data in the app. To do this, we can click on the **Import** button in the window shown in *Figure 5.9*. The window shown in *Figure 5.10* will open:

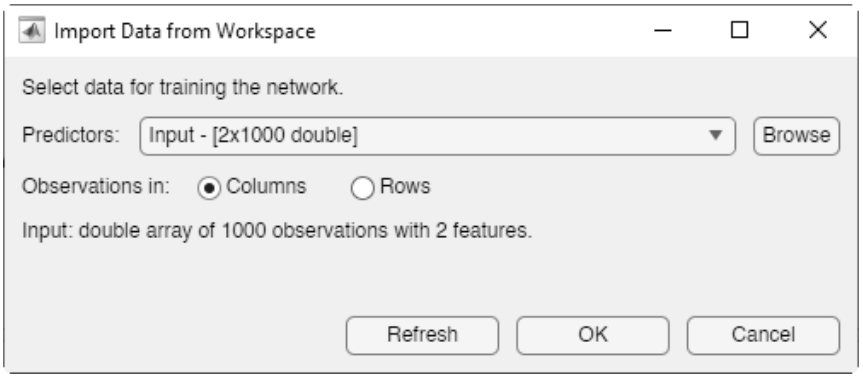


Figure 5.10 – Import data in Clustering app

The right matrix was already detected for **Predictors**; if you want to change the data, you can select the right one using the browse button next to the **Predictors** field. In this case, we have to set the observations in columns, so we will have 1,000 observations with 2 features. Let's just click on the **OK** button to import the data.

3. We have the ANN architecture already set in the app, so we can identify the number of inputs and the topology of the network. The output will be a network with 10x10 nodes (100 nodes). The `nctool` guides you in solving clustering problems by employing a **self-organizing map** (**SOM**). This map creates a condensed depiction of the input space, capturing both the density patterns of input vectors in that space and a compressed, two-dimensional representation of the input space's topology.

SOMs, also known as **Kohonen maps**, are a type of ANN that belongs to the family of unsupervised learning algorithms. They were introduced by the Finnish professor Teuvo Kohonen in the 1980s. SOMs are used for tasks such as dimensionality reduction, data visualization, clustering, and feature extraction.

The primary idea behind SOMs is to map high-dimensional input data onto a lower-dimensional grid or lattice in such a way that similar input data points are mapped to nearby grid cells. This results in a topological representation of the input data, where similar data points are located close to each other on the map, allowing for easier visualization and interpretation.

SOMs consist of two essential layers: the input layer and the output layer, often referred to as the feature map. The input layer serves as the initial stage in a SOM. Each data point from the dataset competes for representation to identify its own characteristics. The process begins with weight vector initialization, kickstarting the mapping process of the SOM.

The mapped vectors are subsequently scrutinized to identify the weight vector that best represents the chosen sample using a randomly selected sample vector. Nearby weights to each weighted vector are considered, and the chosen weight evolves into a vector for the random sample, fostering the map's growth and the emergence of new patterns. In a two-dimensional feature space, these patterns often take on hexagonal or square shapes. This entire process is repeated over 1,000 times.

In essence, learning occurs as follows:

I. Each node is analyzed to determine whether its weights are similar to the input vector. The node that best matches the input vector is termed the **best matching unit** (**BMU**).

II. The neighborhood value of the BMU is then established, and over time, the number of neighbors tends to decrease.

III. The BMU's weight vector further adapts to resemble the sample vector, leading to similar changes in the surrounding areas. The weight of a node changes more as it gets closer to the BMU and less as it moves away from its neighbors.

IV. Repeat step 2 for *N* iterations.

This iterative process allows the SOM to refine its representation of the input data, ultimately leading to a more organized and compressed representation that captures the underlying structure of the data.

Now, we can train the network just by clicking on the **train** button of the app. After a few seconds, the ANN will be trained and ready for use.

4. Finally, we can view the results using the plots available at the top of the **App** window. We will first use the neighbor distances, and the following plot will be drawn:

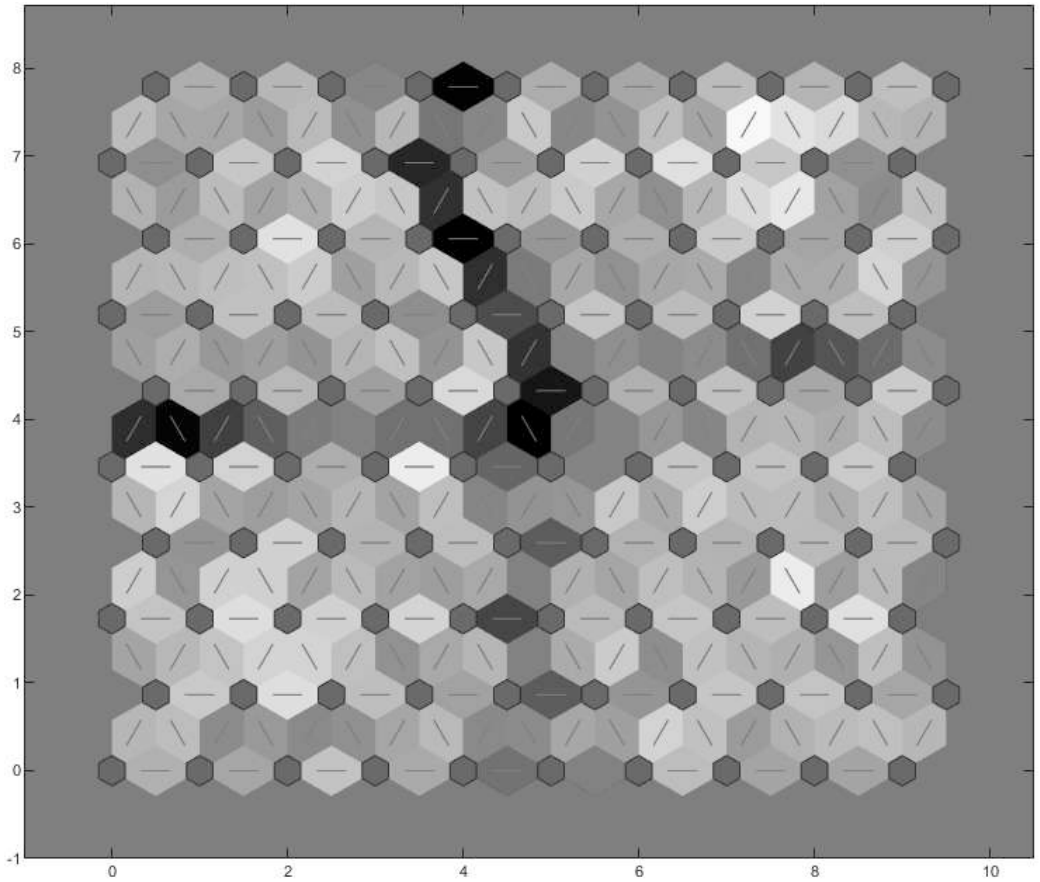


Figure 5.11 – SOM neighbor weight distances

This is a graphical representation that illustrates the distances between weight vectors of neighboring nodes in a SOM. This plot helps visualize how the weights of neighboring nodes change during the learning process, providing insights into the topology and organization of the SOM. We can identify more colored cells that separate the nodes in four areas using the class available in the `Target` matrix.

The blue hexagons represent the neurons. The red lines connect neighboring neurons. The colors in the regions containing the red lines indicate the distances between neurons. The darker colors represent larger distances, and the lighter colors represent smaller distances.

One visualization tool for the SOM is the weight distance matrix (also called the U-matrix). To view the U-matrix, click **SOM Neighbor Distances** in the training window.

This shows us that there are four potential clusters in the data correctly detected by the algorithm. To have a confirmation of this indication, we can plot the weight position plot:

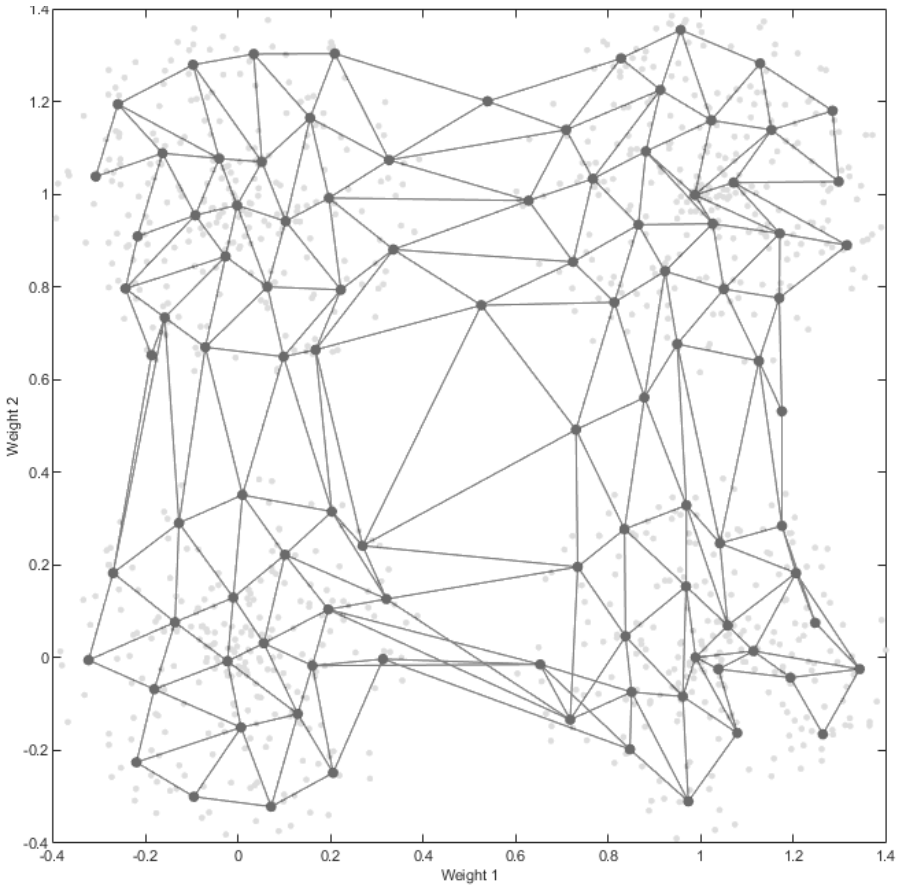


Figure 5.12 – SOM weight positions

In *Figure 5.12*, the four clusters can be easily identified. In this diagram, the input vectors are depicted as green dots, and it illustrates the SOM's classification of the input space by displaying blue-gray dots representing each neuron's weight vector. Additionally, neighboring neurons relate to red lines, offering a visual representation of the SOM's organizational structure.

After having also analyzed a case of clustering with ANNs, to complete the topic, we will see how to optimize the results obtained.

Exploring advanced optimization techniques

Advanced optimization techniques are powerful methods used to enhance the efficiency and effectiveness of optimization algorithms. These techniques aim to overcome the limitations of traditional optimization approaches, particularly in complex, high-dimensional, or non-convex optimization problems.

In machine learning, advanced optimization techniques are essential for training complex models effectively, improving convergence, avoiding overfitting, and handling high-dimensional data. In the following subsection, some advanced optimization techniques commonly used in machine learning are listed.

Understanding SGD

SGD is a popular and fundamental optimization algorithm used in machine learning for training models, especially in large-scale and complex settings. It's a variant of the traditional gradient descent method designed to address efficiency and convergence issues when dealing with large datasets. The stochastic aspect of SGD comes from the fact that it uses a random mini-batch for each iteration, making the optimization process more stochastic (randomized) compared to the deterministic nature of regular gradient descent, which uses the entire dataset for each update. This stochasticity introduces noise into the gradient estimates, which can help the algorithm escape local minima, converge faster, and generalize better.

The primary objective of any optimization algorithm is to minimize the discrepancy between the predicted values of the model and the actual values observed in the data. The smaller the error between the observed and predicted values, the more effective the algorithm is at simulating the real-world scenario. Minimizing this discrepancy is equivalent to optimizing an objective function on which the model construction is based.

Descent methods are iterative techniques that, commencing from an initial point $x_0 \in R_n$, produce a sequence of points $\{x_n\} \in N$ based on the subsequent equation:

$$x_{n+1} = x_n + \gamma_n * g_n$$

In a descent method, the vector g_n represents the search direction, and the scalar γ_n serves as a positive parameter known as the step length, determining the distance of movement in the g_n direction. These choices of g_n and γ_n are made to ensure the reduction of the objective function f in each iteration, following the principle:

$$f_{x_{n+1}} < f_{x_n} \forall\, n \geq 0$$

We select the vector g_n as a descent direction, ensuring that the line $x = x_n + \gamma_n * g_n$ creates an obtuse angle with the gradient vector $\nabla f(x_n)$. This guarantees the decrease of the objective function f, as long as the value of γ_n is sufficiently small. This approach allows for various descent methods, contingent on the specific choice of g_n.

A gradient is a function that produces a vector, representing the slope of the tangent to the graph of a function. It points in the direction of the greatest increase in the function. Let's examine the convex function illustrated in the following figure:

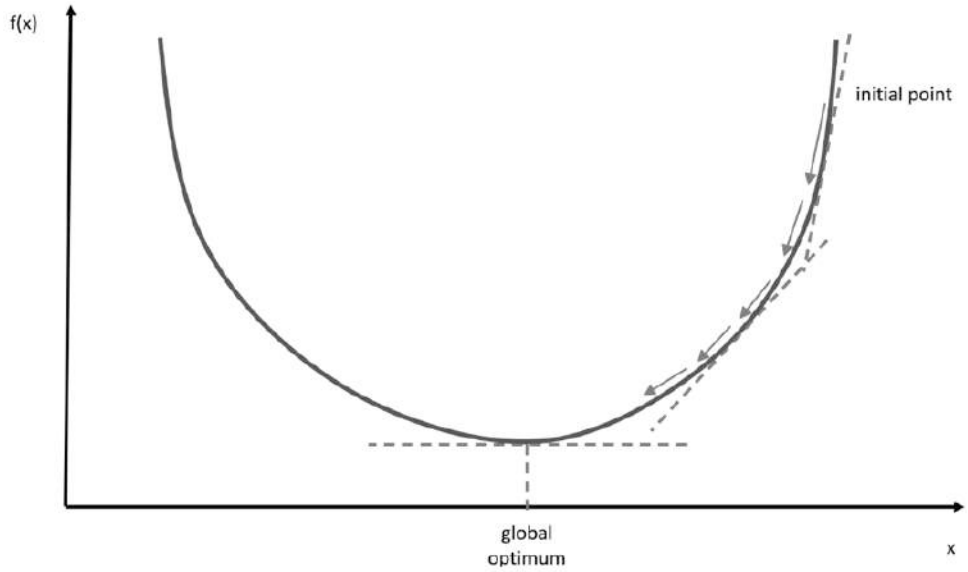


Figure 5.13 – How the gradient descent algorithm searches the global optimum

The primary objective of the gradient descent algorithm is to locate the function's nadir, or lowest point. In more precise terms, the gradient functions as a derivative, indicating the incline or steepness of the objective function.

To provide a clearer analogy, let's imagine we find ourselves lost in the mountains at night with limited visibility. Our perception is restricted to sensing the slope of the terrain beneath our feet. The aim is to reach the lowest point of the mountain. Achieving this goal involves taking successive steps in the direction of the steepest slope. We proceed iteratively, advancing step by step, until we finally arrive at the valley of the mountain.

We'll examine a two-variable function, denoted as *f(x, y)*, with its gradient represented as a vector encompassing the partial derivatives of *f*. The first derivative pertains to *x*, while the second derivative pertains to *y*. Upon computation of these partial derivatives, the results are as follows:

$$\frac{\delta f}{\delta x}, \frac{\delta f}{\delta y}$$

The initial expression corresponds to the partial derivative concerning *x*, whereas the subsequent expression pertains to the partial derivative regarding *y*. The gradient is represented by the ensuing vector:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \end{bmatrix}$$

The given equation represents a function in a two-dimensional space, effectively forming a two-dimensional vector. Each component of this vector signifies the steepest ascent direction for the respective function variable. Consequently, the gradient points toward the direction where the function exhibits the most significant increase.

Likewise, if we consider a function with five variables, the resulting gradient vector will encompass five partial derivatives. In general, a function with *n* variables gives rise to an *n*-dimensional gradient vector, exemplified as follows:

$$\nabla f(x, y, \ldots . z) = \begin{bmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \\ \cdots \\ \cdots \\ \frac{\delta f}{\delta z} \end{bmatrix}$$

When utilizing gradient descent, our objective isn't to maximize *f* as rapidly as possible; instead, we aim to minimize it—specifically, to locate the smallest point that minimizes the function.

Consider a function $y = f(x)$. The foundation of gradient descent relies on the observation that when the function *f* is well defined and differentiable within a vicinity of *x*, it experiences a faster decrease as we proceed in the direction opposite to the negative gradient. Commencing from an initial value of *x*, we can express this as follows:

$$x_n + 1 = x_n - \gamma * \nabla f(x_n)$$

where gamma (γ) is learning rate and delta (∇) is the gradient.

In this context, we have the following:

- γ represents the learning rate
- ∇ signifies the gradient

When using sufficiently small values of γ, the algorithm converges to the minimum value of the function *f* within a finite number of iterations. In essence, when the gradient is negative, it indicates a decreasing objective function at that point, implying that the parameter should shift toward larger values to approach a minimum point. Conversely, when the gradient is positive, the parameters should move toward smaller values to attain lower values of the objective function.

Exploring Adam optimization

Adaptive moment estimation (**Adam**) is an advanced optimization algorithm commonly used for training deep learning models, particularly in neural networks. It is an extension of the SGD optimization method that adapts the learning rate for each parameter individually.

This gradient descent algorithm aims to locate the objective function's minimum using an iterative approach. In each step, an approximation of the gradient is computed to guide the descent in the direction that effectively reduces the objective function. Within this process, the selection of the learning rate parameter holds significant importance. This parameter governs the speed at which we approach the optimal objective function values:

- If the learning rate is excessively small, a substantial number of iterations are required to converge toward the optimal values
- Conversely, an overly high learning rate might cause us to overlook the optimal solution

Adam adjusts the learning rates for each parameter based on the first-order moment (mean) and second-order moment (uncentered variance) of the gradients. This adaptability helps the algorithm converge more efficiently, especially in high-dimensional spaces with varying gradient magnitudes.

This method addresses the issue of initial bias in the moments by correcting them. In the early stages of training, the moments' estimates may be biased toward 0, and Adam compensates for this bias, particularly when the learning rate is low.

Adam combines the concepts of momentum (accumulating a fraction of the past gradients to enhance convergence) and **RMSProp** (scaling the learning rates based on the magnitudes of recent gradients) to perform well on a wide range of optimization problems.

The Adam optimization algorithm involves the following calculations during each iteration:

- Compute the gradient of the loss with respect to the model's parameters
- Update the first and second moments (mean and uncentered variance) of the gradients using a moving average
- Correct the bias in the moments (especially in the early iterations)
- Update the parameters using the corrected moments and the learning rate

Overall, Adam is effective in many deep learning scenarios, often providing faster convergence than standard SGD with manually tuned learning rates. It's a popular choice for training neural networks and has become a standard optimization algorithm in many deep learning frameworks due to its adaptive properties and strong performance.

Introducing second-order methods

Second-order optimization methods, also known as **Newton-like methods**, are advanced techniques used for solving optimization problems. Unlike first-order methods (such as gradient descent) that primarily rely on gradients, second-order methods take advantage of both gradients and second-order derivatives (**Hessian matrix**) of the objective function. This additional information can lead to faster convergence and more accurate solutions, especially in complex and non-convex optimization landscapes.

Second-order methods consider not only the first-order derivative (gradient) of the function but also its second-order derivative (Hessian matrix). The Hessian matrix captures curvature information and provides insights into the shape of the objective function's surface. These methods often use quadratic approximations of the objective function around the current point. These approximations consider the first and second derivatives and provide a more accurate representation of the local behavior of the function.

The use of curvature information from the Hessian matrix can lead to faster convergence rates compared to first-order methods, especially when the objective function is well behaved and smooth. Second-order methods can be more robust against the choice of learning rates or step sizes, as they inherently adjust the step size based on the curvature of the function.

Second-order methods are valuable tools for optimization, particularly when computational resources allow for the calculation of second-order derivatives. However, their application may depend on the specific characteristics of the problem being solved.

An example of a second-order method is Newton's method. This method directly minimizes a quadratic approximation of the objective function using the Hessian matrix. Newton's method is a classic optimization algorithm used to find the minimum or maximum of a function. Named after Sir Isaac Newton, this method utilizes both the gradient (first derivative) and the second derivative (Hessian matrix) of the function to iteratively approach the optimal solution.

This approach adopts the structure of Newton's method, which is commonly used to locate the roots of a function, but in this case, it's applied to the derivative of function *f*. The rationale behind this is that identifying the minimum point of function f is equivalent to finding the root of its first derivative f'. In this scenario, the updated formula can be expressed as follows:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

In the given equation, we have the following:

- $f'(xn)$ represents the first derivative of function f
- $f''(xn)$ signifies the second derivative of function f

Second-order methods are typically favored over gradient descent due to their faster convergence rate, provided that the analytical expressions for both first and second derivatives are available. However, these methods converge regardless of whether they're approaching minima or maxima. Variations of this method exist that ensure global convergence and reduce computational costs by sidestepping the need to solve the system for determining the search direction using direct methods.

Summary

In this chapter, we gained insight into simulating typical human brain activities using ANNs. We grasped the fundamental concepts behind ANNs, delving into the creation of a basic neural network architecture. This exploration encompassed elements such as input, hidden, and output layers, connection weights, and activation functions. Our understanding extended to crucial decisions regarding hidden layer count, node quantity within each layer, and network training algorithms.

Then we focused on data fitting and pattern recognition using neural networks. We engaged in script analysis to master the utilization of neural network functions via the command line. We then ventured into the Neural Network Toolbox, featuring algorithms, pre-trained models, and apps for crafting, training, visualizing, and simulating shallow and deep neural networks. The Neural Network Toolbox offers an accessible interface—the **Neural Network** getting started GUI—which serves as the launchpad for tasks such as neural network fitting, pattern recognition, clustering, and time-series analysis.

Finally, we introduced some of the commonly used advanced optimization techniques. These methods are potent techniques employed to amplify the efficiency and efficacy of optimization algorithms. Their purpose is to surmount the constraints posed by conventional optimization methodologies, especially in intricate, high-dimensional, or non-convex optimization scenarios.

In the next chapter, we will understand the basic concepts of deep learning. We also will learn about the different types of deep learning and understand **convolutional neural networks** (**CNNs**). Additionally, we will learn how to build a CNN using MATLAB and understand recurrent neural networks, long short-term memory networks, and transformer models.

6

Deep Learning and Convolutional Neural Networks

Deep learning (**DL**) is a **machine learning** (**ML**) technology based on multilayer **artificial neural networks** (**ANNs**) that has allowed many applications to reach a high degree of accuracy. **Deep NNs** (**DNNs**) are capable of modeling and have the ability to capture intricate connections between input and output information. Among the highly effective uses, **computer vision** (**CV**) stands out, encompassing activities such as categorization, image regression, and the identification of objects. As an illustration, an advanced NN can produce a stratified portrayal of entities, wherein each entity is recognized by a collection of features taking the shape of visual basics, such as specific contours, directed lines, surface details, and repetitive designs. **Convolutional networks** (**CNNs**) are characterized by convolutional layers, which use filters to analyze data in a local region and produce an activation map. These activation maps are then processed by pooling layers, which aggregate the low-resolution data to reduce the dimensionality of the representation and make processing more computationally efficient. The convolutional and pooling layers are then alternated several times until the image is represented by a low-resolution activation map. In this chapter, we will learn the basic concepts of DL and discover how to implement an algorithm based on CNNs in the MATLAB environment.

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

- Understanding DL basic concepts
- Exploring DL models
- Approaching CNNs
- Building a CNN in MATLAB
- Exploring the model's results
- Discovering DL architectures

As always, I thank the technical reviewers for their helpful advice on improving my chapter. I will try to apply them as they were proposed to me.

Technical requirements

In this chapter, we will introduce ML basic concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. A working knowledge of the MATLAB environment is also required.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `CNNPistachioClassification.m`
- `PistachioShort.zip`

Understanding DL basic concepts

DL is a branch of ML based on using algorithms to model high-level abstractions about data. This discipline is part of a range of approaches that aim to learn methods for representing data. For example, an observation such as an image can be described in different ways: as a vector of intensity values for each pixel, or more abstractly as a set of edges or regions that have shapes or significant features. Some of these possible representations may prove more effective than others in facilitating the process of training another ML system.

For automatically identifying and extracting relevant features from raw data, we can use automated feature extraction to eliminate the need for manual **feature engineering** (**FE**). This process streamlines ML tasks and improves model performance.

Automated feature extraction

In this context, one of the central aspects of DL is the development of learning algorithms that specialize in automatically extracting significant features from a dataset, which can then be used to train ML systems. This is a significant contribution, considering that without such techniques, these features would have to be manually generated and vetted prior to training. The basic concept of DL is to successively subject the input data to different levels of processing, the result of which is the emergence of features. Consider an image of a cat. Automated feature extraction can identify features such as the cat's eyes, ears, and whiskers.

In the field of NNs, the concept of DL was introduced with the creation of so-called **DNNs**. The operating principle is like that of traditional NNs but with one obvious difference: the significant increase in the number of intermediate layers of hidden neurons. As with classical NNs, DNNs are capable of modeling complex relationships between input and output data.

One of the most successful areas of application is CV, which includes tasks such as classification, image regression, and object detection. In the latter task, a DNN can build a layered representation of objects, in which each object is identified by a set of features that appear in the form of primitive

visual elements, such as specific edges, oriented lines, textures, and recurring patterns. This modeling ability is rooted in a large number of hidden layers of neurons.

Training a DNN

As far as the training phase is concerned, it is still possible to use the well-established backpropagation algorithm. As with traditional NNs, DNNs can suffer from the age-old problem of overfitting. To mitigate this situation, **regularization techniques** are usually employed, which intervene in the optimization process during the training phase. Among the most frequently employed methods are the following:

- **L2 regularization** (**weight decay**), which influences the optimizer's operation by adding the **L2 norm** of the network weights, multiplied by a specific constant, to the loss function. The L2 norm, also known as the Euclidean norm, is a mathematical function that measures the length of a vector. It is defined as the square root of the sum of the squares of the vector's components. The L2 norm is often used in ML and statistics to measure the distance between two vectors.

  The formula for the L2 norm of a vector x is as follows:

  $$||x||2 = sqrt(\Sigma(x_i)\wedge 2)$$

  Please refer here as follows:

  - $||x||_2$ is the L2 norm of x
  - x_i is the i-th component of x

- **L1 regularization** (**sparsity**), functioning similarly to the former but using the **L1 norm**. The L1 norm is often used in ML and statistics to measure the distance between two vectors. It is also known as the Manhattan norm or the Taxicab norm because it corresponds to the distance traveled by a taxicab moving along city streets. The formula for the L1 norm of a vector x is:

  $$||x||1 = \Sigma|x_i|$$

  Here:

  - $||x||_1$ is the L1 norm of x
  - x_i is the i-th component of x

- **Dropout**, where, during each training step, a set number of randomly chosen neurons within the hidden layers are turned off, preventing their outputs from affecting subsequent neurons.

All these techniques serve to lessen the interdependencies between the network and the training data samples, effectively curbing overfitting. Although backpropagation training provides a solid solution due to its simplicity and tendency to converge toward better local minima compared to other methods, it can pose considerable challenges in deep networks concerning actual computation. Indeed, an array of **hyperparameters** must be considered in DNNs, encompassing factors such as dimensions (in terms of layer count and units per layer), learning rate, initial weights, and the optimization of these parameters—a process that can become unwieldy in terms of time and computational resources.

Various solutions have been suggested in this context, including the utilization of mini-batches of data to expedite training. However, a substantial breakthrough has been the advancement of GPUs with increasingly powerful computational capabilities in recent years. In network operations, the primary computational tasks involve matrix and vector operations, which align well with parallel implementation on GPU hardware.

Another issue stemming from gradient descent-based techniques, accentuated by the intricate structure of deep networks, is the **vanishing gradient problem**. This challenge originates from gradient computation within the chain and the network's substantial number of layers. Typically used activation functions tend to generate gradients with minute values, typically within the range of [-1, 1]. Due to the chain computation, this results in the multiplication of many small values when calculating the gradient in the initial layers of an *n*-level network. Consequently, the gradient decreases exponentially with *n*, causing the initial layers to learn at a sluggish pace.

Another solution provides the preliminary training in an unsupervised way of one level of the network at a time, to then carry out a final complete training by means of backpropagation. Another way around the problem, encouraged by the development of GPUs in recent times, is the use of faster hardware to counteract what is the main symptom of the problem—namely, the slowness of the training process. Another point against DNNs, in the case of **supervised learning** (**SL**), is represented by the very large amount of sample data (including the desired output) necessary for the network to reach the required results at the end of the training. This represents a significant obstacle since, for certain tasks, the production of the expected outputs for each example is an operation that can take a very long time.

After introducing DL, let's now see the different types of DL available.

Exploring DL models

Various types of DL architectures and techniques have been developed to tackle different tasks and challenges. Here are some examples:

- **CNNs**: Mainly used for image and video analysis, CNNs are designed to learn spatial hierarchies of features automatically and adaptively from input data. They have been highly successful in tasks such as image classification, object detection, and image segmentation.
- **Recurrent NNs** (**RNNs**): RNNs are well suited for tasks involving sequences, such as **natural language processing** (**NLP**) and speech recognition. They have an internal memory that allows them to maintain information about previous inputs, making them effective for handling sequential data.
- **Long short-term memory** (**LSTM**) **networks**: A type of RNN, LSTMs are designed to overcome the vanishing gradient problem in training deep networks. They are particularly useful for capturing long-range dependencies in sequential data, making them popular for tasks such as language modeling and machine translation.

- **Generative adversarial networks** (**GANs**): GANs consist of two NNs—the generator and the discriminator—engaged in a game-like setting. GANs are used for generating new data samples that are like a given dataset. They have been used for image generation, style transfer, and data augmentation.
- **Autoencoders** (**AEs**): AEs are used for **unsupervised learning** (**UL**) and data compression. They consist of an encoder that maps the input data to a latent space and a decoder that reconstructs the input from the latent representation. AEs find applications in dimensionality reduction, denoising, and anomaly detection.
- **Transformer networks**: Transformers have gained popularity in NLP tasks. They utilize self-attention mechanisms to process input data in parallel, making them efficient for handling long-range dependencies in sequences. The **Bidirectional Encoder Representations from Transformers** (**BERT**) model is a prominent example.
- **Capsule networks**: Capsule networks aim to improve the robustness of NNs to variations in object poses and viewpoints. They use *capsules* to represent different aspects of an object, allowing them to capture hierarchical relationships between features.
- **Neural Turing machines** (**NTMs**): These architectures combine NNs with external memory, enabling them to perform algorithmic tasks. They are designed to learn algorithmic procedures and have been used for tasks such as sorting and associative recall.
- **Graph NNs** (**GNNs**): GNNs operate on graph-structured data, making them suitable for tasks such as node classification, link prediction, and graph-level classification. They have applications in social network analysis, molecular chemistry, and recommendation systems.

These are just a few examples of the diverse range of DL architectures and techniques available, each tailored to specific tasks and data types. For example, CNNs are designed for image and spatial data analysis using convolutional layers to automatically learn hierarchical features and patterns from images. Characteristic claims are image classification, object detection, and image segmentation. RNNs are suited for sequential and temporal data analysis, maintaining the memory of previous inputs through recurrent connections and enabling them to capture sequential patterns. Typical applications are NLP, speech recognition, and time-series prediction. LSTM networks are an improved version of RNNs to address the vanishing gradient problem, incorporating memory cells to selectively retain and update information over long sequences. Applications of these technologies provide language modeling, machine translation, and **sentiment analysis** (**SA**). GANs are used to generate new data samples that resemble a given dataset using a generator and a discriminator network that compete in a minimax game. This type of DL is particularly suitable for image generation, style transfer, and data augmentation. Finally, AEs as UL for data compression and feature extraction comprise an encoder to map input data to a lower-dimensional latent space and a decoder to reconstruct input data. Dimensionality reduction, denoising, and anomaly detection are representative applications for these algorithms.

The field of DL continues to evolve, with new architectures and approaches emerging to address various challenges across different domains.

At this point, we can delve into the main technology of DL—namely, CNNs.

Approaching CNNs

As outlined in *Chapter 5*, *Introducing Artificial Neural Networks Modeling,* ANNs draw their inspiration from biological NNs. These ANNs aim to replicate human cognitive processes by emulating the mechanisms observed in natural NNs. They serve the purpose of estimating or approximating functions that might rely on numerous inputs, many of which could be unfamiliar. ANNs are typically conceptualized as networks of interconnected neurons, facilitating the exchange of messages. Each connection possesses an associated weight, the value of which can be adjusted through learning from experience. This adaptive characteristic empowers NNs to accommodate diverse input types and facilitates their capacity to learn (*Figure 6.1*):

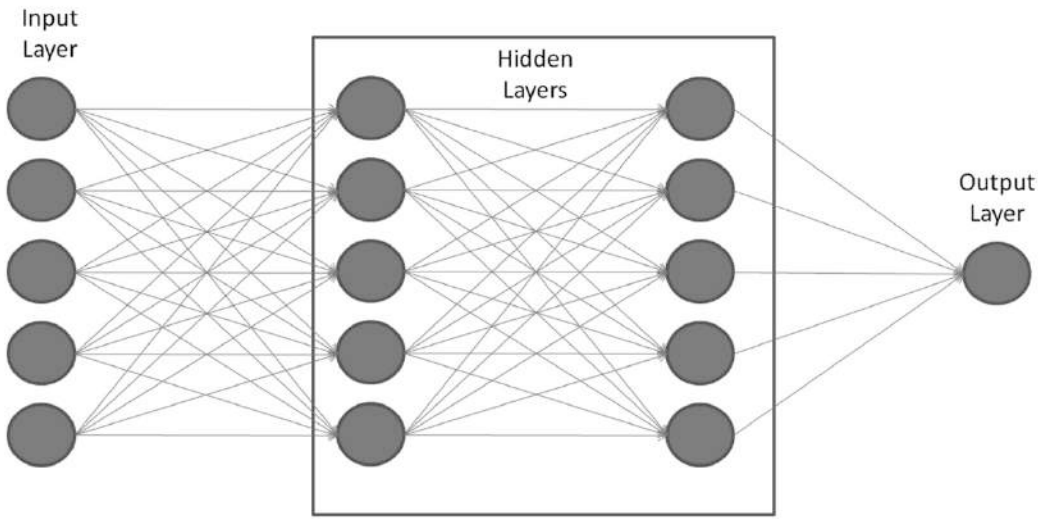


Figure 6.1 – ANN architecture with hidden layers

ANNs define a neuron as the **central processing unit** (**CPU**) that executes a mathematical operation to produce a single output from a set of input values. The neuron's output is determined by the weighted sum of inputs and an added bias. Each neuron undertakes a straightforward task: activation occurs if the cumulative signal surpasses a specified activation threshold. The previous diagram illustrates a basic ANN structure, which essentially embodies CNNs. Indeed, like the latter, CNNs consist of interconnected neurons linked by weighted branches (weights); the networks' training parameters remain the weights and biases.

Within CNNs, the inter-neuron connectivity pattern takes inspiration from the arrangement observed in the animal world's visual cortex. Neurons within this section of the brain (visual cortex) react to specific stimuli within limited observation zones referred to as receptive fields. These fields partially overlap to encompass the entire field of vision. The response of an individual neuron to stimuli within its receptive field can be approximated mathematically through a convolution operation.

All aspects pertaining to NN training, encompassing forward/backward propagation and weight updates, hold true in this context. Additionally, an entire CNN utilizes a singular differentiable cost function. However, CNNs are based on a particular assumption: their input possesses a specific data structure, such as an image. This characteristic empowers them to incorporate tailored architectural elements for enhanced processing of such data.

CNNs adopt a **fully connected** (**FC**) **architecture**, where each neuron in each layer connects to all neurons in the preceding layer (excluding bias neurons). Generally, these architectures do not scale effectively when confronted with expanding input data sizes.

The most common layers in a CNN are:

- Convolutional layer
- Pooling layer
- **Rectified linear units** (**ReLUs**)
- FC layer

The function of each layer is addressed in detail in the following subsections.

Usually, a CNN is composed of multiple sequential levels of convolution and subsampling (pooling), which are subsequently succeeded by one or more FC final levels, especially in the context of classification tasks.

For addressing real-world challenges, these procedures can be merged and repeated as required. For instance, you can incorporate two, three, or even multiple layers of convolution. Additionally, you have the flexibility to apply pooling operations repeatedly to downsize the data dimensions. As mentioned earlier, a variety of levels are typically employed within a CNN. Certain layers encompass adjustable training parameters (weight and bias), whereas others are designed to execute predefined functions.

Convolutional layer

In a CNN, the primary layer type is the **convolutional layer**, and including one or more of these layers within a CNN is indispensable. In practical terms, the parameters of a convolutional layer are tied to a collection of adaptable filters. In contrast to CNNs, convolutional layers are structured in three dimensions, encompassing width, height, and depth.

Every filter occupies a small spatial area (along width and height dimensions) while spanning the complete depth of the input volume it is applied to. In forward propagation, each filter is shifted—or, more precisely, convolved—across the input volume's width and height, generating a two-dimensional activation map (also known as a feature map) specific to that filter. While the filter traverses the input region, a scalar product operation takes place between the filter values and those of the corresponding input section.

Conceptually, the network's objective is to learn activated filters that detect specific functionalities within distinct spatial regions of the input. Combining all these feature maps (for every filter) along the depth dimension forms the output volume of a convolutional layer. Each element within this volume can be interpreted as the output of a neuron observing a confined input area, and it shares its parameters with other neurons within the same feature map. This sharing stems from their common origin—the application of the same filter.

The mathematical steps for filtering are as follows:

1. Align the filter and the image patch.
2. Multiply each image pixel by the corresponding feature pixel.
3. Sum the products.
4. Divide each sum by the total number of pixels in the feature.

Here is the formula for the output of a convolutional layer:

$$\text{Output} = \Sigma F(k, l) * I(i + k, j + l)) / S(k, l)$$

Here:

- $F(k, l)$ is the feature map element in (k, l)*position*
- $I(i + k, j + l))$) is the image element in$(i + k, j + l)$position
- $S(k, l)$ is the sum of the feature pixel

In summary, key points to emphasize include:

- **Local receptive field**: Neurons in a layer are connected to a small segment of the input, known as a local receptive field. Each connection learns a weight.
- **Shared weights**: Since significant features (such as edges or blobs) can appear anywhere in an image, neurons within the same layer share weights. This implies that identical features are recognized by all neurons in the layer, even if they're positioned at different input points.
- **Convolution**: Identical weight patterns are employed at various positions. The outcome of convolution is termed a **feature map**.

Each filter captures a specific feature from the preceding layer. Therefore, training multiple convolutional filters is necessary to extract diverse features. Each filter produces a feature map highlighting distinct characteristics.

Pooling layer

These layers are systematically integrated into a network to periodically diminish the spatial dimensions (width and height) of ongoing representations and volumes within a specific stage of the network. This serves a dual purpose of minimizing the number of parameters, reducing computational time, and keeping a check on overfitting. The role of a **pooling layer** is to operate on each depth slice of the input volume independently to resize it spatially.

For each feature obtained during the convolutional process, a matrix is constructed, and the maximum value within each selected matrix is identified to compress the entire input. The steps are as follows:

1. Choose a window size (typically `2` or `3`).
2. Select a stride distance for pixel movement (usually 2). In a CNN, the stride distance in a pooling layer determines how much the pooling operation slides across the input feature map. A higher stride distance means that the pooling operation will skip over more pixels, resulting in a smaller output feature map. A lower stride distance means that the pooling operation will slide more carefully across the input feature map, resulting in a larger output feature map.
3. Slide the window across the filtered images.
4. For each window, determine the maximum value.

Illustratively, this method partitions an input image into groups of squares, and for each resulting segment, the highest value is extracted as the output. CNNs additionally incorporate pooling layers, strategically placed right after the convolutional layers. A pooling layer dissects the input into sections and chooses a representative value, employing methods such as max pooling or average pooling. The **max pooling** layer captures the highest features detected by preceding convolutional layers. The output layer assesses the presence of a potential feature within a region of the previous layers, albeit without exact spatial coordinates. The objective is to enable subsequent layers to process larger data sections. Max pooling expedites convergence rates, thereby enabling the selection of highly invariant features that enhance generalization performance.

The integration of a pooling layer offers the following benefits:

- Reduces subsequent layer computations
- Enhances feature robustness concerning spatial positioning
- Operates under the premise that after identifying a particular feature, its precise location within the input becomes less significant than its general position relative to other features

Average pooling serves as a downsampling operation that consolidates the representation of features within patches of the feature map. Typically, it is implemented in 2x2 patches of the feature map with a stride of (2, 2). The process entails computing the average for each patch of the feature map, indicating that every 2x2 square in the feature map is downsampled to its average value. To illustrate, consider the 6x6 feature map resulting from the line detector convolutional filter discussed in the preceding section. We can manually explore the application of the average pooling operation to the initial line of that feature map.

In a typical CNN architecture, convolution layers and pooling layers are interleaved in a recurring pattern.

ReLUs

ReLUs serve as crucial activation functions within NNs. Their frequent utilization occurs multiple times within a single network, often following each convolutional layer. A ReLU layer is composed of neurons that execute the function $f(x) = max(0, x)$. Incorporating these layers enhances the network's non-linearity while maintaining the convolutional levels' receptive fields unchanged.

ReLUs are favored over alternative functions such as hyperbolic tangent or sigmoid, primarily due to their capacity to accelerate the training process significantly without substantially compromising generalization accuracy. By employing ReLU layers, network training becomes remarkably swifter, all while preserving comparable performance.

These layer types lack adjustable parameters, thus executing a fixed function. Furthermore, they don't possess any adaptable parameters either. Layers devoid of tunable parameters simplify the backward propagation process: errors computed up to that point, stemming from the subsequent layer, are propagated backward to the preceding layer.

FC layer

This layer type mirrors the structure of any layer found in a traditional ANN featuring an FC architecture. In an **FC layer**, each neuron establishes connections with all neurons from the preceding layer, specifically interacting with their activations.

Unlike what has been observed thus far in CNNs, this layer type does not adhere to the principle of local connectivity. An FC layer establishes connections spanning the entire input volume, which naturally results in a multitude of connections. The sole adjustable parameter in this layer type is the number of neurons, K, that constitute it. The fundamental characteristic of an FC layer can be summarized as follows: linking its K neurons with the entire input volume and calculating the activation for each of these K neurons.

In practice, the outcome will be a singular 1 x 1 x K vector, encapsulating the computed activations. The transition from an input volume, organized in three dimensions, to a singular output vector in one dimension (1 x 1 x K) after implementing an FC layer signifies that the utilization of additional convoluted layers becomes infeasible. In the realm of CNNs, FC layers primarily serve the purpose of

consolidating information amassed up to that point, presenting it as a singular value (the activation of one of its neurons). This singular value subsequently forms the basis for subsequent computations in the final classification process.

Having thoroughly examined each element of a CNN, it is now fitting to delve into the overarching structure of a complete CNN.

Building a CNN in MATLAB

In this section, we will see how to train a CNN for image classification, starting from the images as input layers. The overall architecture of a CNN typically comprises a sequence of convolutional layers, interspersed with ReLU layers and, when appropriate, standardization and pooling layers. Ultimately, the network concludes with a series of FC layers leading to the output layer.

CNNs consist of two primary types of neurons:

- **Processing neurons**: These neurons undertake the responsibility of processing specific sections of the input image through convolution functions. Their primary role involves extracting distinctive features from the input data.
- **Aggregation or pooling neurons**: These neurons aggregate the input data and reduce its dimensions through subsampling, enhancing efficiency for subsequent layers' processing.

By assembling the output values from a given layer, it becomes possible to reconstruct an intermediate image that serves as a foundation for subsequent layers' operations:

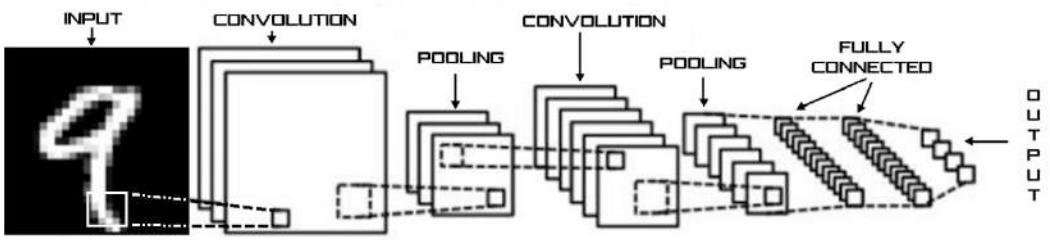


Figure 6.2 – CNN architecture

The fundamental concept involves commencing with a sizable image and progressively condensing the data through incremental steps until a singular outcome is achieved. As the number of convolutional passages increases, the NN's capacity to comprehend and handle intricate functions amplifies.

The configuration of a basic CNN can be succinctly described through the following components:

- An input layer tasked with acquiring input elements, such as images
- A sequence of convolutional layers, occasionally punctuated by a ReLU layer and, when deemed appropriate, pooling layers

- A series of FC layers
- An output layer that furnishes the final results

Recent research has indicated that the significance of FC layers might be less substantial. Nevertheless, for the present context, the outlined structure represents the typical architecture of a CNN.

Object recognition refers to the capacity to locate a specific object within a series of images or videos. Humans effortlessly identify diverse objects in images, even when those objects appear differently. Moreover, objects can be distinguished even when only parts of them are visible. However, this remains a challenging task for the broader field of CV.

In each image, every object boasts numerous intriguing features that can be extracted to craft a depiction of it. This portrayal subsequently serves to distinguish the object when attempting to spot it amid multiple objects in a test image. Ensuring the features taken from the reference image are unaffected by differences in image scale, disturbances, lighting variations, and distortions is crucial for establishing dependable recognition.

CNNs are particularly well suited for this endeavor, providing algorithms that excel in the identification of objects with remarkable accuracy. This is because they are able to learn features from images that are invariant to translation, scale, and rotation. This is due to the use of convolutional layers, which apply a filter to a small region of the input image and then slide the filter across the image. This allows the network to learn features that are specific to different parts of the image, regardless of where they are located. Additionally, pooling layers are used to reduce the dimensionality of the feature maps, which makes the network more efficient to train and run.

For CNN training, we will use Pistachio image dataset. The dataset offers data for analyzing the species of pistachio using a large collection of photos.

It encompasses a total of 2,148 images categorized into 2 classes—`Kirmizi` and `Siirt`:

1. To start, we must load the dataset in the MATLAB workspace. The dataset is available in ZIP format on the site linked in the *Technical requirements* section. To facilitate the reader's work, the images have been divided according to class into folders named according to the category and resized to reduce the weight of the data. These folders are available for download on the book's GitHub site. Once the file has been downloaded and the contents extracted, it will be enough to define the path of the root folder. At this point, it will be possible to upload the data to the MATLAB workspace as follows:

```
Data = imageDatastore('C:\MatlabScript\PistachioShort',
'IncludeSubfolders',true,'LabelSource','foldernames');
```

To load the data, we used the `imageDatastore()` function, which employs an `ImageDatastore` entity to oversee a compilation of image files, with each singular image fitting within memory, although the complete image collection might not. Constructing an `ImageDatastore` entity involves utilizing the `imageDatastore` function, where you can define its attributes. Subsequently, you can import and manipulate the data using the functions associated with the entity. Three parameters were used, as follows:

- `Location= 'C:\MatlabScript\PistachioShort'`: This is the location of the dataset. To replicate the example, the reader has to simply substitute the path of the folder in which the dataset was stored.
- `IncludeSubfolders`: The inclusion of subfolders can be controlled using the `IncludeSubfolders` name-value argument, which accepts either `true` or `false`. Opt for `true` to encompass all files and subfolders within each main folder, or select `false` to solely encompass files within each main folder. When `IncludeSubfolders` is not explicitly specified, the default value is `false`.
- `LabelSource`: The origin of label data is determined through the `LabelSource` name-value argument, which accepts either `none` or `foldernames`. By specifying `none`, the `Labels` property remains devoid of labels. On the other hand, designating `foldernames` results in the assignment of labels based on the folder names, which are then stored within the `Labels` property. Subsequent adjustments to the labels can be made by directly accessing the `Labels` property. It's important to note that the `LabelSource` name-value argument cannot be utilized when a `FileSet` object serves as the file or folder location.

We can now view some of the images loaded through a random process:

```
figure;
RandId = randperm(2148,9);
for i = 1:9
    subplot(3,3,i);
    imshow(Data.Files{RandId(i)});
end
```

In this piece of code, we have used the `randperm()` function, which generates a row vector comprising `9` distinct random integers chosen from the range of `1` to `2148`. Each number generated was used as an index to identify an image file path stored in the `Data.Files` property.

The following output was returned:

Figure 6.3 – Image samples from PistachioShort dataset

We are now able to determine the image count in each category. The `ClassItemsNumber` variable represents a table containing the labels and the corresponding number of images associated with each of them:

```
ClassItemsNumber = countEachLabel(Data)
```

We used the `countEachLabel()` function that produces a summary table indicating the labels within the `Data` dataset and the respective count of associated files for each label.

The following table was printed:

```
ClassItemsNumber =
  2×2 table
     Label      Count
    ________    _____
    Kirmizi     1232
    Siirt        916
```

As we can see, there are two categories, and there are enough images in each category.

2. To start, it is necessary to divide the data at our disposal into two subsets; the first will be used for training, and the second for algorithm validation. Data splitting is a crucial step in ML and data analysis whereby a dataset is divided into two or more subsets for the purpose of training, validation, and testing of a model. We have `2148` samples divided into `1232` Kirmizi species

and 916 Siirt species. Usually, the data would be split into 80% for the training and 20% for validation. For this purpose, we decide to use 700 samples for the training and the remainder for validation:

```
TrainSamples = 0.8;
[DataTrain,DataValidation] =
splitEachLabel(Data,TrainSamples,'randomize');
```

To do this, the `splitEachLabel()` function was used: this function divides the image files in the Data dataset into two distinct datastores—namely, `DataTrain` and `DataValidation`. The freshly created `DataTrain` datastore encompasses the initial `TrainSamples` value from every category, while the `DataValidation` datastore holds the remaining files from each category. `TrainSamples` can either be a fractional value between `0` and `1`, denoting the proportion of files to allocate to `DataTrain`, or an integer representing the exact count of files to be placed in `DataTrain` for each category.

3. Now, we can start building our CNN. As anticipated, a CNN is formed by a series of layers connected to each other. To get started, you need to use a layer to import your input data:

```
layers = [
    imageInputLayer([64, 64, 3])
```

The `layers` variable is an array containing a list of layers of the CNN and defines the architecture of NNs for DL. As a first layer, we set the `imageInputLayer` layer: this layer serves as an image input, feeding 2D images into a NN while also implementing data normalization. This layer provides an image input layer and defines the unchangeable `InputSize` attribute.

The `InputSize` attribute is a depiction of the input data" dimensions, represented as a 1D array of integers `[h w c]`, where `h`, `w`, and `c` denote the height, width, and number of channels respectively. In this case, we have an RGB image with height = 64 and width = 64.

After the input layer, we set the first block of three layers in sequence:

```
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
```

Filter size *3x3*, number of filters = 8, and padding = `'same'` indicate the convolution process would yield the same size of input image (*64x64x3*).

In this block, we have:

- `convolution2dLayer`: A 2D convolutional layer utilizes sliding convolutional filters on 2D input data. This layer performs convolution by shifting the filters across the input data in both vertical and horizontal directions. It calculates the dot product between the filter weights and the input at each position and subsequently incorporates a bias term.

- `batchNormalizationLayer`: A layer for batch normalization standardizes a mini-batch of data independently across all observations for each channel. To accelerate the training process of a CNN and decrease vulnerability to network initialization, incorporate batch normalization layers between convolutional layers and non-linear functions such as ReLU layers.
- `reluLayer`: A ReLU layer applies a thresholding operation to each input element, nullifying any value that is less than zero and setting it to zero.

After this first block, we have to apply a pooling layer as follows:

```
maxPooling2dLayer(2,'Stride',2)
```

Downsampling at a 2:1 ratio and stride = 2 indicate a shift by 2 columns and 2 rows in the pooling operation. A 2D max pooling layer conducts downsampling by partitioning the input into rectangular pooling regions and subsequently determining the maximum value within each region.

4. Now, we will apply a second block of layers like the first, changing the parameters:

```
convolution2dLayer(3,16,'Padding','same')
batchNormalizationLayer
reluLayer
```

5. After that, a new pooling layer is applied:

```
maxPooling2dLayer(2,'Stride',2)
```

6. And finally, a third block of layers:

```
convolution2dLayer(3,32,'Padding','same')
batchNormalizationLayer
reluLayer
```

Now, we need an FC layer:

```
fullyConnectedLayer(2)
```

In an FC layer, the input is multiplied by a weight matrix and then augmented with a bias vector. A bias vector is a set of values added to the weighted input of a neuron before applying the activation function. It plays a crucial role in adjusting neuron activation thresholds and enabling the network to represent complex relationships in data. The `Size` parameter output specifies the output size; we are classifying a two-species pistachio, so the size is `2`. This layer category replicates the configuration of layers typically present in a conventional ANN that employs an FC design. Within an FC layer, each neuron establishes connections with every neuron from the previous layer, engaging directly with their respective activations.

Then, we need a softmax layer:

```
softmaxLayer
```

`softmaxLayer` is a type of layer used in NNs, specifically designed to apply the `softmax` function to the input. The `softmax` function is often used in classification tasks to convert raw scores or logits into a probability distribution over multiple classes. This layer is commonly utilized as the final layer in a NN for multi-class classification, where the output values are transformed into probabilities that sum up to `1`, making it easier to interpret the model's predictions.

Finally, to close the network, a classification layer was applied:

```
  classificationLayer];
```

The classification layer computes cross-entropy loss for both standard and weighted classification tasks involving mutually exclusive classes. The layer automatically determines the number of classes based on the dimension of the output from the preceding layer.

Right now, we have to set the option of the CNN:

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',50, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

The `trainingOptions()` function was used to set several options relevant to the training of the network. The following options were set:

- `sdm`: Solver for training the NN; we have used the **stochastic gradient descent with momentum** (**SGDM**) optimizer. The momentum optimizer is a modified version of the gradient descent algorithm. It is designed to accelerate the optimization process by incorporating momentum into the weight updates.
- `InitialLearnRate`: The initial learning rate employed in training is a positive numerical value. When the learning rate is excessively low, the training process might become prolonged. Conversely, if the learning rate is excessively high, the training could yield suboptimal results or diverge.
- `MaxEpochs`: The maximum number of training epochs is defined as a positive integer. In the context of gradient descent optimization using mini-batches, an iteration signifies a single step aimed at minimizing the loss function. An epoch refers to the full iteration of the training algorithm across the entire training dataset.
- `Shuffle`: Option for data shuffling. For this, we employed the practice of shuffling the training data before every training epoch, and similarly, shuffling the validation data before each validation of the NN.
- `ValidationData`: The validation data to be utilized during training is defined as either a datastore, a table, or an array of cells containing the predictors and validation responses.

- `ValidationFrequency`: The NN's validation rate is expressed as a positive integer, representing the count of interactions. `ValidationFrequency` signifies the interval, in terms of iterations, at which validation metrics are assessed.
- `Verbose`: For displaying training progress information.
- `Plots`: Upon configuring the `training-progress` training option in `trainingOptions` and initiating the network training using `trainNetwork()`, a plot is generated. This plot showcases training metrics for each iteration. Each iteration involves an estimation of the gradient and an adjustment of the network parameters. If you provide validation data within `trainingOptions`, the plot will additionally present validation metrics whenever the network undergoes validation by the `trainNetwork` process.

7. Finally, we have to train the network:

```
Cnnet = trainNetwork(DataTrain,layers,options);
```

The `trainNetwork()` function facilitates the training of DNNs. This training can be executed on either a CPU or a GPU. For tasks such as image classification and image regression, it's possible to train a single NN concurrently using multiple GPUs or a local/remote parallel pool. However, the employment of GPUs or parallel processing necessitates the presence of Parallel Computing Toolbox. Additionally, utilizing a GPU for DL mandates the availability of a compatible GPU device.

The following plot was returned to the `trainNetwork()` function:

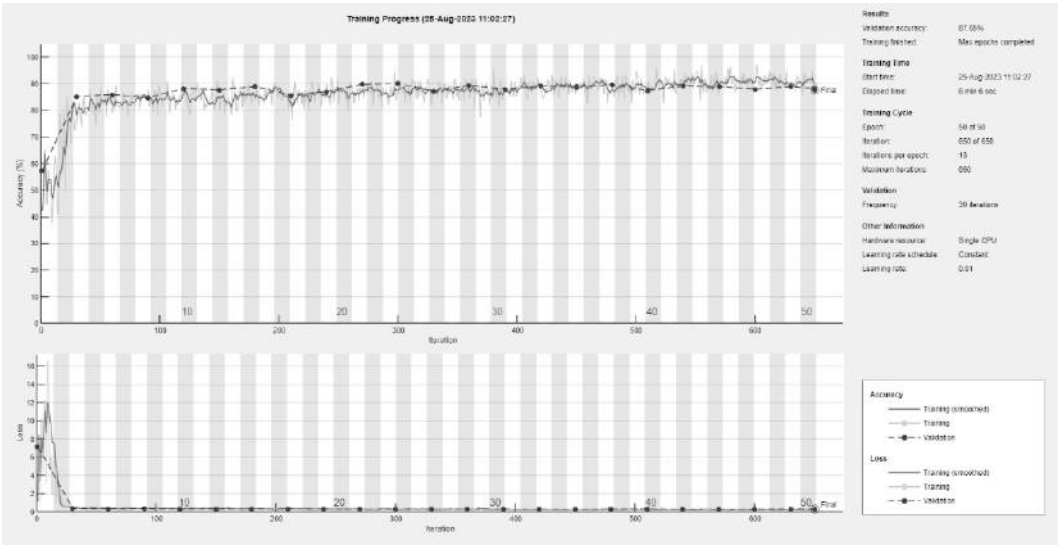


Figure 6.4 – Training process

The plot in *Figure 6.4* will be progressively updated throughout the training process. In this way, it will be possible to check how the algorithm manages to adjust the weights to reach convergence. The plot in *Figure 6.4* consists of three curves:

- **Training error**: This curve shows the average error of the network on the training data as a function of the number of training epochs. Epochs are the number of times the network is trained on the entire training dataset. A low training error indicates that the network is learning well from the training data.
- **Validation error**: This curve shows the average error of the network on the validation data as a function of the number of training epochs. Validation data is data that the network was not trained on, and it is used to evaluate how well the network generalizes to unseen data. A low validation error indicates that the network is learning to generalize well to new data.
- **Network weights**: This curve shows the magnitude of the weights of the network as a function of the number of training epochs. The weights are the parameters that the network learns from the training data. A stable weight curve indicates that the network is not overfitting to the training data.

After seeing how to train a CNN in MATLAB, we now need to learn how to interpret the results and use the validation metrics correctly.

Exploring the model's results

Evaluating results is an essential part of any CNN implementation process. This, of course, is true for any algorithm based on ML. Evaluation metrics are quantitative measures used to assess the performance and quality of a model, algorithm, or system in various tasks, such as ML, data analysis, and optimization. These metrics provide a way to objectively quantify how well a model is performing and to compare different models or approaches.

The type of metric to adopt obviously depends on the type of algorithm we are implementing; in the previous section, we implemented a CNN for the classification of the pistachio species. So, let's take a look at the metrics available for this type of algorithm.

For a classification task, we can use the following metrics:

- **Accuracy**: The proportion of correctly classified instances out of the total instances
- **Precision**: The ratio of true positive predictions to the total number of positive predictions, indicating the accuracy of positive predictions
- **Recall (Sensitivity)**: The ratio of true positive predictions to the total number of actual positives, indicating the model's ability to identify positive cases
- **F1-score**: The harmonic mean of precision and recall, providing a balance between them

- **Receiver Operating Characteristic (ROC) curve**: A graph showing the trade-off between true positive rate and false positive rate at various thresholds
- **Area Under the ROC Curve (AUC-ROC)**: A metric that quantifies the overall performance of a binary classification model

Let us try to deepen the concept of accuracy. Accuracy is a fundamental evaluation metric used in classification tasks to measure the proportion of correctly predicted instances out of the total instances in a dataset. It provides an overall view of how well a classification model is performing. The accuracy formula is:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

For example, if you have a binary classification problem with 100 instances, and your model correctly predicts 90 of them, the accuracy would be:

$$\text{Accuracy} = \frac{90}{100} = 0.90, \textit{ or } 90\%$$

While accuracy is a simple and intuitive metric, it might not be suitable in cases of imbalanced classes, where one class significantly outnumbers the other. In such situations, a high accuracy could be misleading if the model is performing well on the dominant class but poorly on the minority class. In these cases, precision, recall, F1-score, and other metrics might provide a more accurate representation of the model's performance. Also, in this case, it is important to point out that when the dataset is not sufficiently balanced between the classes, the classification result may not be applied to all the classes.

So, let's try to apply this metric to the case study analyzed in the previous section for the classification of the pistachio species. We have already trained the network, and at this point, we can use it to make classifications using data never seen before by the algorithm:

1. To carry out the validation, we start by classifying the collected images in the `DataValidation` subset:

   ```
   CNNPredLabel = classify(CNnet,DataValidation);
   ```

   This function employs a trained DNN to classify data. Predictions can be generated using the trained network on either a CPU or GPU. Utilizing a GPU necessitates both a *Parallel Computing Toolbox™ license* and a compatible GPU device. Two parameters are passed: the CNN trained (`CNnet`) and the input dataset (`DataValidation`).

2. After that, we must extract the label of the input data; this information is contained in the image datastore. We have only to use the `labels` parameter, as follows:

   ```
   DataValLabel = DataValidation.Labels;
   ```

3. Let's calculate the accuracy using the formula previously introduced:

   ```
   ACC = sum(CNNPredLabel == DataValLabel)/numel(DataValLabel)
   ```

The following result is returned:

```
ACC =
    0.8695
```

An accuracy of 86.95% was obtained; this tells us that the CNN-based algorithm can correctly classify 86.95% of the pistachio images we passed it. We also recall that these images had never been used by the network training algorithm.

4. To better understand the performance of the model, we can draw a confusion matrix. A confusion matrix is a tabular representation used in the field of ML and statistics to evaluate the performance of a classification model. It provides a comprehensive view of how well the model's predictions align with the actual classes in a dataset.

   A confusion matrix is organized into four categories:

   - **True Positives** (**TP**): Instances that are correctly predicted as positive
   - **True Negatives** (**TN**): Instances that are correctly predicted as negative
   - **False Positives** (**FP**): Instances that are incorrectly predicted as positive when they are actually negative (**Type I error**)
   - **False Negatives** (**FN**): Instances that are incorrectly predicted as negative when they are actually positive (**Type II error**)

   The matrix format looks like this:

	Predicted Positive	Predicted Negative
Actual Positive	True Positives	False Negatives
Actual Negative	False Positives	True Negatives

Figure 6.5 – Confusion matrix terms' meaning

In MATLAB, we can use the following command:

```
ConfMatrix=confusionchart((DataValLabel,CNNPredLabel))
```

This function generates a confusion matrix chart using true labels stored in `DataValLabel` and predicted labels stored in `CNNPredLabel`. It then returns a `ConfusionMatrixChart` object.

The following results are printed:

```
ConfMatrix =
  ConfusionMatrixChart with properties:
    NormalizedValues: [2×2 double]
         ClassLabels: [2×1 categorical]
```

The following output was returned:

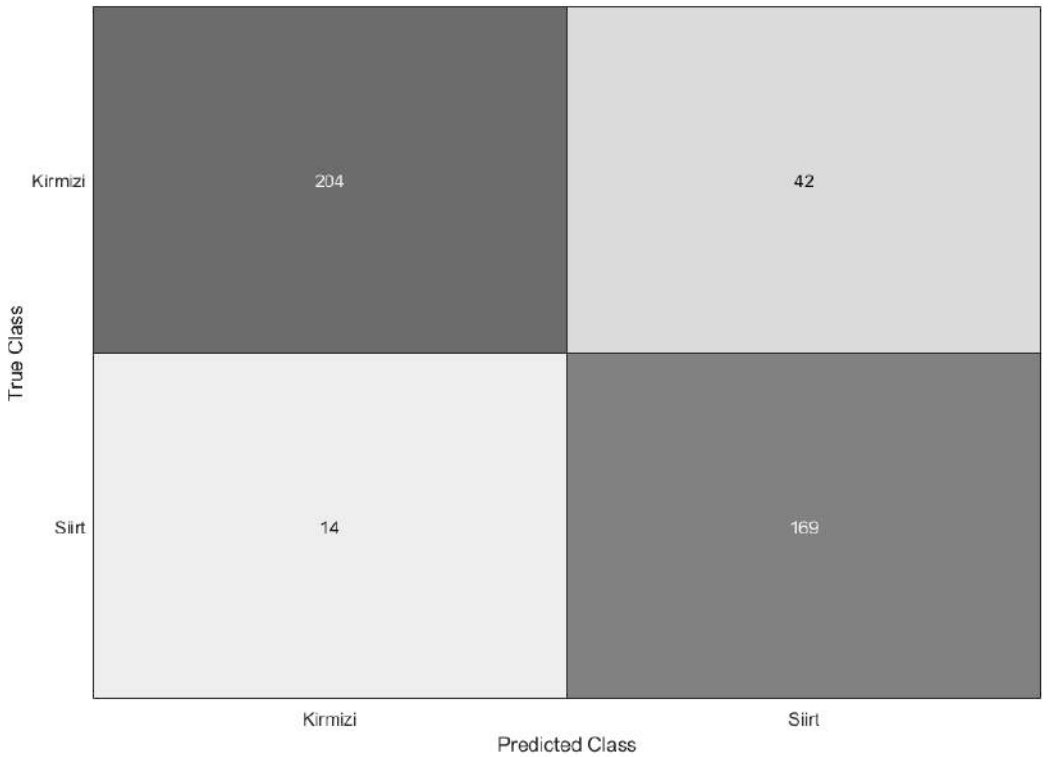


Figure 6.6 – Confusion matrix

In that matrix, rows correspond to the true classes, and columns correspond to the predicted classes. Cells along the diagonal represent correctly classified instances, while off-diagonal cells represent instances that were classified incorrectly.

We can also show the accuracy calculation from the confusion matrix table as this:

$$(TP + TN)/(Totalvalidationdata) = (204 + 169)/916 = 0.8695$$

In *Figure 6.6*, we can notice that false negatives are greater than false positives. In some applications, false negatives are more serious than false positives. For example, in the context of medical diagnosis, it is generally more important to avoid missing a disease than to incorrectly flag a healthy person as having a disease. However, in other applications, false positives may be more serious. For example, in the context of fraud detection, it is generally more important to avoid incorrectly accusing an innocent person of fraud than to miss a fraudulent transaction.

After seeing how to implement a CNN and correctly interpret the results, we can now explore some of the many DL architectures available.

Discovering DL architectures

DL models are essentially multi-layered NNs, which refers to NNs that comprise multiple hidden layers (at least two) structured hierarchically. This hierarchical arrangement facilitates the sharing and reuse of information. Across this hierarchy, one can pinpoint features while disregarding unnecessary intricacies, thereby enhancing invariance. Within the realm of multi-level ML, deeper tiers acquire inputs from the outputs of prior layers and execute more complex transformations and abstractions. This layering approach to learning draws inspiration from the information processing and learning methods of mammalian brains, enabling them to react to external stimuli.

DL architectures are the fundamental blueprints that underlie the construction of DNNs, enabling them to effectively learn and represent complex patterns and features from data. These architectures define the layout, connections, and flow of information within the network, determining how data is transformed and features are extracted at various layers. Several notable DL architectures have been developed over the years, each designed to tackle specific types of tasks and data. In the following sections, we will address the most used.

Understanding RNNs

Feedforward NNs (**FNNs**) operate by providing input data to the network that is then transformed into output. In SL scenarios, the output typically represents a label that corresponds to the input. These algorithms essentially link raw data with specific categories by recognizing underlying patterns. In contrast, recurrent networks incorporate not only the current input data being fed into the network but also information they have accumulated over time.

An RNN embodies a neural model with bidirectional information flow. Unlike feedforward networks where signal propagation only moves unidirectionally from inputs to outputs, RNNs exhibit a different behavior. In RNNs, signal propagation can traverse between neural layers, from a prior layer to a subsequent one, among neurons within the same layer, or even from a neuron to itself. A decision made by an RNN at a given moment influences its subsequent decisions. Consequently, RNNs possess two sources of input—the present and the recent past—combining to determine responses to new data, mirroring how people make decisions in everyday life.

The key distinction between recurrent and feedforward networks lies in the **feedback loop** that connects RNNs to their past decisions, momentarily utilizing their output as input. This aspect underscores the memory aspect of RNNs. The inclusion of memory in NNs serves a purpose: there is intrinsic information in the sequence itself, and RNNs leverage this information to perform tasks that feedforward networks are unable to accomplish.

Access to this memory occurs based on content rather than specific addresses or locations. One approach entails considering memory content as the activation patterns on nodes within an RNN. The concept involves initiating the network with an activation pattern that partially or noisily represents the desired memory content, allowing the network to stabilize around the required content. An RNN belongs to the category of NNs where there exists at least one feedback connection between neurons, forming a directed cycle within the network's structure.

Figure 6.7 illustrates a standard RNN with connections linking the hidden layer to the output layer:

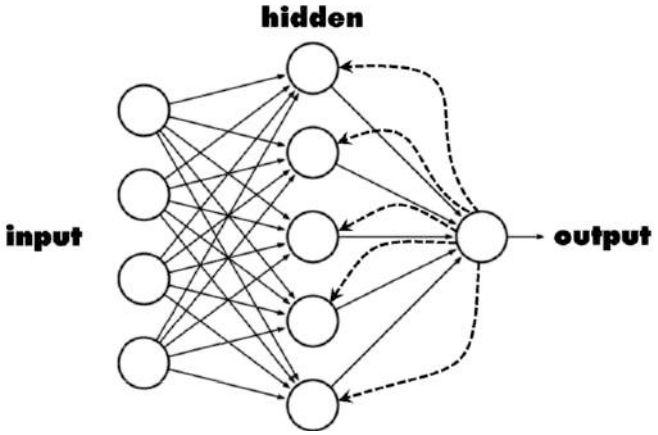


Figure 6.7 – RNN architecture

In the preceding diagram depicting a recurrent network, the hidden layer's weights are defined using both the input and output layers. Essentially, we can view RNNs as a variation of ANNs, with differences in the number of hidden layers and the flow of data patterns. RNNs exhibit distinct data flow patterns due to the cyclic connections between neurons. Unlike feedforward networks, RNNs can leverage internal memory during their computations. RNNs belong to the class of ANNs that incorporate connections between hidden layers, which are carried forward across time to facilitate the learning of sequences.

RNNs derive their strength and effectiveness from their unique mechanism of storing and flowing data across varying time intervals. These networks are adept at identifying patterns in data sequences, making them valuable tools for prediction and forecasting. Their versatility spans multiple domains, encompassing text, images, speech, and time series data. Positioned among the potent members of the ANN family, RNNs embody the intricacies of the biological brain, encompassing both memory and computational capabilities.

Analyzing LSTM networks

A specific form of RNN is the LSTM network, originally conceptualized by Hochreiter and Schmidhuber in 1997. This architecture has garnered renewed attention within the realm of DL due to its immunity to the vanishing gradient problem and its proven excellence in practical results and performance.

The vanishing gradient problem poses challenges in training ANNs through gradient-based learning techniques. Methods such as backpropagation adjust weights based on error gradients. However, these gradients can diminish exponentially as they propagate deeper into the network, sometimes rendering them exceedingly small and preventing weight adjustments. In extreme cases, this can halt network training altogether.

LSTM networks offer an ideal solution for predicting and classifying sequential data, outperforming numerous traditional ML methodologies. A significant example is Google's transition in 2012 from **hidden Markov models** (**HMMs**) to DNNs for voice recognition. By 2015, Google had adopted LSTM-based RNNs combined with **Connectionist Temporal Classification** (**CTC**) for improved voice recognition.

CTC serves as an output and scoring function for training RNNs. An LSTM network's strength lies in its ability to capture long-term dependencies among data, enhancing its utility in contexts such as speech recognition where understanding sentence context is crucial.

An LSTM network comprises interconnected LSTM cells, each featuring input, output, and **forget gates**. These gates facilitate writing, reading, and resetting functions on the cell memory. Rather than binary, the gates are analogical, typically managed by sigmoid activation functions that map values to a range (0, 1). This multiplicative nature empowers the cells to retain information over extended periods. The input gate controls whether the current state is combined with incoming input, while the forget gate resets the cell's state when its value drops to zero. The output gate determines whether the cell's content is retrieved or not, influencing the network's overall output.

The following diagram explains an LSTM unit:

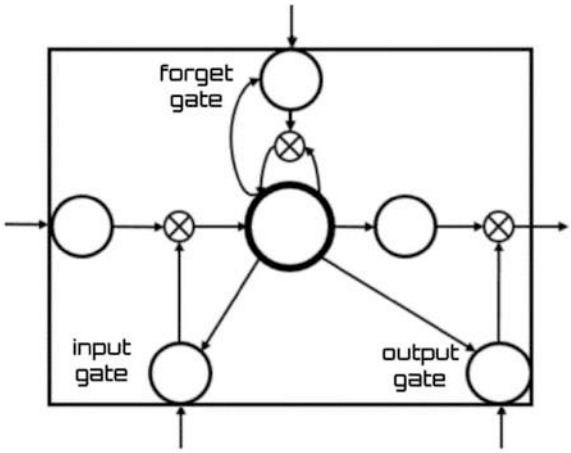


Figure 6.8 – LSTM architecture

NN-based approaches possess significant potency, enabling the extraction of inherent data characteristics and relationships. Notably, LSTM networks have demonstrated impressive real-world efficacy, boasting remarkable recognition rates. However, one drawback is that NNs function as **black-box models**. This means their behavior lacks predictability, and it's impossible to decipher the underlying logic by which they process data.

Introducing transformer models

Transformer models are a revolutionary architecture in the field of NLP and beyond. Introduced in 2017, transformers have since become a foundational framework for a wide range of tasks, from language translation to image generation. The key innovation in transformer models is the self-attention mechanism. Traditional models process sequences in a linear manner, which limits their ability to capture long-range dependencies. In contrast, transformers employ self-attention to weigh the importance of different words in a sequence with respect to each other, enabling them to consider global context and dependencies.

The transformer architecture consists of an encoder and a decoder, each comprising multiple layers. The encoder processes the input data, while the decoder generates the output. **Attention mechanisms** within these layers allow the model to focus on relevant parts of the input or output sequence.

Transformers can be applied with success in various applications:

- **Machine translation**: Models such as *transformers* and *BERT* have set new benchmarks in machine translation, allowing for more accurate and contextually meaningful translations
- **Language understanding**: BERT captures bidirectional context, significantly improving language understanding tasks
- **Text generation**: **Generative Pre-trained Transformer** (**GPT**) models generate coherent and contextually relevant text, finding applications in chatbots, content generation, and more
- **Image generation**: **Vision Transformer** (**ViT**) extends transformers to images, achieving competitive performance in image classification and generation tasks

One of the reasons behind transformers' success is their parallelizable structure, which speeds up training. Additionally, the attention mechanism enables capturing relationships without explicitly defining sequence lengths.

Despite their immense capabilities, transformers have their own challenges, such as a high computational cost for large models and the potential for generating biased or inappropriate content. Researchers continue to refine and adapt the transformer architecture to address these issues and push the boundaries of AI capabilities.

Summary

In this chapter, we learned the basic concepts of DL and discovered how to implement a CNN algorithm in the MATLAB environment. First, we looked at how DL enables automated feature extraction, then we looked at how to train a deep network, and then we got a taste of the most popular DL architectures.

We then focused on CNN to analyze it in detail. We learned about the different layers that make up this network and what functions these layers perform. We then saw in practice how to implement a CNN in the MATLAB environment for image classification of pistachio nuts. We learned how to correctly import the image database, how to draw the architecture of the network with the different layers one after the other, and how to set the network parameters. Finally, we saw how to use evaluation metrics for the correct interpretation of the results.

In the last section, we introduced some of the most used networks, which will be the subject of more detailed study and application in the following chapters.

In the next chapter, we will understand the basic concepts of NLP, how to use corpora and word and sentence tokenization, and how to build a model to label sentences, and finally, we will implement gradient boosting techniques using MATLAB.

Part 3: Machine Learning in Practice

In this part, we will take a journey into the specialized domains of MATLAB application, where we will reveal the immense potential of the software in various fields.

Explore the enchanting world of natural language processing using MATLAB, where language intricacies are deciphered, enabling machines to comprehend and respond to human communication. Transitioning into the visual realm, *Chapter 8* opens doors to the manipulation and understanding of visual data, bringing pixels to life through sophisticated algorithms. Delve into the intricate patterns of *Chapter 9*, where historical data paves the way for informed predictions and strategic decision-making. Then, witness the power of *Chapter 10*, where personalized suggestions and recommendations emerge from intricate algorithms, enhancing user experiences. Concluding our exploration, immerse yourself in the realm of *Chapter 11*, where subtle deviations in data patterns are unveiled, providing a critical edge in identifying outliers and potential threats. Join us as we navigate through these diverse applications, unlocking the capabilities of MATLAB in shaping the future of data analysis and decision support.

This part has the following chapters:

- *Chapter 7*, *Natural Language Processing Using MATLAB*
- *Chapter 8*, *MATLAB for Image Processing and Computer Vision*
- *Chapter 9*, *Time Series Analysis and Forecasting with MATLAB*
- *Chapter 10*, *MATLAB Tools for Recommender Systems*
- *Chapter 11*, *Anomaly Detection in MATLAB*

7

Natural Language Processing Using MATLAB

Natural language processing (**NLP**) automatically processes information conveyed through spoken or written language. This task is fraught with difficulty and complexity, largely due to the innate ambiguity of human language. To enable **machine learning** (**ML**) and interaction with the world in ways typical of humans, it is essential not only to store data but also to teach machines how to translate this data simultaneously into meaningful concepts. As natural language interacts with the environment, it generates predictive knowledge. In this chapter, we will learn the basic concepts of NLP and how to build a model to label sentences.

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

- Explaining NLP
- Exploring corpora and word and sentence tokenize
- Implementing a MATLAB model to label sentences
- Understanding gradient boosting techniques

Technical requirements

In this chapter, we will introduce basic ML concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also require working knowledge of the MATLAB environment.

To work with the MATLAB code in this chapter, you'll need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `IMDBSentimentClassification.m`
- `ImdbDataset.xlsx`

Explaining NLP

NLP is a field that's dedicated to the development of technology that enables computers to interact with, understand, and generate human language in a way that mimics natural human communication. This involves various techniques and approaches aimed at processing and analyzing the complexities of natural languages, such as English, Chinese, Arabic, and more. The goal is to bridge the gap between human language and computer language, allowing computers to comprehend and generate text as if they were engaging in a conversation with a human interlocutor (*Figure 7.1*):

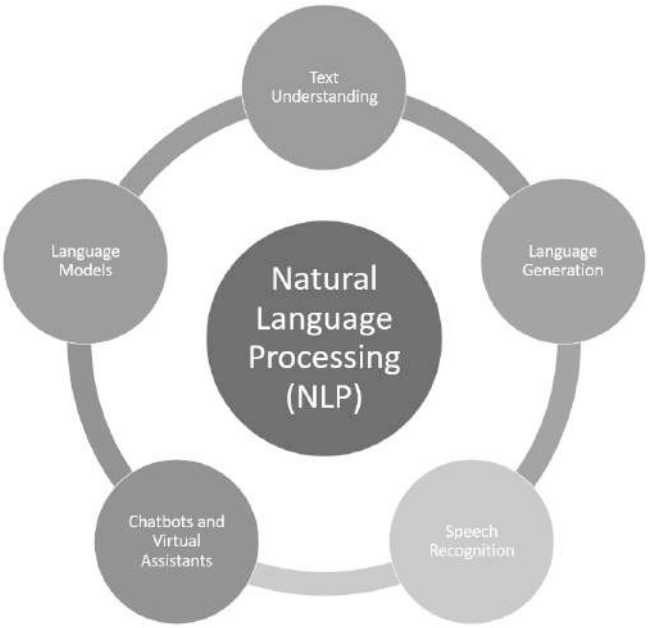


Figure 7.1 – NLP tasks

NLP strives to develop information technology tools for analyzing, comprehending, and creating texts in a manner that resonates with human understanding, mimicking interactions with another human rather than a machine. Natural language, both spoken and written, represents the most instinctive and widespread mode of communication. In contrast to formal languages, it holds a greater level of intricacy, often carrying connotations and uncertainties, which renders its processing quite challenging.

NLP encompasses a range of tasks, including but not limited to the following:

- **Text understanding**: This involves extracting meaning and information from text. It includes tasks such as sentiment analysis (determining the emotional tone of a text), named entity recognition (identifying names of people, places, organizations, and so on), and text classification (categorizing text into predefined classes).
- **Language generation**: This aspect focuses on generating coherent and contextually appropriate human-like text. It includes tasks such as machine translation (translating text from one language to another), text summarization (creating concise summaries of longer texts), and dialogue generation (constructing natural-sounding conversational responses).
- **Speech recognition**: NLP also extends to the realm of spoken language. Speech recognition technology converts spoken language into text, enabling applications such as voice assistants and transcription services.
- **Language models**: Recent advances in NLP have led to the development of large language models such as GPT-3, which can generate remarkably human-like text based on the input it receives. These models are trained on massive amounts of text data and can be fine-tuned for various NLP tasks.
- **Chatbots and virtual assistants**: NLP is the foundation of chatbots and virtual assistants, which can engage in text-based or voice-based conversations with users, providing information, assistance, and responses.

The field of NLP is interdisciplinary, drawing from computer science, linguistics, cognitive psychology, and more. It involves working with linguistic structures, statistical models, ML algorithms, and **deep learning** (**DL**) techniques to process, understand, and generate human language. While significant progress has been made, NLP still faces challenges, such as handling context, understanding nuances, and truly comprehending the intricacies of human communication:

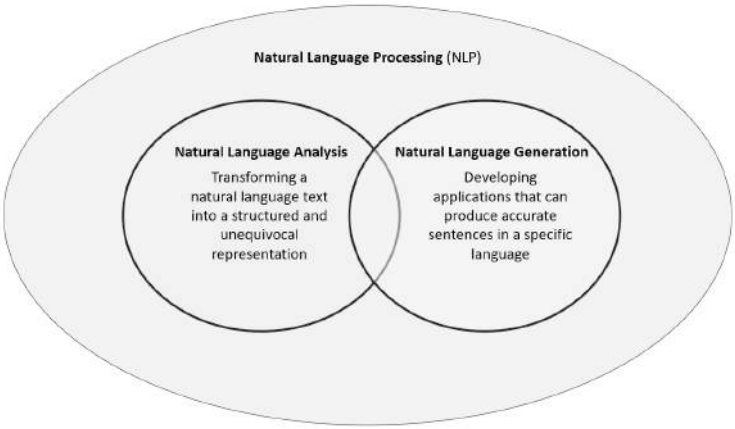


Figure 7.2 – Text analysis and text generation in NLP

The pursuits in this domain encompass two main objectives: text analysis and text generation. These principles give rise to the subsequent disciplines, known as **natural language analysis** (**NLA**) and **natural language generation** (**NLG**). We'll delve deep into both.

NLA

This field centers on enhancing machines' grasp of natural language. It involves transforming a natural language text into a structured and unequivocal representation. NLA involves the process of enabling machines to comprehend and interpret human language. Its primary goal is to bridge the gap between the unstructured nature of natural language and the structured representation that computers can work with. NLA encompasses several tasks:

- **Syntax analysis**: This involves parsing the grammatical structure of sentences to understand the relationships between words and their roles (subject, object, verb, and so on). It helps in identifying how words are organized to convey meaning.
- **Semantic understanding**: NLA seeks to comprehend the meaning of words and phrases in context. It involves extracting the underlying concepts and intentions behind the words used in a text.
- **Named entity recognition** (**NER**): NER is the process of identifying and classifying named entities, such as the names of people, organizations, locations, dates, and more, within a text.
- **Sentiment analysis**: Sentiment analysis aims to determine the emotional tone expressed in a piece of text, classifying it as positive, negative, neutral, or even more nuanced emotions.
- **Coreference resolution**: Coreference refers to cases where different words or phrases in a text refer to the same entity. Coreference resolution identifies and connects these references to provide a coherent understanding of the text.
- **Text classification**: Text classification involves categorizing text into predefined classes or categories based on its content. This is used for tasks such as spam detection, topic classification, and content tagging.
- **Information extraction**: This involves extracting specific information or structured data from unstructured text. An example of this is extracting dates, events, relationships, or numerical data from news articles.
- **Dependency parsing**: Dependency parsing analyzes the grammatical relationships between words in a sentence, often represented as a tree structure that shows how words depend on each other.

- **Language modeling**: Language models, often trained using ML techniques, are used to predict the likelihood of words or phrases occurring based on the context of the surrounding text. These models play a crucial role in understanding the probabilities and patterns of language.
- **Parsing ambiguity**: Languages often contain ambiguous structures that can be interpreted in multiple ways. NLA aims to resolve such ambiguities to arrive at the intended meaning of a sentence.

NLA draws from linguistic theories, computational linguistics, and ML. It's a complex field that continues to evolve, driven by advances in technology and the increasing demand for machines to understand and process human language in diverse applications, including search engines, chatbots, sentiment analysis tools, and more.

NLG

Here, the emphasis is on enabling machines to construct sentences in natural language. This involves developing applications that can produce accurate sentences in a specific language. NLG is a pivotal aspect of NLP that revolves around the creation of human-like text by computers. The main objective of NLG is to enable machines to generate coherent, contextually appropriate, and linguistically accurate textual content, mimicking the way humans communicate. NLG encompasses various techniques and processes to transform structured data or information into readable and understandable language. Here are some key points regarding NLG:

- **Machine translation**: One of the earliest forms of NLG, machine translation involves translating text from one language into another while retaining its meaning and context. Modern machine translation systems often utilize neural network-based approaches for improved accuracy.
- **Text summarization**: NLG is employed in summarizing longer texts, such as articles or documents, by condensing the main ideas and relevant information into a concise and coherent summary.
- **Dialogue generation**: NLG powers chatbots, virtual assistants, and other conversational agents. It involves constructing natural-sounding responses in conversations, considering the ongoing context and user inputs.
- **Data-to-text generation**: NLG is used to convert structured data, such as statistical figures or database entries, into human-readable narratives. An example of this is generating weather reports from weather data.
- **Content creation**: NLG systems can create articles, reports, and other content based on predefined templates or prompts. This can be particularly useful for generating routine reports or news updates.

- **Personalized messaging**: NLG can be employed to generate personalized messages or recommendations tailored to individual users – for instance, crafting product recommendations based on a user's browsing history.
- **Storytelling and creative writing**: Some NLG systems can generate fictional stories, poetry, or creative pieces of writing. While these might not be as advanced as human-generated content, they showcase the potential of NLG in creative domains.
- **Automatic code generation**: In programming, NLG can be used to generate code snippets or documentation based on high-level descriptions or specifications.
- **Medical reports**: NLG can assist in generating medical reports or patient summaries based on electronic health records and medical data.

NLG techniques can range from rule-based approaches to more advanced ML methods, such as **recurrent neural networks** (**RNNs**) and transformer models such as **generative pre-trained transformer** (**GPT**). These models learn patterns and structures from vast amounts of text data to produce coherent and contextually relevant language.

NLG is integral in applications where conveying information in a human-readable format is essential, making technology more accessible and user-friendly. While NLG systems have made significant progress, challenges still exist, including maintaining coherence over longer texts, ensuring accuracy, and avoiding biases in generated content.

Analyzing NLP tasks

NLP encompasses a series of tasks within text analysis, resulting in a layered structure. The foundational strata upon which sentence analysis relies are outlined as follows:

- **Morphology analysis**: The objective of this stage is to break down input language strings into sets of tokens, which correspond to discrete words, sub-words, and punctuation elements. Through the process of tokenization, the text is fragmented, resulting in a sequence of tokens, with each token representing a word from the text. Within this stage, two concepts hold significance: the stem and the lemma. The stem is the foundational form of a word and is obtained by removing inflections (such as verb conjugations or noun plurals) from its altered version. Conversely, the lemma represents the standardized form of the word, chosen conventionally to encompass all its inflected variations. This phase involves identifying both the stem and lemma for each word, which can be accomplished through two distinct operations: stemming and lemmatization, respectively. This information becomes instrumental in subsequent analysis stages. The rationale behind this approach lies in the efficient utilization of memory – maintaining rules based on word components and their combinations that form specific inflected forms is much more resource-effective than managing each word as an individual element within an extensive inventory.

- **Syntax analysis**: To grasp the meaning of a sentence, it's insufficient to merely comprehend the definitions of its constituent words; it's equally crucial to discern the relationships between these words. This stage addresses the syntactic examination of the provided text. All components of speech are recognized and encompass verbs, nouns, adjectives, adverbs, prepositions, and pronouns. The process that annotates each word with its respective part of speech is termed **part-of-speech tagging**. This process is divided into two sub-processes: firstly, shallow parsing generates a binary tree in which elementary segments – namely, the **nominal part** (**NP**) and the **verbal part** (**VP**) – are identified. Secondly, full parsing produces a syntactic tree that designates the syntactic role of each word within the sentence.
- **Semantic analysis**: The semantic analysis phase leverages insights garnered from preceding analytical stages, encompassing the meanings of individual words and their interconnected relationships, to interpret the overall significance of the sentence. The process of named entity recognition is employed to detect and classify groups of words that collectively form an entity, which might include personal names, countries, events, and the like. Semantic processing endeavors to deduce the potential interpretation of a sentence, with a particular focus on the interplay between the meanings of the words within the sentence being examined. This level of processing may encompass semantic disambiguation for words that possess multiple meanings, akin to how disambiguation is performed for words that can adopt multiple syntactic roles. Diverse methodologies can be employed for disambiguation, some of which involve assessing the frequency of each meaning within a specific corpus, while others consider contextual cues. Additionally, certain methods tap into pragmatic knowledge within the document's domain to aid in disambiguation.
- **Pragmatic analysis**: This phase revolves around identifying the contextual environment in which the text is situated and the subsequent utilization of that context. Specifically, pragmatics delves into how the context influences the interpretation of meanings. In this context, *context* refers to the surrounding circumstances, including a range of non-linguistic factors (such as social dynamics, environmental cues, and psychological factors) that impact linguistic expressions. Indeed, human language is not solely rooted in its morphological, syntactic, and semantic attributes; it is also deeply intertwined with external knowledge linked to the situation in which a sentence is embedded. Within pragmatic analysis, a distinction is often drawn between the literal meaning of an utterance and the intended meaning of the speaker. The literal meaning pertains to the direct interpretation of the expression, while the speaker's intention refers to the underlying concept the speaker aims to convey. For accurate interpretation of communication, several factors may be essential. These include the following:
    - Understanding the roles and statuses of the conversational participants
    - Recognizing the spatiotemporal context of the situation
    - Possessing knowledge about the subject matter being discussed

The ability to comprehend implied meanings from another speaker is termed **pragmatic competence**. Despite its significance, this type of analysis is still not extensively detailed in the literature, primarily due to the considerable challenges it poses.

Introducing automatic processing

From an automated processing perspective, the syntactic level presents fewer challenges. Complex sentences can be analyzed, their grammatical accuracy identified, and their syntactic structures reconstructed seamlessly. The semantic level, on the other hand, proves significantly more intricate. In simpler scenarios, the following approach is feasible:

- Sentence meaning is deduced from the meanings of individual words once the sentence's syntactic structure has been discerned
- The meanings of individual words are derived from a readily accessible dictionary through automated means

However, numerous issues arise, as follows:

1. Firstly, the same word can bear distinct meanings in diverse contexts.
2. Secondly, the syntactic structure of a sentence can be ambiguous, occasionally attributing different structures to the same sequence of words.
3. Lastly, certain idiomatic phrases possess meanings separate from the literal interpretations derived from composing the meanings of their constituent words.

Artificial intelligence endeavors to tackle these challenges by leveraging a suitable knowledge base that the language processing program can access.

The pragmatic level introduces an even more intricate terrain. The primary challenge stems from our capacity, during a conversation, to anticipate the mental states of our conversational partners. In essence, effective communication necessitates the representation of interlocutors' intentions, which are only partially evident in their spoken words.

Now that we've introduced the basic concepts of NLP, let's analyze the specific contents that allow us to approach the problem from a practical point of view.

Exploring corpora and word and sentence tokenizers

The analysis of corpora, words, and sentence tokenization forms the basis for comprehensive language understanding. Corpora provides real-world language data for analysis, words constitute the elements of expression, and sentence tokenization structures the text into meaningful units for further investigation. This trio of concepts plays a central role in advancing linguistic research and enhancing NLP capabilities.

Corpora

In linguistics and NLP, corpora refer to extensive collections of written or spoken texts that serve as valuable sources of data for linguistic analysis and language-related studies. Corpora provides a diverse range of language samples, enabling researchers to examine patterns, trends, and variations in language usage, syntax, and semantics across different contexts and genres.

Linguistic corpora represent sizable collections of spoken or written texts, often originating from authentic communication contexts (including speeches or newspaper articles). These collections are digitized and frequently accompanied by computerized tools for convenient access:

Figure 7.3 – Word cloud of corpora basic key concepts. The words are displayed in different sizes, with the most frequent words being the largest. Word clouds are often used to identify the key concepts in a corpus

Corpora serve the purpose of examining how language is practically utilized and validating overarching trends through statistical analysis. They hold a pivotal role in contemporary lexicography, aiding in tasks such as selecting lemmas based on their usage frequency, identifying typical linguistic structures involving specific words, and grasping subtle nuances of meaning within various contexts.

Furthermore, corpora wield significance in the advancement of language technologies such as automatic translation and speech recognition. In these applications, corpora are employed to construct statistical language models. They also find utility in language education, supporting the creation of instructional resources. Particularly for advanced learners, corpora enable the deduction of word properties and linguistic structures by observing their contextual applications.

There are various examples of corpora; the most used are as follows:

- **The Brown Corpus**: One of the first and most well-known corpora in English, it contains over a million words of text from various sources
- **The New York Times Corpus**: A large corpus of news articles from the New York Times, spanning from 1987 to 2007

- **The Project Gutenberg Corpus**: A collection of over 60,000 free electronic books in the public domain
- **The Linguistic Data Consortium** (**LDC**) **Corpora**: A repository of a wide variety of corpora, including those for different languages and domains

MATLAB provides a variety of tools for accessing and processing corpora. Some of the most commonly used tools are as follows:

- **The Text Analysis Toolbox**: A comprehensive toolbox for text processing, including functions for tokenization, stemming, lemmatization, and sentiment analysis
- **The Natural Language Processing Toolbox**: A toolbox for NLP tasks, such as named entity recognition, part-of-speech tagging, and dependency parsing
- **The Web Services Toolbox**: A toolbox for accessing and interacting with web services, including those that provide access to corpora

Words

Words are the foundational units of language that carry meaning and convey ideas. The analysis of words involves investigating their forms, meanings, relationships, and usage patterns. This examination can encompass aspects such as morphological structure, part-of-speech categorization, frequency distribution, and semantic associations. Studying words allows linguists to understand how language is structured and how meanings are conveyed through vocabulary.

In ML and DL tasks, words are represented as vectors of numbers. This is done because ML algorithms can only work with numerical data. The process of converting words into vectors is called word embedding.

There are two main types of word embedding:

- **Bag of words** (**BoW**): In **BoW**, each unique word in a corpus is assigned a unique index. A document is then represented as a vector of the counts of each word in the document. Let's assume vocab size = 4; here, a sample document that states "I love cats" is represented as [1 1 1 0]. In BoW, we simply represent the document by the frequency of each word. For example, if we have a vocabulary of 1,000 words, then the whole document is represented by a 1,000-dimensional vector, where the *ith* entry of the vector represents the frequency of the *ith* vocabulary word in the document.
- **Word2vec**: Word2vec is a more sophisticated approach to word embedding that takes into account the context in which words appear. Word2vec learns two types of embeddings: word embeddings and context embeddings. Word embeddings are vectors that represent individual words, while context embeddings are vectors that represent the words that appear around a given word. For example, the word embedding for "cat" might be [0.1, 0.2, 0.3], while the context embedding for "cat" might be [0.4, 0.5, 0.6].

Sentence tokenize

During the **sentence tokenization** process, we identify atomic elements known as tokens within each sentence. These tokens serve as the basis for analyzing and evaluating the sentence itself. Consequently, during the tokenization phase, we not only recognize and assess these elements but also occasionally convert negative constructs. This results in the text being divided into tokens, which can be thought of as *indivisible* units. This task is relatively straightforward for languages that use spaces to separate words but becomes considerably more intricate for languages with continuous spelling systems. Focusing on languages such as English, which fall into the former category, a token can be defined as any sequence of characters enclosed by spaces. However, it's worth noting that this definition allows for numerous exceptions. In languages where word boundaries aren't explicitly indicated in written text, this process is referred to as word segmentation:

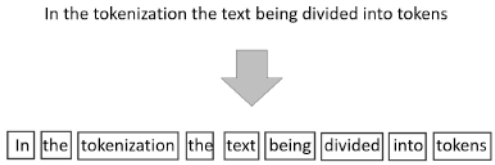


Figure 7.4 – Word tokenization process

In the broader context of tokenization, several challenges need to be addressed:

- Lack of spaces separating words from preceding or succeeding punctuation
- Instances where sequences of characters are not separated by spaces should be treated as two separate tokens
- Occasions when sequences of characters are separated by spaces should be considered a single token
- Managing uppercase and lowercase variations
- Navigating variations in spelling conventions

Ultimately, at the end of this procedure, we identify a string with an assigned meaning, which we refer to as a **token**. Each token is structured as a pair comprising a token name and an optional token value.

Now that we have examined the basics of NLP in detail, we can proceed to analyze the implementation of a sentence classifier in MATLAB.

Implementing a MATLAB model to label sentences

In this section, we will discuss a very interesting topic that is very popular in today's society. I am referring to the importance of reviews in influencing a customer's interest in making the right decision.

Introducing sentiment analysis

Sentiment analysis, a technique that utilizes NLP, extracts and analyzes subjective information from text. Analyzing vast datasets reveals collective opinions that impact various domains. While manual sentiment analysis is challenging, automated methods have emerged. However, automating language modeling is complex and costly due to the nuances of human language. Additionally, the methodology varies across languages, increasing complexity.

A major challenge lies in determining the polarity of opinions. Polarity classification is subjective, with one sentence perceived differently by individuals based on their value systems. The rise of social media has heightened interest in sentiment analysis. As online expressions proliferate, this field has become valuable for businesses to promote products, identify opportunities, and protect their reputation. Challenges arise from the use of simplistic terms in sentiment analysis algorithms to convey opinions. Cultural influences, linguistic disparities, and contextual factors make converting written text into a binary positive or negative sentiment highly intricate:

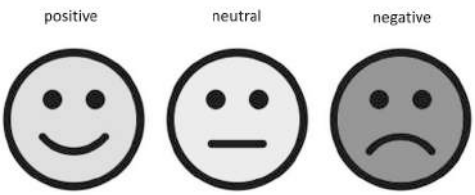


Figure 7.5 – Sentiment analysis polarity

Advanced sentiment analysis endeavors to pinpoint specific moods, such as happiness, sadness, or anger. For instance, it can involve tasks such as categorizing a song review not just as positive or negative, but also predicting a numerical score, conducting a comprehensive analysis of hotel reviews, and providing ratings for various aspects such as comfort, noise levels, and design.

Furthermore, it's feasible to categorize a given text into one of two classes: objective or subjective. A text may contain objective information, as seen in a news article, or it can be subjective, such as the expression of political opinions in an interview. The subjectivity of sentences often hinges on their context, and even an objective document may feature subjective phrases, such as quotations. Discerning subjectivity versus objectivity can be more challenging than classifying polarity, as it relies heavily on the surrounding textual context.

Sentiment analysis offers several approaches, with the most commonly employed methods falling into four broad categories:

- **Lexicon-based methods**: These techniques identify emotional keywords within the text and assign them a predetermined affinity to represent specific emotions.
- **Rule-based methods**: This approach classifies text based on explicit emotional categories, relying on the presence of unambiguous emotional words such as *happy*, *sad*, or *bored*.

- **Statistical methods**: In this category, the aim is to ascertain the source of the sentiment (the subject) and the target (the object being evaluated). To gauge opinions in context and identify the assessed attribute, grammatical relations among the words are examined through an extensive analysis of the text.
- **ML methods**: These methods employ various learning algorithms to deduce sentiment, often utilizing a dataset with predefined classifications (supervised methods). The learning process is iterative, requiring the construction of models that associate polarity with different types of comments and, when necessary, topic analysis for a more comprehensive understanding.

The utilization of emojis and emoticons is on the rise. Instead of relying solely on words, people often express their emotions using tiny round faces and other symbols. These visual elements prove valuable for swiftly and pleasantly conveying one's message. Using DL architectures, it is possible to learn the relationship between emoticons and sentiments. For example, a DL method might be trained on a large corpus of text and emoticons to learn the features of emoticons that are most predictive of sentiment.

Let's learn how to apply the tools made available by DL to tackle a practical case of sentiment analysis.

Movie review sentiment analysis

Reviews have become an essential element for businesses to thrive, providing valuable feedback from customers and driving purchasing decisions. They serve as a reliable source of information for prospective buyers and offer sellers insights to enhance their products and services. Moreover, they foster user engagement and amplify the power of word-of-mouth marketing, significantly impacting a company's online reputation. Securing positive reviews is crucial for businesses of all sizes to maintain profitability.

The rise of social media and the internet has dramatically transformed consumer buying behaviors. Potential buyers now actively seek online reviews and information before making purchasing decisions, relying less on traditional word-of-mouth recommendations. They have become more independent and discerning, carefully considering their choices.

In the past, film critics played a dominant role in shaping consumer preferences, influencing movie choices and fostering critical thinking. However, the advent of consumer-generated reviews has shifted the paradigm, with consumer opinions now holding greater weight in influencing purchasing decisions. This democratization of opinion has empowered consumers to share their experiences and shape the cultural landscape.

For the task of classifying movie reviews as either positive or negative sentiment, we will employ a short version of the IMDb Movie Reviews dataset. This dataset is specifically designed for binary sentiment classification and contains 748 reviews opportunely labeled. The dataset was adequately revisited in this example using only two features: review and class.

Let's learn how to build a model for classifying sentences using an RNN.

Using an LSTM model for label sentences

An LSTM model is an advanced form of **recurrent neural network** (**RNN**) that can learn long-term dependencies within sequences, making it a powerful tool for processing sequential data, such as text, speech, and time series data. This makes it ideal for tasks such as label sentence classification, where the goal is to assign a label to a sentence based on its overall sentiment. LSTM has been successfully applied to a variety of tasks, including sentiment analysis, machine translation, speech recognition, and natural language generation.

To learn how to implement a model to label sentences in MATLAB, we will use movie reviews that have been made by a lot of online user:.

1. To begin, we need to import the dataset into the MATLAB workspace. You can find the dataset in `.xlsx` format on the project's GitHub page (`ImdbDataset.xlsx`). To simplify this task, the dataset has been appropriately trimmed. Once you've downloaded the file, all you need to do is specify the root folder's path. Then, you can load the data into the MATLAB workspace like so:

```
ImdbData = readtable("ImdbDataset.xlsx",'TextType','string');
```

   To load the data, we used the `readtable()` function. When dealing with spreadsheet files, this function generates a distinct variable for each column found in the file and extracts variable names from the initial row of the file.

   The objective of this example is to categorize reviews based on the labels in the `Class` column. To categorize the data into classes, you should convert these labels into categorical variables:

```
ImdbData.Class = categorical(ImdbData.Class);
```

   After loading the data, we need to visualize the distribution of the classes in the dataset by creating a histogram. This will help us assess whether the data is evenly distributed:

```
figure
histogram(ImdbData.Class);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```

   The diagram shown in *Figure 7.6* will be drawn:

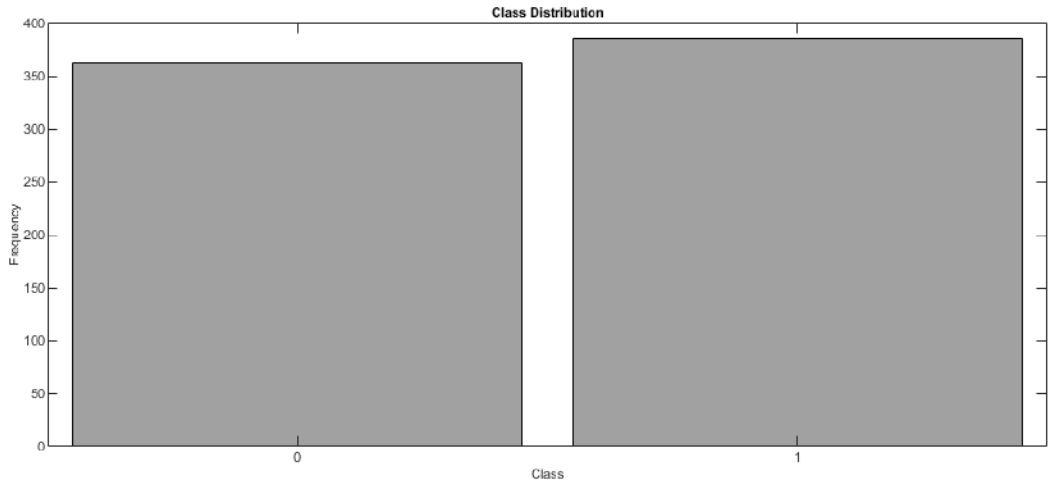


Figure 7.6 – Class distribution of the movie review: 0 negative, 1 positive

2. The subsequent step involves dividing the data into training and validation sets. Partition the data into a training set and a separate held-out set for validation and testing purposes:

```
DataSplitting = cvpartition(ImdbData.Class,'Holdout',0.3);
```

3. The `cvpartition()` function establishes a random dataset partition. You can employ this partition to create training and testing sets for assessing a statistical model through cross-validation. Set the holdout percentage to 30%; this means that 70% of the data will be allocated for training, while the remaining 30% will be reserved for validation.

   Now, we will utilize the training option to obtain the training indices and the test option to obtain the testing indices for cross-validation:

```
TrainDataset = ImdbData (training(DataSplitting),:);
TestDataset = ImdbData (test(DataSplitting),:);
```

   Following this, we will retrieve the review data and corresponding classes from the partitioned tables:

```
TrainReview = TrainDataset.Review;
TestReview = TestDataset.Review;
TrainClass = TrainDataset.Class;
TestClass = TestDataset.Class;
```

4. At this point, our data has been prepared for use, and we can start preprocessing the review data. We will employ a function designed for tokenization and text preprocessing called `preprocessText`. This function carries out the following tasks:

   I. Tokenizes the text using `tokenizedDocument`.

   II. Converts the text into lowercase using `lower`.

   III. Removes punctuation using `erasePunctuation`.

   We will apply the `preprocessText` function to both the training and validation data for preprocessing:

   ```
   TrainDoc = preprocessText(TrainReview);
   TestDoc = preprocessText(TestReview);
   ```

   After that, we have to apply the word encoding technique. This method involves transforming words from a human language, such as English, into a numerical or symbolic format that's suitable for a wide range of computational tasks, with a particular focus on applications in NLP and ML.

5. To transform the documents into sequences of numeric indices, we will employ the `wordEncoding()` function, as follows:

   ```
   EncText = wordEncoding(TrainDoc);
   ```

   This function associates words within a vocabulary with numeric indices.

   To use the reviews as input for an LSTM, all inputs must have the same length. However, the reviews in the dataset vary in length. One solution is to truncate longer reviews and pad shorter ones to ensure they are all the same length. To truncate and pad the reviews, start by choosing a target length. Then, truncate longer documents and pad shorter ones. To achieve optimal results, the target length should be short enough that you can avoid discarding a significant amount of data.

6. To find an appropriate target length, we can visualize a histogram of the lengths of the training documents:

   ```
   NumDoc = doclength(TrainDoc);
   figure
   histogram(NumDoc)
   xlabel("Number of tokens")
   ylabel("Number of Reviews")
   ```

   To start, we counted the length of each review. Then, we drew a histogram of the distribution. The following chart was printed:

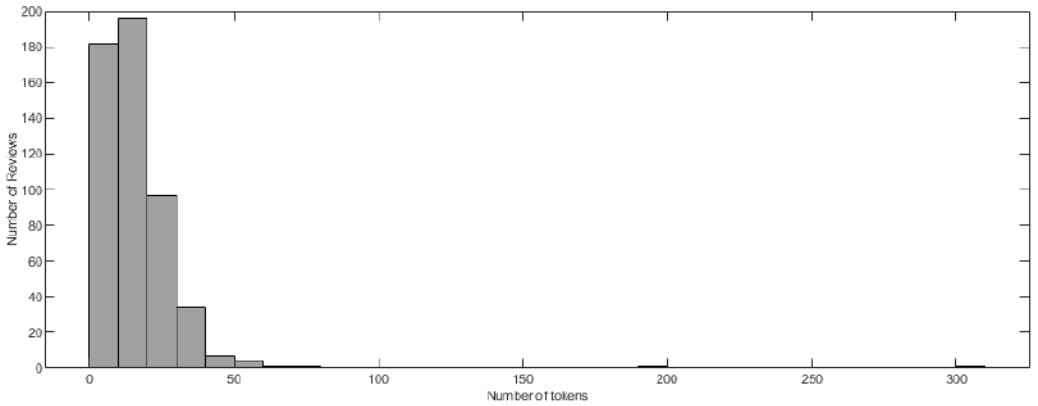


Figure 7.7 – Tokens distribution of the movie reviews

By analyzing the tokens distribution of the movie reviews, we'll notice that most reviews have several tokens between 0 and 30. So, we can choose this number as the target length:

```
TokNum = 30;
```

7. Now, we need to convert the reviews into numeric indices sequences:

```
RewTrain = doc2sequence(EncText,TrainDoc,'Length',TokNum);
RewTest = doc2sequence(EncText,TestDoc,'Length',TokNum);
```

To do this, we used the `doc2sequence()` function, which transforms documents into sequences that are suitable for DL. This function provides a cell array containing the numeric indices of words in the documents, as determined by the word encoding provided. Each element within the sequences array is a vector consisting of indices representing the words within the respective document.

8. Now, we will proceed with defining the architecture of the LSTM network:

```
inputSize = 1;
embeddingDimension = 150;
numHiddenUnits = 50;
numWords = EncText.NumWords;
numClasses = numel(categories(TrainClass));
layers = [ ...
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension,numWords)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

To feed sequence data into the network, incorporate a sequence input layer with an input size of 1. A sequence input with 1 dimension means each numerically encoded token is represented by a 1x1 scalar. Following that, introduce a word embedding layer with a dimension of 150, which should match the number of words specified in the word encoding. Word embedding layers learn to map words to vectors in a way that captures the meaning of the words. Subsequently, include an LSTM layer with 50 hidden units. For a sequence-to-label classification task using the LSTM layer, configure `OutputMode` last. Finally, add a fully connected layer that's the same size as the number of classes, followed by a softmax layer and a classification layer.

The architecture of the LSTM that's been defined is shown here:

```
layers =
6×1 Layer array with layers:
1   ''   Sequence Input          Sequence input with 1
dimensions
2   ''   Word Embedding Layer    Word embedding layer with 150
dimensions and 2410 unique words
3   ''   LSTM                   LSTM with 50 hidden units
4   ''   Fully Connected         2 fully connected layer
5   ''   Softmax                softmax
6   ''   Classification Output   crossentropyex
```

9. Now, we have to specify the training options:

```
options = trainingOptions('adam', ...
    'MiniBatchSize',16, ...
    'GradientThreshold',2, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{RewTest,TestClass}, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

The following options were set:

- `adam solver`: Adam, derived from *Adaptive Moment Estimation*, is a widely embraced optimization algorithm in the realms of ML and DL. Its primary role is to adjust the parameters, while encompassing weights and biases, within a model as it undergoes training. Adam is especially renowned for its effectiveness in training neural networks.
- `MiniBatchSize`: This is a hyperparameter that dictates the number of examples, which are individual data points or samples, that are employed in each training iteration during the execution of Adam or a similar optimization algorithm.
- `GradientThreshold`: This specifies a threshold value for gradients during the training process. It is used to make training models more stable and efficient, particularly when dealing with RNNs or deep architectures.

- `Shuffle`: This option shuffles the data in each epoch to increase the degree of randomness in the algorithm.
- `ValidationData`: This option specifies the data used for the validation process.
- `Plots`: With this option, the training process will be checked through a plot that upgrades the evolution of the process.
- `Verbose`: The output of the training process will be hidden.

10. Finally, we can start the training process:

```
LSTMNet = trainNetwork(RewTrain,TrainClass,layers,options);
```

The following plot will be printed:

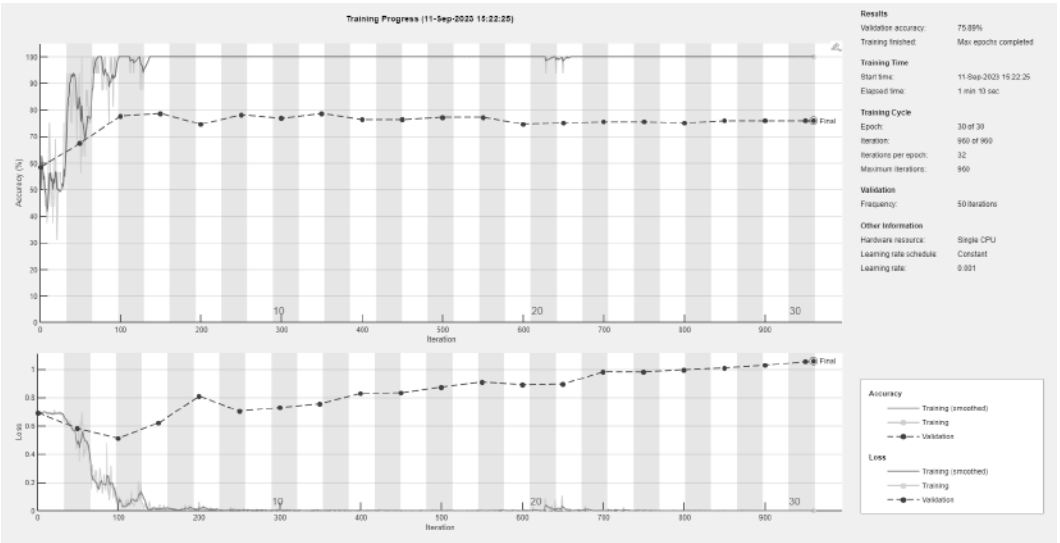


Figure 7.8 – Training process of the LSTM

The training process processed the data by building a network capable of returning a validation accuracy of 75.89%. Given the small amount of data available, this result can be considered satisfactory.

To improve the performance of the model, we can try the simulation with a slightly large number of LSTM units (100 instead of 50). However, changing the embedding dimension (from 150 to 100) could increase the validation accuracy to at least 85%.

Now that we've analyzed how to build an LSTM to classify movie reviews, let's learn how to improve the accuracy of predictions using ensemble methods.

Understanding gradient boosting techniques

To improve the performance of an algorithm, we can perform a series of steps and use different techniques, depending on the type of algorithm and the specific problems being addressed. The first approach involves a thorough analysis of the data to identify possible inaccuracies or shortcomings. In addition, many algorithms have parameters that can be adjusted to achieve better performance – not to mention techniques such as feature scaling or feature selection. A popular technique is to combine the capabilities offered by different algorithms to achieve better overall performance.

Approaching ensemble learning

The concept of **ensemble learning** involves the use of multiple models combined in a way that maximizes performance by exploiting their strengths and mitigating their relative weaknesses. These ensemble learning methods are based on weak learning models that do not achieve high levels of accuracy on their own, but when combined can produce robust predictions. In the context of ensemble learning, a **weak learner** is a model that can produce slightly better results than randomly, while a **strong learner** approaches an ideal model and can overcome problems typical of traditional models, such as overfitting.

Each model that's used returns predictions that may or may not be correct. The prediction error, in a supervised learning problem, is computed as follows:

$$error = bias^2 + variance$$

The two fundamental components of error are precision and variance. The goal of models that use ensemble approaches is to obtain reliable predictions by trying to reduce both **variance** and **bias**. Variance is the variability of a model's prediction for a given data point. A model with high variance is said to be overfitting, which means that it is learning the noise in the training data rather than the underlying patterns. This can also lead to poor generalization performance. Bias is the difference between the average prediction of a model and the correct value. A model with high bias is said to be underfitting, which means that it is not learning the underlying patterns in the data. This can lead to poor generalization performance, where the model does not perform well on unseen data. There is often a trade-off between these two components: classifiers with low accuracy tend to have high variance and vice versa. The goal of ML is to find a classifier that has a good balance of bias and variance. This is known as the bias-variance trade-off. In general, it is difficult to find a classifier that has both low bias and low variance. However, several techniques can be used to reduce bias and variance, such as feature selection, regularization, and cross-validation.

To improve the accuracy of the model, it is assumed that the different classifiers can make different errors on each sample, but agree on the correct classification. By averaging the outputs of the classifiers, the overall error is reduced by averaging the error components. Combining the outputs by the model does not necessarily guarantee significantly better classification performance than other models, but it does help to reduce the likelihood of selecting a classifier with disappointing performance.

There are several ensemble methods. They mainly differ in the way they process the data. The most common are as follows:

- **Bagging**
- **Random forest**
- **Boosting**

The first ensemble models were developed in the 1990s, but at the time of writing, there is no specific study that can help a user select the most appropriate meta-classifier. These methods are widely used to solve a variety of learning problems, including feature selection. In the subsequent sections, we will analyze the basics of each kind of ensemble technique.

Bagging definition and meaning

Bagging is an ensemble learning method that collects the predictions of weak learners (basic models), compares them, and combines them into a single prediction. The distinguishing feature of bagging is how the training sets for each decision tree are selected.

To create each training set, bagging uses sampling with replacement, known as bootstrapping, on the original dataset. **Bootstrapping** is a statistical resampling technique that's used to approximate the sample distribution of an estimator. This means that data is randomly selected from the original training set, with the possibility of some data being selected multiple times. This procedure increases the diversity of the decision trees that are produced.

A special feature of this method is that it works best when the basic learners (weak learners) are not very stable and therefore very sensitive to variations in the training data. In the presence of stable basic learners, the overall performance may even deteriorate.

The basic concept behind bagging is to combine multiple models that are unstable but have a low bias to produce a low variance overall model. The logic behind bagging is based on the idea that the expected value of the mean of identically distributed random variables is equal to the expected value of a single random variable.

The bagging equation is as follows:

$$y_predicted = 1/T\ \Sigma_t\ =\ 1\ \wedge\ T\ f_t(x)$$

Here, we have the following:

- T is the number of models in the ensemble
- $f_t(x)$ is the prediction of the t-th model

Therefore, the goal is to reduce variance by exploiting this averaging operation.

Typically, the decision tree is an appropriate choice for this approach because, despite its simplicity, it can effectively capture non-linear relationships and interactions between variables.

The key parameter for adjusting bagging is the number of trees. However, simulation studies and empirical evidence have shown that the performance of this model is not overly influenced by the choice of this parameter. Consequently, it is sufficient to choose a sufficiently large number of trees to obtain satisfactory results.

In summary, bagging is an ensemble learning approach that's based on repeatedly sampling training data and combining weak learner predictions and is particularly effective when working with basic models that are sensitive to variations in the data.

Discovering random forest

Random forest is a special case of bagging, with the basic aim of reducing variance by averaging over many models. Again, models based on decision trees are often the ideal choice as they can effectively capture the complex interaction structures in the data, which can be difficult to achieve with other approaches. By generating a large number of trees, bias tends to be low.

Although the conceptual basis of both techniques is similar, random forest can produce superior results with the correct adjustment of model parameters. This is due to its ensemble learning approach and the ability to tune its hyperparameters effectively. This is because, although the trees have errors, the use of bagging allows trees to be generated from an identical distribution so that the expected value of each tree is equal to the expected value of the mean of the *N* trees generated.

The bias of bagged models is similar to that of individual trees, so improvement focuses on reducing variance. This is the distinguishing feature of random forest models, which aim to improve the variance of bagging by reducing the correlation between trees.

For *N* identically distributed random variables, each with variance, *σ2*, and with variance of the mean over *N* variables equal to *(1/N)σ2*, if the variables are identically distributed but not necessarily independent, and a positive pairwise correlation, ρ, is assumed, the variance of the mean is expressed as follows:

$$\textit{Variance of the mean} = (1/N)\,\sigma^2 + [(N-1)/N]\rho\,\sigma^2$$

In this formula, the first term represents the variance of the individual random variables divided by *N*, which represents the variance of the mean when the variables are independent. The second term considers the pairwise correlation between the random variables, increasing the variance of the mean according to how correlated they are.

The technique that's used by random forest is to reduce the correlation between the trees, and hence the variance is to grow the trees by randomly selecting the input variables. The main difference with bagging is that random forest selects fewer candidate variables for partitioning than the total number of variables available.

Random forest is a method that combines a series of decision trees to produce more accurate results. These random decision trees are constructed while taking into account two key concepts:

- **Random sampling of the training data**: During the training phase, each tree in a random forest learns by focusing only on a random sample of the original data. These samples are extracted with replacement, a technique known as bootstrapping, which means that some data can be used multiple times within the same tree. While this theoretically produces a high variance in each tree compared to training on a specific dataset, the overall forest (the set of trees) tends to have a lower variance without a significant increase in bias.
- **Random feature sets for partition choice**: In random forest, only a subset of the available features is considered for partitioning the nodes in each decision tree. Typically, this subset is the square root of the total number of features. For example, if there are 16 features, only four random features are considered at each node in each decision tree to perform the partitioning. Of course, it is also possible to train trees by considering all the features at each node, as you would typically do in regression.

In summary, random forest combines hundreds or thousands of decision trees, each trained on a slightly different set of data and using only a limited number of features for subdivisions. The final predictions of random forest are obtained by averaging the predictions of each decision tree, which means that each tree contributes equally to the final decision.

Boosting algorithms explained

Boosting is another approach that can be applied to decision trees to improve their predictive ability. Like bagging, boosting can be applied to both regression and classification statistical models.

Both Bagging and boosting can be applied to decision trees to predict their abilities. However, there are some differences. The crucial aspect to focus on in bagging is the independence of each tree generated by a subset from the others.

Boosting, however, takes a different approach. Instead of generating completely independent trees as in bagging, boosting trains a sequence of trees iteratively. Each successive tree tries to correct the errors of the previous ones. In other words, the subsets of data used to train successive trees are weighted according to the erroneous predictions of the previous trees. This continuous process aims to progressively improve the performance of the overall model.

In summary, while bagging is based on creating independent trees from subsets of data, boosting focuses on the iterative training of trees that correct the errors of their predecessors. Both are used to improve the predictive capabilities of decision trees, but they take different approaches.

The special feature of boosting is the sequential construction of decision trees – that is, each tree is constructed based on the information processed and obtained from the previous tree.

Unlike other techniques that attempt to model a decision tree on the entire dataset, or at least a training set, the goal of boosting is to obtain a predictive model with incremental learning. We say stepwise learning because a first model is built on the data of the training set. A second model is then built that attempts to correct the errors of the first model. Rather than building a new model based on the previous *Y*-prediction, the approach is to build the next tree by trying to model the errors that were obtained as best as possible. The idea behind this is to create sequential weak learners (weak models), each of which tries to correct the errors of the previous model. In boosting, each subsequent tree in the ensemble is trained on the residuals of the previous tree, which are the differences between the actual and predicted values. This approach allows each tree to focus on correcting the errors of the previous trees, leading to improved overall accuracy.

Each of these trees may have as few final nodes as possible, even though the initial dataset is large. This may be due to the parameters of the algorithm or because smaller trees may perform better in correcting the errors of the previous tree.

The goal, tree by tree, is to improve the final model in the areas where it is not performing best.

Boosting offers three parameters that can be manipulated to optimize the modeling of decision trees:

- **Number of trees** (**T**): As with bagging techniques, increasing the number of trees (*T*) generally leads to greater prediction accuracy. However, it is important to note that too high a value of *T* can increase the risk of overfitting the model. For example, compared to random forest, boosting tends to be more sensitive to overfitting at high values of *T*.
- **Learning rate** (**λ**): This parameter indicates the learning rate of the tree and usually has a low value, typically between 0.01 and 0.001. The value of λ has a huge impact on the results of the model. A very low value of λ requires a larger number of trees (*T*) to obtain a well-performing model. The choice of the optimal value of λ can significantly influence the behavior of the model.
- **Number of splits** (**s**): *s* indicates the number of splits and has a value of one by default, which means that you get "stumps" or trees with only one parent node and two leaf nodes. Although a value of 1 for *s* is often used to speed up the calculation, changing it can significantly affect the model results. By changing the value of *s*, it is possible to obtain trees with different structures and complexities.

There are several boosting techniques, each with variations in how trees are constructed or weighted to improve overall predictive ability. Some of the main boosting techniques are as follows:

- **Adaptive Boosting** (**AdaBoost**): AdaBoost is one of the earliest boosting algorithms and one of the most popular. In this approach, each training sample is given an initial weight, and subsequent iterations focus on correcting errors made in previous iterations. Weak learning trees, often stumps, are built based on these weights. AdaBoost gives more weight to past errors, allowing subsequent trees to focus on improving predictions for previously misclassified examples.

- **Gradient boosting**: Gradient boosting is a generic boosting framework that can be implemented in different ways. In gradient boosting, trees are built sequentially, but unlike AdaBoost, each tree tries to reduce the gradient of the loss function. This means that each successive tree focuses on examples that were misclassified or mispredicted by the model, gradually improving performance.
- **eXtreme Gradient Boosting** (**XGBoost**): XGBoost is an advanced implementation of gradient boosting that has gained popularity in ML competitions. It introduces several optimizations and regularizations to improve model performance and stability. XGBoost is known for its speed and can be used for both classification and regression problems.

Each of these boosting techniques has its characteristics and specific adaptations for different data types and ML problems. The choice of boosting algorithm will depend on the specific needs of your problem and the desired performance.

Summary

In this chapter, we studied NLP, which automatically processes information that's transmitted through spoken or written language. To begin, we analyzed the basic concepts of NLP by identifying the tasks that can be tackled and then moved on to the main approaches concerning text analysis and text generation. We then moved on to analyze corpora, words, and sentence tokenization. Corpora offers authentic language data for examination, with words serving as the fundamental components of expression, and sentence tokenization organizing the text into coherent units for in-depth analysis.

In the second part of this chapter, we analyzed a practical case of using NLP for labeling movie reviews. This is a sentiment analysis problem that aims to automatically identify the polarity of a textual comment. In this example, we were able to practically learn which tools to use in MATLAB to perform this type of analysis. In the final part of this chapter, we analyzed ensemble learning techniques to improve the performance of the algorithms. We understood the differences between boosting and bagging and discovered the different boosting techniques.

In the next chapter, we will understand the basic concepts of image processing and computer vision. We will discover the MATLAB tools for computer vision and how to implement a MATLAB model for object recognition. We will also understand transfer learning, domain adaptation, multi-task learning and saliency maps, feature importance scores, and gradient-based attribution methods.

8

MATLAB for Image Processing and Computer Vision

Computer vision is a discipline that explores methods for processing, analyzing, and comprehending visual data. In the realm of image content analysis, a multitude of computer vision algorithms is employed to develop insights into the objects depicted in the image. Encompassing diverse facets of image analysis, computer vision addresses tasks such as object recognition, shape analysis, pose estimation, 3D modeling, visual search, and more. While humans excel at identifying and recognizing objects in their surroundings, the objective of computer vision is to faithfully replicate the capabilities of the **human visual system** (**HVS**) using computational methods. In this chapter, we will understand the basic concepts of computer vision and how to implement a model for object recognition using MATLAB.

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

- Introducing image processing and computer vision
- Exploring MATLAB tools for computer vision
- Building a MATLAB model for object recognition
- Training and fine-tuning pretrained deep learning models in MATLAB
- Interpreting and explaining machine learning models

Technical requirements

In this chapter, we will introduce basic machine learning concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also require a working knowledge of MATLAB.

To work with the MATLAB code in this chapter, you'll need the files available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`.

Introducing image processing and computer vision

Through the five senses, humans gather information from the external world and process it, making decisions to carry out the actions that shape their daily lives. One of the most intriguing challenges in computer science is replicating this sequence of events, identifying and harnessing new sources of information. The ability to acquire and interpret information by simulating the human sensory system is called machine perception, and it is fundamental in the field of artificial intelligence.

Being able to interpret and acquire information from the external world is made possible through techniques such as encoding and information processing. Through digital image encoding techniques, it is possible to represent what humans can perceive in the form of bits. Depending on the methodologies used, it is possible to select the quantity and quality of the information to be represented. Through processing methods, on the other hand, it is possible to interpret the information contained in images and attempt to replicate the mechanisms of human decision-making. One such human ability is the capacity to recognize, through sight, what types of objects are present within a scene, thereby identifying the distinctive characteristics of each object. The highest level of information that can be extracted from images is done through identifying and recognizing individual objects within a scene. This information can be used to categorize and group images based on the objects they contain.

Understanding image processing

For humans, a fundamental sense is vision. Through digital images, it is possible to represent what the HVS captures in an instant numerically. An image is a 2D representation of visual perception; it is perceived in the form of electromagnetic waves that enter the eye and impact the retina. The elements that make up the retina capture information, such as luminance and spectral characteristics. These are transformed into nerve signals to be sent, through the optic nerve, to the brain structures responsible for visual interpretation.

In the digital domain, images are commonly represented as an ordered set of points and pixels, arranged in rows and columns. This mode of representation is called raster or bitmap. Essentially, it involves 2D sampling of a continuous signal in two dimensions:

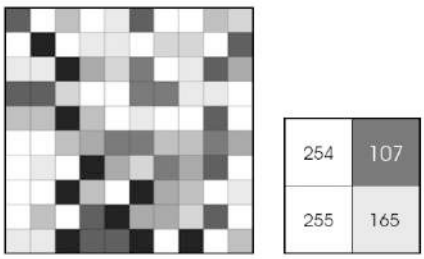


Figure 8.1 – Image representation as a sequence of pixels. Each pixel has a value from 0 (black) to 255 (white)

The simplest technique for representation is the use of grayscale. In this type of representation, pixels contain the amount of luminance. Luminance is a fundamental quantity in the visual field and represents the amount of light that reaches the observer's eye. Pixel values range from absence (black) to the maximum level of light (white), while intermediate states are perceived as varying shades of gray.

In computer graphics, the pixel is the smallest conventional unit of a digital image's surface. The more pixels an image has, the more information it contains, and consequently, our ability to notice details within it increases. The amount of information can be indicated by measuring resolution, either in absolute terms (pixels) or concerning physical measurements (**dots per inch** (**dpi**)).

When deciding to change the resolution of an image, two situations can arise:

- **Constant pixel size**: The number of pixels that make up the image is reduced, resulting in the image shrinking in size
- **Increase in pixel size**: The pixels (dpi) are reduced in size, while the overall dimensions of the image remain constant, causing the individual pixel size to increase

An important aspect of digital images is how to represent the information contained in their pixels. In the case of grayscale, it represents luminance, which needs to be quantized by representing it in a finite number of bits. The greater the number of bits, the lower the quantization noise. Using b bits, there are 2^b possible values. Typically, 8 bits is the most common value, allowing for a total of 256 levels of luminance to be represented. It has been demonstrated that 8 bits provide an acceptable representation of grayscale gradients in most applications. This quantity effectively adapts to the HVS's ability to distinguish different luminance levels in the image.

This holds for grayscale images that contain only luminance information for each pixel. Introducing color increases the complexity of representation because it requires the use of a model to represent it. This model must be able to capture the chromatic information significant to the HVS and translate it into numbers. The goal is to obtain a vector of numbers that "summarizes" the frequencies contained in the electromagnetic wave for each pixel.

The most commonly used model in the digital domain is undoubtedly RGB. It is based on the combination of three chromatic components with different intensities: red, green, and blue.

These components roughly correspond to the three types of cones in the human retina. Therefore, it is not necessary to represent all the color information that exists in the real world, but only the information to which the HVS is sensitive. The information that's carried by this electromagnetic wave corresponds to the light that hits the organs inside the retina; hence, the color corresponds to the spectrum of the electromagnetic signal. Therefore, to represent the signal, only the three components related to the red, green, and blue colors, known as primary colors, which coincide with the luminance in three different frequency bands, are required. The RGB model represents how much energy is present in the spectral bands corresponding to the primary colors and provides this information in the form of three distinct values or components.

Once you understand how it's possible to represent a digital image numerically, it's necessary to know how to process it to facilitate its representation and extract relevant information. Image processing techniques can leverage digital transformation algorithms that modify the pixels of the original image, resulting in a new one. However, they also include techniques that extract numerical or tabular values from the image, representing a particular feature of it. Depending on their complexity, these techniques can be placed into different categories.

The simplest processing methods are those that pertain to the transformation of individual pixels:

- **Grayscale conversion**: This is a type of transformation that allows you to transition from the RGB model to grayscale. There is a linear relationship between luminance and the three chromatic components of the RGB model. This conversion allows us to have simpler pixels to manage and process in subsequent operations. Each pixel will have only one value corresponding to luminance rather than three values, one for each chromatic component (R, G, B) (*Figure 8.2*):

Figure 8.2 – Grayscale conversion of the Flavian amphitheatre

- **Thresholding**: This is a very useful transformation in the image segmentation phase, where the goal is to isolate an object of interest from the background. The idea is to set pixels above a certain threshold value equal to the maximum luminance intensity value, while pixels below the threshold are set to the minimum intensity value.
- **Aliasing**: This is an effect that makes two different signals that are indistinguishable during sampling. Aliasing occurs when sampling or interpolation produces a lower resolution in the image, distorting the output compared to the original signal. Anti-aliasing filters can be used to correct this problem. In the case of a digital image, aliasing manifests as a moiré effect or a wavy pattern.

Then, there are the direct comparison and feature extraction methods. The first group of techniques is used to compare two images pixel by pixel, obtaining a value that measures the discrepancy between them. The second, on the other hand, allows for creating a summary of the initial image and using a smaller dataset that still adequately describes the original set. Let's see something:

- **Direct comparison methods**: One of the most interesting pieces of information that can be derived from images is their degree of similarity. The human brain can process the information contained in visual perceptions that arrive through the retina. There are dedicated neurons for interpreting visual information (shape, color, motion, space, lines, and so on). Recognition, for example, of faces and objects, occurs only after this information is extracted and memory is accessed. One type of recognition methodology is the direct approach. This method of comparing images is also called brute force as it involves comparing each pixel in the two images. These methodologies apply algebraic formulas and calculate a discrepancy index that indicates the degree of similarity between two images. If we have two images called *image1* and *image2*, we can calculate the discrepancy index using the following formula:

$$discrepancy_index = 1 - sum(sum(abs(image1 - image2))) / sum(sum(image1))$$

- **Mean squared error** (**MSE**): This is a value that indicates, in an absolute sense, how similar two images are. This index compares the images pixel by pixel and represents the average discrepancy of these values. The closer the MSE value is to 0, the greater the similarity between the analyzed images.
- **Structural similarity index measure** (**SSIM**): Another method of direct comparison is SSIM. The difference compared to MSE is that the latter makes estimates using absolute errors. SSIM, on the other hand, is a perception-based model that considers image degradation as a change in the perception of structural information. Instead of performing a pixel-by-pixel comparison, the algorithm divides the image into grids of N x N pixels. Within each grid, an average value of the pixels is calculated, allowing for relationships between neighboring pixels to be considered rather than the absolute value of a single pixel.

Direct comparison methodologies come with a significant cost associated with the amount of data to process as they consider all the information contained in the image. Through some techniques, it's possible to reduce dimensionality. When there's an excessive amount of data to process and the possibility of redundancy, a transformation can be applied, adopting a reduced representation of the data. This reduced representation is nothing but a set of features. The process that transforms the input data into the feature set is called **feature extraction**.

The chosen characteristics encompass pertinent details from the input data, enabling the intended task to be accomplished by utilizing this condensed representation rather than the entirety of the original data. This method minimizes the cost and resources needed for an accurate depiction of a vast dataset. When tackling intricate data analysis, a primary hurdle involves diminishing the number of variables in play. Analyzing a large number of variables usually translates into high memory usage and computational power requirements. Moreover, when applying classification algorithms, there is a risk of **overfitting**. In this case, the model adapts too closely to the dataset that was used for learning, failing to generalize and thus losing effectiveness. Feature extraction is a broad term that's used to describe methods of creating variable combinations aimed at addressing these issues while maintaining sufficient accuracy in depicting the data. Now that we've introduced the most widespread image processing methodologies, we can focus on how to extract knowledge from images.

Explaining computer vision

Computer vision is an interdisciplinary field of computer science and artificial intelligence that deals with the development of algorithms, models, and computer systems capable of interpreting, understanding, and analyzing visual information from images or videos. The main goal of computer vision is to replicate some of the capabilities of the HVS, allowing computers to perceive and understand the world around them through visual data.

This field of study focuses on a wide range of tasks, including object recognition, motion detection, pattern recognition, information extraction from images, image segmentation, 3D reconstruction, and much more. Computer vision has applications in a wide range of industries, including medicine, automotive, security, manufacturing, retail, entertainment, and robotics, to name a few.

Recent developments in computer vision have been driven by the use of deep neural networks, in particular **convolutional neural networks** (**CNNs**), which have delivered exceptional results in many visual recognition tasks. These advances are fundamentally changing the way machines can interact with the visual world, opening up new possibilities in areas such as autonomous driving, medical diagnosis, augmented reality, and much more.

This task is the most complex in terms of abstraction. The most common use of this category is for object recognition. Conceptually, the process can be divided into two steps. The first step involves defining an object of interest according to a model using feature extraction techniques. In the second step, the object is searched for within an image. These types of transformations require the use of machine learning and data mining algorithms that, through a dataset, allow a model to be constructed for the object to be searched for. Subsequently, it can be determined whether there are pixels within the image that match the previously created model. There are different approaches to this:

- **Shape matching**: This is an approach that involves searching for a silhouette. This method allows you to measure the similarity between shapes and identify correspondences between points that belong to the contour of the object being searched for. The basic idea is to select *n* points on the contour of the silhouette. For each point, the n - 1 vectors connecting it to all the others are considered. A set of all these vectors forms a complex descriptor of the silhouette localized at that point. This set of vectors is obtained through a shape extraction process, which is part of feature extraction. The idea is to obtain a descriptor using these vectors and use it to identify a similar shape within other images and perform classification:

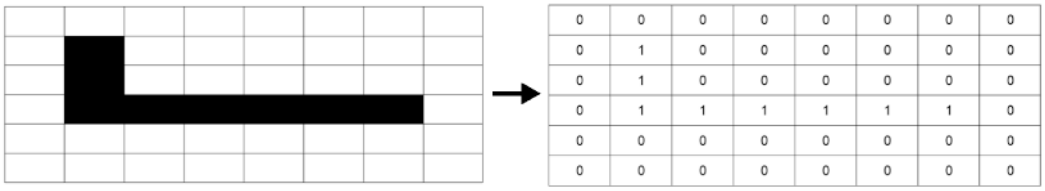


Figure 8.3 – Shape matching

- **Pattern matching**: This is a technique that involves identifying a specific sequence or regularity of data (referred to as a pattern) within a large dataset. The field of pattern recognition involves automatically searching for regularities within data using computer algorithms and patterns to perform actions such as classifying data into different categories. In digital images, the identification process involves preparing a pattern and corresponding to a set of pixels that describes a specific object of interest or a part of it. Then, a pixel classification process is performed within the image to determine whether a group of pixels comparable to the pattern being searched for exists or not.
- **Feature-based object recognition**: Through this technique, it is possible to create descriptors that represent the typical characteristics of an object. Each object has unique features that describe it. If we can identify all these features within a dataset, we can assume that the object is present. For example, a human face can be modeled based on specific anatomical features such as the placement of eyes, nostrils, angles formed by the lips, and so on. The combination of these anatomical elements and the vectors that connect them forms a patch model. This model represents the ordered set of elements that are sufficient to accurately describe an object.

It's interesting to understand how object recognition technology within images has evolved and the impact it has had on society. One of the most fascinating cases that leverages image processing techniques is facial recognition. Until not long ago, this technology was commonly seen as something straight out of science fiction. However, in the last decade, it has not only become a reality but also become widely used.

Now that we've analyzed the basic concepts connected to image processing and computer vision, let's analyze the necessary tools to address these problems in MATLAB.

Exploring MATLAB tools for computer vision

Computer vision encompasses the development of algorithms, techniques, and systems that allow computers to acquire, process, analyze, and make decisions based on visual data from images and videos. The primary goal of computer vision is to enable machines to perform tasks that typically require human visual perception and comprehension.

Computer vision can automate various tasks that would be time-consuming or even impossible for humans to perform consistently and at scale. This includes tasks such as object detection, image classification, and tracking. When trained and configured properly, computer vision algorithms can achieve high levels of accuracy in tasks such as image recognition and segmentation. They don't suffer from fatigue or distraction, leading to consistent results. These algorithms can process images and videos in real time or near real time, making them suitable for applications that require rapid analysis and decision-making.

Computer vision can also be applied to a wide range of industries and applications, from healthcare and automotive to agriculture and manufacturing. It can adapt to various domains with appropriate training. These systems can scale easily to handle large volumes of data and images, making them suitable for big data applications. In applications such as medical imaging, computer vision provides a non-invasive way to diagnose and monitor conditions without the need for invasive procedures.

In contrast, these algorithms heavily rely on large datasets for training. Insufficient or biased training data can lead to poor performance and inaccurate results. Developing and fine-tuning computer vision models can be complex and time-consuming. It often requires expertise in machine learning, deep learning, and image processing. Deep learning models used in computer vision can be computationally intensive and require powerful hardware, such as GPUs, for training and inference. Deep learning models, especially CNNs, are often perceived as black boxes, posing challenges in interpreting the rationale behind their specific decisions.

Computer vision systems may struggle to perform well under adverse conditions, such as poor lighting, occlusions, or variations in camera angles. The use of computer vision in surveillance and facial recognition has raised concerns about privacy and potential misuse. These algorithms may adopt biases embedded in their training data, resulting in unjust or discriminatory outcomes. Ensuring fairness and mitigating bias remains a persistent challenge. While computer vision can identify and classify objects, it often lacks a deep understanding of the context in which those objects appear, which can limit its usefulness in some scenarios.

Overall, computer vision is a powerful and rapidly evolving field with the potential to transform many industries. However, it's essential to be aware of its limitations and challenges and to apply it responsibly and ethically, addressing issues related to data quality, privacy, and bias.

MATLAB provides a variety of tools and functions for computer vision tasks. These tools can be found in Computer Vision Toolbox, Image Processing Toolbox, and other related toolboxes. Here are some of the key MATLAB tools and functionalities for computer vision:

- **Computer Vision Toolbox**: This toolbox is specifically designed for computer vision tasks. It includes a wide range of functions and algorithms for image processing, feature extraction, object detection and recognition, 3D vision, camera calibration, and more.
- **Image Processing Toolbox**: While not exclusively for computer vision, this toolbox is often used in conjunction with Computer Vision Toolbox. It provides fundamental image processing functions such as filtering, morphological operations, and image enhancement.
- **Camera calibration**: MATLAB offers tools for camera calibration, which is essential for mapping 2D image points to 3D world coordinates. This is crucial for tasks such as 3D reconstruction and object tracking.
- **Object detection and recognition**: MATLAB provides functions and pretrained models for object detection and recognition. You can use popular deep learning models such as YOLO, SSD, and Faster R-CNN for these tasks.

- **Feature extraction**: MATLAB supports feature extraction techniques such as **Scale-Invariant Feature Transform** (**SIFT**), **Speeded-Up Robust Features** (**SURF**), and **Histogram of Oriented Gradients** (**HOG**) for object detection and matching.
- **Stereo vision**: MATLAB supports stereo vision techniques for depth estimation and 3D reconstruction from stereo camera setups.
- **Motion analysis**: You can perform motion analysis tasks such as optical flow estimation, tracking, and motion segmentation using MATLAB functions.
- **Machine learning and deep learning**: MATLAB integrates with various machine learning and deep learning frameworks, making it suitable for training custom models for computer vision tasks.
- **Semantic segmentation**: MATLAB includes tools for semantic segmentation, which is the process of labeling each pixel in an image with the class it belongs to.
- **Point cloud processing**: For 3D point cloud data, MATLAB provides tools for visualization, manipulation, and analysis.
- **Apps**: MATLAB offers interactive apps for tasks such as image labeling, camera calibration, and object training, which simplify the development workflow.
- **Parallel computing**: MATLAB supports parallel computing, allowing you to speed up computationally intensive computer vision tasks by leveraging multiple CPU cores or GPUs.

These tools and functions make MATLAB a powerful environment for developing and prototyping computer vision applications, whether you're working on image analysis, object detection, 3D reconstruction, or any other related tasks. Now, let's learn how to recognize an object using MATLAB and CNN.

Building a MATLAB model for object recognition

An enduring challenge within the realm of computer vision involves ascertaining the presence of specific objects (object recognition) or activities within an image. For objects under predefined conditions, such as the identification of specific geometric shapes such as polyhedral or the recognition of faces and handwritten characters, this problem can be tackled effectively and without significant hurdles. However, the complexity escalates when dealing with arbitrary objects in unrestricted scenarios.

Object recognition entails the capacity to detect a particular object within a series of images or videos. Human beings possess the remarkable ability to identify various objects in images effortlessly, even when the objects' appearances may vary. Moreover, objects can be recognized even when they are partially obscured from view. However, this remains a formidable challenge for computer vision.

Each object within an image exhibits a multitude of intriguing characteristics that can be extracted to construct a comprehensive description of the object. This description serves to identify the object when seeking it within a test image containing multiple objects. Crucially, the set of characteristics that are extracted from the reference image must be resilient to variations in image scale, disturbances, lighting conditions, and geometric distortions to ensure dependable recognition. CNNs excel in this endeavor, offering algorithms with exceptional object recognition performance.

Introducing handwriting recognition (HWR)

HWR encompasses the computer's capability to receive and comprehend handwritten input, transforming it into readable text. This input can originate from various sources, including paper documents, photographs, and touchscreens. Detection of written text can be accomplished through optical scanning, which involves **optical character recognition** (**OCR**), or through intelligent word recognition techniques.

We have long been acutely aware of the challenge of automating HWR to facilitate smoother interactions between humans and machines. In recent years, this challenge has witnessed intriguing advancements and increasingly efficient solutions, primarily fueled by substantial economic interest and the growing computational capabilities of modern computers. Notably, certain countries, such as Japan and various other Asian nations, have made significant investments in research and financial resources to pioneer cutting-edge OCR technologies.

The rationale behind the enthusiasm of these countries in this research domain is quite evident. Their goal is to develop devices capable of interpreting the intricate ideograms that characterize their respective cultures, thereby enhancing the ease of interaction with machines. Given that there are currently no input devices, such as keyboards, capable of representing thousands of characters, the focus is on acquiring this information directly from handwritten scripts through digitized scanning.

Nevertheless, even in Western countries, substantial attention has been dedicated to the field of optical HWR. There exist numerous applications that stand to benefit from automated text interpretation. Consider, for instance, the automatic parsing of preprinted templates or the recognition of addresses and postal codes on envelopes, which are just a few instances where OCR technology proves invaluable.

HWR is accomplished using diverse techniques that typically involve OCR. Nevertheless, a comprehensive script recognition system goes beyond OCR and encompasses tasks such as formatting, accurate character segmentation, and identifying the most probable words.

To better understand how HWR can be approached using machine learning methodologies, we will use a very popular dataset that's largely used in the community to address this type of topic. This is the **Modified National Institute of Standards and Technology** (**MNIST**), a large database of handwritten digits. This dataset consists of 70,000 data examples, which is a subset of a larger dataset maintained by NIST. These examples represent digits and are in a format of 28 x 28-pixel resolution, organized as a matrix with 70,000 rows and 785 columns. In each row, there are 784 columns corresponding to

pixel values from the 28 x 28 matrix, and one column containing the actual digit label. These digits have been size-normalized and positioned at the center of a fixed-size image.

The digit images within the MNIST dataset were originally chosen and processed by Chris Burges and Corinna Cortes, who employed bounding box normalization and centering techniques. Yann LeCun's version of the dataset, on the other hand, utilizes centering based on the center of mass within a larger window.

This dataset is already available in the MATLAB environment, in a short version with only 10,000 images evenly distributed over the 10 digits (0-9). Let's get started:

1. First, we will import the dataset into the MATLAB workspace. The dataset is available in the standard MATLAB installation with Deep Learning Toolbox. Under our MATLAB installation folder, we will find the `toolbox\nnet\nndemos\nndatasets\DigitDataset` path. The `DigitDataset` folder contains 10 subfolders, each of which contains 1,000 images of a single digit; each folder is named after the digit it contains. To find the path in an automated way, we can use the following command:

```
FolderPath = fullfile(toolboxdir('nnet'),'nndemos',
'nndatasets','DigitDataset');
```

   The `fullfile()` function constructs a complete file path by combining the provided folder names and filenames. We have used the sequence of the folder, starting from the toolbox directory. The output can be in the form of a character array, a string array, or a cell array of character vectors. If any of the input arguments are in the form of a string array, the output will also be a string array. Conversely, if any of the input arguments are in the form of a cell array containing character vectors, the output will be a cell array of character vectors. In all other cases, it will be a character array. Now that we have the path, we can extract the image dataset:

```
DigitData = imageDatastore(FolderPath,
'IncludeSubfolders',true,'LabelSource','foldernames');
```

   The `imageDatastore()` function generates a data store by incorporating the dataset's path for a specified collection of image data. We passed three arguments:

   - `FolderPath`: The path of the folder containing the images
   - `IncludeSubfolders`: The possibility to include all the subfolders in the main folder
   - `LabelSource`: We used the folder names to label the data

   An `ImageDatastore` object will be created with some properties. These properties delineate the characteristics of the data and provide instructions on how to retrieve data from the data store. When creating the data store object, you have the option to set these properties using name-value pairs as arguments. If you wish to inspect or adjust a property after the object's creation, you can do so using the dot notation.

Let's look at some of these properties:

- `Files`: This is a cell array that stores the file paths of all the images in the data store. You can access this property to get a list of file paths.
- `Labels`: This is an array or cell array that associates labels or categories with each image in the data store. This property is often used for image classification tasks.
- `ReadSize`: This specifies the number of images to read at once during data iteration. This can impact memory usage and performance.

Now, we can display a selection of images that have been loaded via a random selection process:

```
figure;
RandId = randperm(10000,9);
for i = 1:9
    subplot(3,3,i);
    imshow(DigitData.Files{RandId(i)});
end
```

Here, we have used the `randperm()` function, which generates a row vector comprising nine distinct random integers chosen from a range of 1 to 10,000. Each number that was generated was used as an index to identify an image file path stored in the `Data.Files` property. *Figure 8.4* shows a collage of nine images extracted from the database:

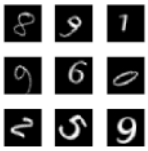


Figure 8.4 – The MNIST dataset

Now, we can check the distribution of the images over the classes:

```
ClassDist = countEachLabel(DigitData)
```

The following results are returned:

```
ClassDist =
  10×2 table
    Label    Count
    _____    _____
      0      1000
      1      1000
      2      1000
      3      1000
      4      1000
      5      1000
      6      1000
```

```
    7      1000
    8      1000
    9      1000
```

With this, we have proof that the images are evenly distributed across the 10 digits (0-9).

2. Before we start building the machine learning algorithm, it's imperative to partition the available data into two distinct subsets. The first subset will serve as the training data, while the second will be earmarked for algorithm validation. Data partitioning plays a pivotal role in machine learning and data analysis as it involves segregating a dataset into multiple subsets for training, validation, and model testing. In our case, we have 10,000 samples, each containing 1,000 images for a specific digit. Our chosen approach is to split the data into a 70% portion for training and a 30% portion for validation. This split rate is used because most of the data must be used for training the network. Accordingly, we will employ 7,000 samples for training and reserve the remainder for validation:

```
SplitRate = 0.7;
[TrainDat,ValDat] =
splitEachLabel(DigitData,SplitRate,'randomize');
```

To achieve this, we utilized the `splitEachLabel()` function. This function effectively separates the image files within the `Data` dataset into two separate data stores: `TrainDat` and `ValDat`. The `TrainDat` data store consists of an initial portion of each category determined by `SplitRate`, while the `ValDat` data store contains the remaining images from each class. `SplitRate` can take the form of a fractional value ranging from 0 to 1, indicating the proportion of images allocated to `TrainDat`, or it can be an integer denoting the exact count of images assigned to `TrainDat` for each class.

3. Let's begin constructing our convolutional network. As expected, a CNN consists of a sequence of interconnected layers. To commence, you'll need to employ a layer so that you can import your input data:

```
layers = [
    imageInputLayer([28, 28, 1])
```

The `layers` variable is an array that contains the list of layers for our CNN, defining the architecture of the neural network used for deep learning. To initiate this architecture, we begin with `imageInputLayer`. This layer functions as an input for images, accepting 2D image data into the neural network and performing data normalization. Additionally, this layer specifies the unmodifiable `InputSize` attribute. This attribute contains the height, width, and number of channels, respectively. In this case, we are working with grayscale images with a height of 28 and a width of 28. Following the input layer, we establish the initial block of three consecutive layers:

```
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
```

In this code, we have the following:

- `convolution2dLayer`: This layer employs 2D convolutional filters that slide across 2D input data. The following parameters are passed:
  - `3`: Filter size.
  - `8`: Number of filters.
  - `Padding`: This is a method that involves augmenting the input image or feature map with additional rows and columns of pixels before conducting convolutional operations. Its primary purpose is to regulate the spatial dimensions of the resulting feature maps following the convolution process.
  - `same`: This specifies that you want the output feature map to have the same spatial dimensions as the input image. This means that the necessary padding will be added automatically to achieve this.
- `batchNormalizationLayer`: This layer facilitates batch normalization, standardizing a mini-batch of data independently across all observations for each channel.
- `reluLayer`: The ReLU layer performs a thresholding operation on each input element, setting any value less than 0 to 0.

Following this initial block, we proceed by applying a pooling layer, as outlined here:

```
maxPooling2dLayer(2,'Stride',2)
```

Downsampling is accomplished via a 2D max pooling layer, which divides the input into rectangular pooling regions and subsequently identifies the maximum value within each of these regions.

4. Next, we will introduce a second set of layers, akin to the first set, with adjusted parameters:

```
convolution2dLayer(1,16,'Padding','same')
batchNormalizationLayer
reluLayer
```

A new pooling layer follows:

```
maxPooling2dLayer(2,'Stride',2)
```

Then, we add a third block of layers, to finish the architecture:

```
convolution2dLayer(3,32,'Padding','same')
batchNormalizationLayer
reluLayer
```

After the three blocks, we add a **fully connected** (**FC**) layer:

```
    fullyConnectedLayer(10)
```

In an FC layer, the input undergoes multiplication with a weight matrix and is subsequently adjusted by a bias vector. The "parameter" in this context specifies the desired output size, which, in our case, is 10 since we are classifying `10` distinct digits. This layer type mirrors the layer configuration commonly found in a traditional **artificial neural network** (**ANN**) employing an FC architecture. Within an FC layer, every neuron establishes direct connections with all neurons from the preceding layer, engaging directly with their respective activation values.

Following this, we require a `softmax` layer:

```
softmaxLayer
```

`softmaxLayer` is a specialized layer that's employed within neural networks, designed specifically to implement the `softmax` function on its input. The `softmax` function finds extensive use in classification tasks as it transforms raw scores or logits into a probability distribution spanning multiple classes. Typically, this layer serves as the concluding component in a neural network designed for multi-class classification. It transforms the network's output values into probabilities that collectively sum to `1`, simplifying the interpretation of the model's predictions.

To complete the network, a classification layer was added:

```
  classificationLayer];
```

The classification layer calculates cross-entropy loss for both regular and weighted classification tasks that pertain to distinct and mutually exclusive classes. It automatically deduces the number of classes by inspecting the dimensions of the output originating from the preceding layer.

5. Before training the CNN, we need to configure the settings:

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',50, ...
    'Shuffle','every-epoch', ...
    'ValidationData',ValDat, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

To learn more about the various training options, please read *Chapter 6*, *Deep Learning and Convolutional Neural Networks*.

6. It's time to train the network:

```
CNNnet = trainNetwork(TrainDat,layers,options);
```

The following plot was printed on the screen (*Figure 8.5*):

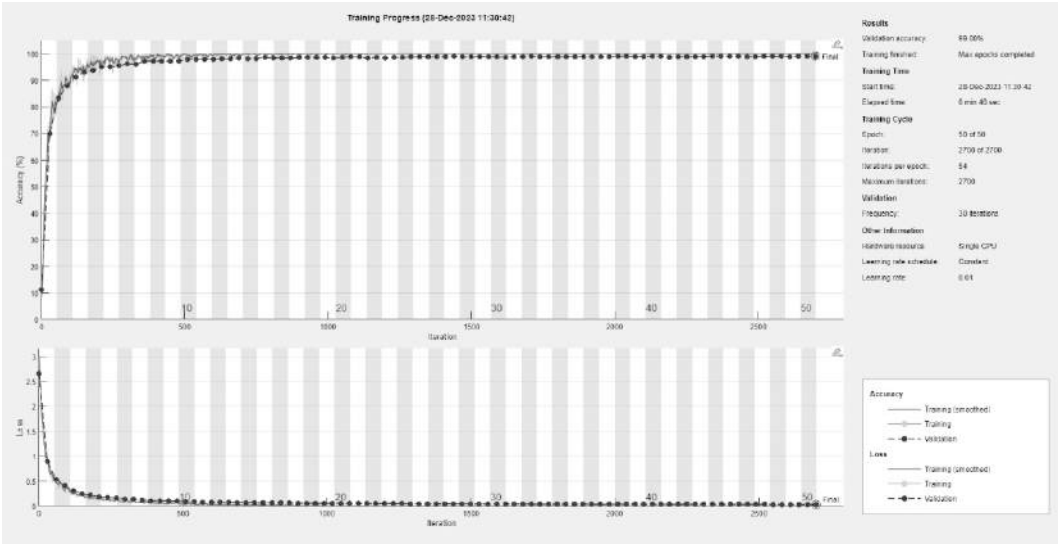


Figure 8.5 – Training process of the CNN for handwritten digit recognition

This plot will undergo continuous updates as the training progresses, enabling us to monitor how the algorithm adapts the weights to achieve convergence. We can see the results with an accuracy of 84.1%, which indicates a good result.

It is natural to ask whether it is possible to improve the performance of the handwritten digit recognition model. In the next section, we will see how to improve the accuracy of the model using transfer learning.

Training and fine-tuning pretrained deep learning models in MATLAB

Transfer learning is a machine learning approach wherein a model created for a particular task is repurposed as the initial foundation for a model addressing a second task. This technique entails leveraging knowledge acquired from one problem and applying it to a distinct yet related problem. Transfer learning is particularly useful in deep learning and neural networks, where pretrained models can be fine-tuned or used as feature extractors for new tasks.

In pretrained models, you start with a pretrained model that has been trained on a large dataset for a specific task, such as image classification, natural language processing, or speech recognition. These pretrained models are often complex neural networks with many layers. In many cases, you can use the layers of the pretrained model as feature extractors. You remove the final classification layer(s) and use the activations from the earlier layers as features for your new task. This is especially common in computer vision tasks. Optionally, you can fine-tune the pretrained model on your specific task by training it further with your own dataset. This involves updating the weights of some or all layers while keeping the knowledge you've learned from the original task.

Transfer learning can significantly reduce the amount of data and time required to train a model for a new task, especially when you have a limited dataset. Pretrained models have already learned useful features from large and diverse datasets, which can be valuable for related tasks. It can help improve model performance when you have limited computational resources or limited labeled data.

Transfer learning is commonly used in various fields, including computer vision, natural language processing, and audio processing, and has been a key technique in advancing the state of the art in machine learning applications.

Introducing the ResNet pretrained network

ResNet stands for Residual Network, a profound CNN architecture introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their 2015 paper titled *Deep Residual Learning for Image Recognition*. This groundbreaking architecture has had a substantial impact on the domains of computer vision and deep learning.

The key innovation in ResNet is the use of residual blocks. In traditional deep neural networks, as the network becomes deeper, it becomes increasingly difficult to train. This is because of the vanishing gradient problem, where gradients become extremely small as they are propagated back through the network during training. As a result, deep networks tend to suffer from degradation in performance as their depth increases.

ResNet addresses this problem by introducing residual blocks, which contain skip or shortcut connections that allow the gradient to flow more easily through the network. These shortcut connections bypass one or more layers, making it easier to train very deep networks. The skip connections essentially learn the residual (the difference) between the output and input of a layer, hence the name "residual network."

We can summarize the following key features of ResNet:

- **Deep architecture**: ResNet can be very deep, with hundreds or even thousands of layers, thanks to the use of residual blocks.
- **Skip connections**: The skip connections allow gradients to propagate effectively, mitigating the vanishing gradient problem.
- **High accuracy**: ResNet achieved state-of-the-art performance on various image classification tasks, including the ImageNet Large Scale Visual Recognition Challenge.
- **Transfer learning**: Pretrained ResNet models are widely used as feature extractors or starting points for various computer vision tasks through transfer learning.
- **Architectural variations**: There are several ResNet architectures with different depths, such as ResNet-18, ResNet-34, ResNet-50, and deeper variants.

ResNet has become a foundational architecture in deep learning, and its principles of skip connections and residual learning have influenced the design of many subsequent neural network architectures. It has been applied not only to image classification but also to various other computer vision tasks, including object detection, semantic segmentation, and more.

The MATLAB Deep Network Designer app

The MATLAB Deep Network Designer app is a **graphical user interface** (**GUI**) tool provided by MATLAB for designing, training, and analyzing deep neural networks. It's part of the MATLAB

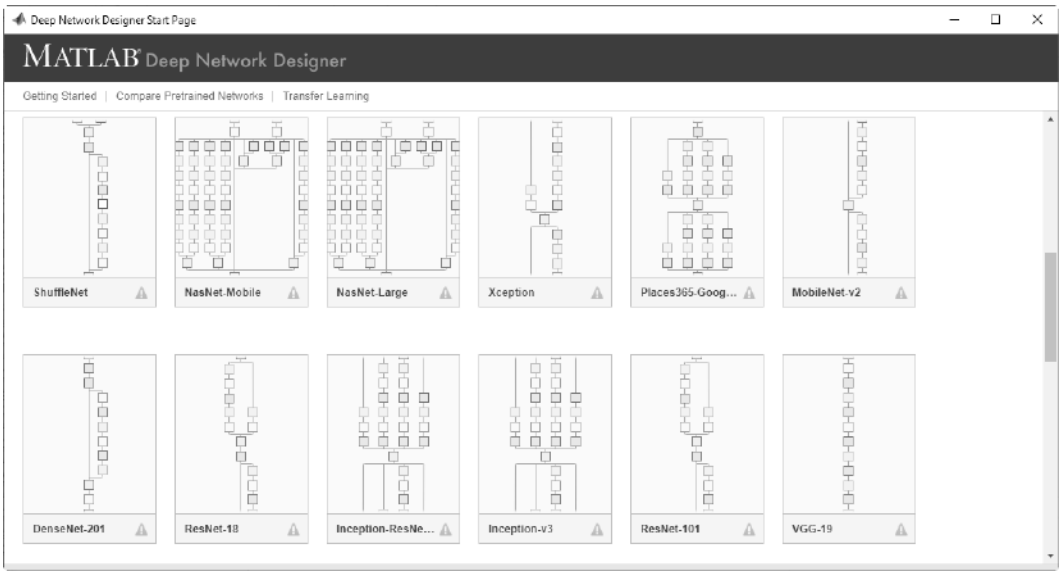


Figure 8.6 – Deep Network Designer Start Page

If you don't find the network, you can use the following command:

```
deepNetworkDesigner(resnet18)
```

In the image pretrained network, we will find ResNet-18. ResNet-18 consists of 18 layers, which include convolutional layers, residual blocks, and FC layers. It's considered relatively shallow compared to deeper variants such as ResNet-50 or ResNet-101. Like all ResNet architectures, ResNet-18 employs residual blocks. These blocks contain skip connections (or shortcut connections) that allow gradients to flow more effectively during training, addressing the vanishing gradient problem. ResNet-18 has been widely adopted in the deep learning community due to its balance between model complexity and performance. It's often used in tasks such as image classification, object detection, and feature extraction in various computer vision applications. Its architectural principles, such as residual learning, have influenced the design of many subsequent neural network architectures. To import ResNet-18, it is necessary to install the relative toolbox.

2. After importing it, we can import the dataset that we'll train the network on. To do that, we can move to the **Data** tab of the app. Upon clicking the **Import Data** icon, the following window will open (*Figure 8.7*):

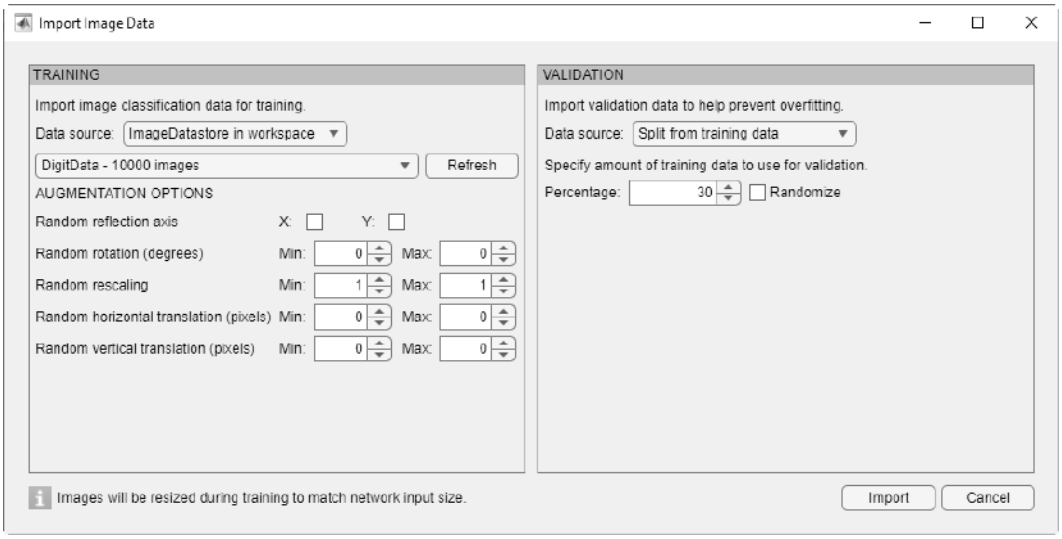


Figure 8.7 – The Import Image Data window

Under **Data source**, we can select **ImageDatastore in workspace** to select the data that's already been imported into the MATLAB workspace, as indicated in the previous section. The MNIST dataset will be imported, as shown in the *Figure 8.8*:

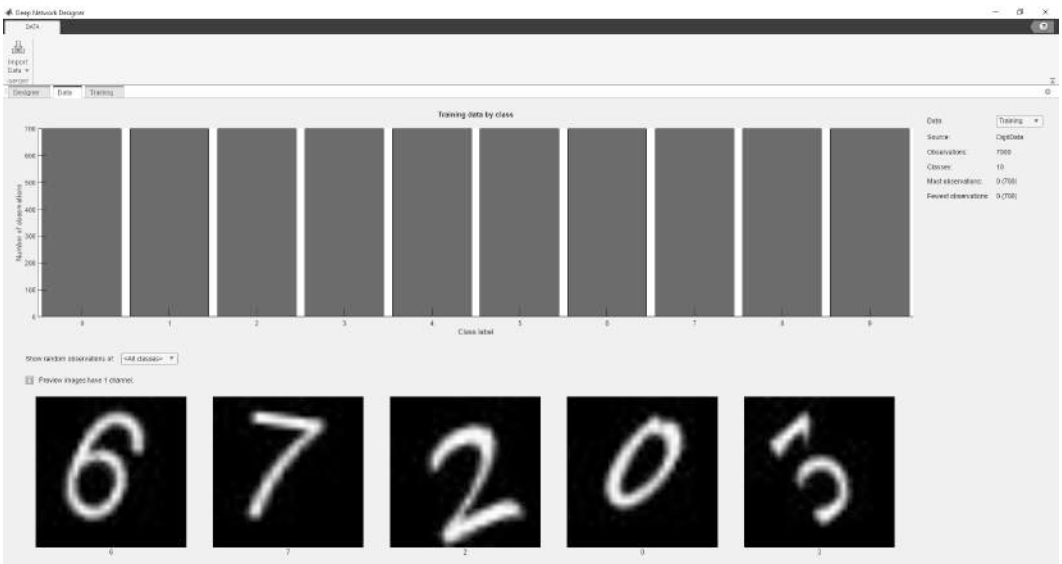


Figure 8.8 – The MNIST dataset imported into the Deep Network Designer app

In *Figure 8.8*, we can see that a good distribution of the images is regularly present in the 10 classes, with 700 images for each class. We will use 70% of the data for training and the rest for validation.

3. At this point, we can set the pretrained network (ResNet-18) by moving to the **Designer** tab. We'll use ResNet-18 for another type of image (an RGB image whose size is 227 x 227 x 3). To do this, we have to change the first layer, which defines the size of the input data. Click on the first layer (**ImageInput**) and then click on the **Canc** button to remove this layer.
4. After that, we can click on the **ImageInputLayer** icon in **Layer Library** to the left of the tab, at which point we have to connect this layer with the next layer.
5. After that, we have to change the **InputSize** property by clicking on the layer and modifying it to `28,28,1` in the **Properties** window to the right of the **Designer** tab (*Figure 8.9*).
6. After that, we have to change the first convolutional layer. First, we must remove that and then drop a **Convolution2Dlayer** and connect it to another layer. Then, we have to set **FilterSize** to `3,3` and **NumFilters** to `64` in the **Properties** window to the right of the **Designer** tab (*Figure 8.9*):

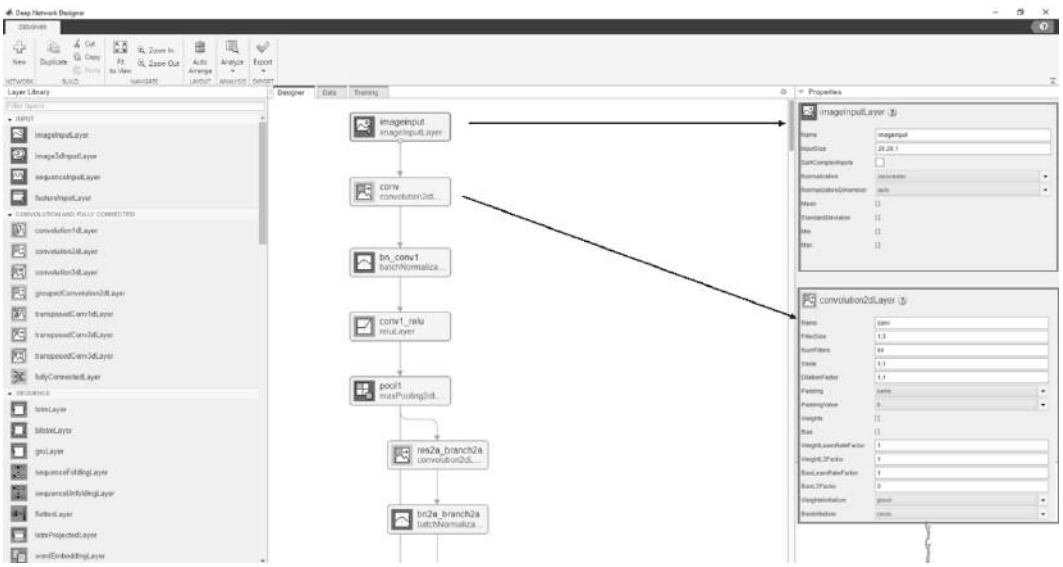


Figure 8.9 – Modifying the first two layers of ResNet-18 to adapt it to the new input data

7. Now, we have to move on to the final part of ResNet-18, which involves setting the classification option. To do that, we have to replace the FC layer and the classification layer, as shown in *Figure 8.10*:

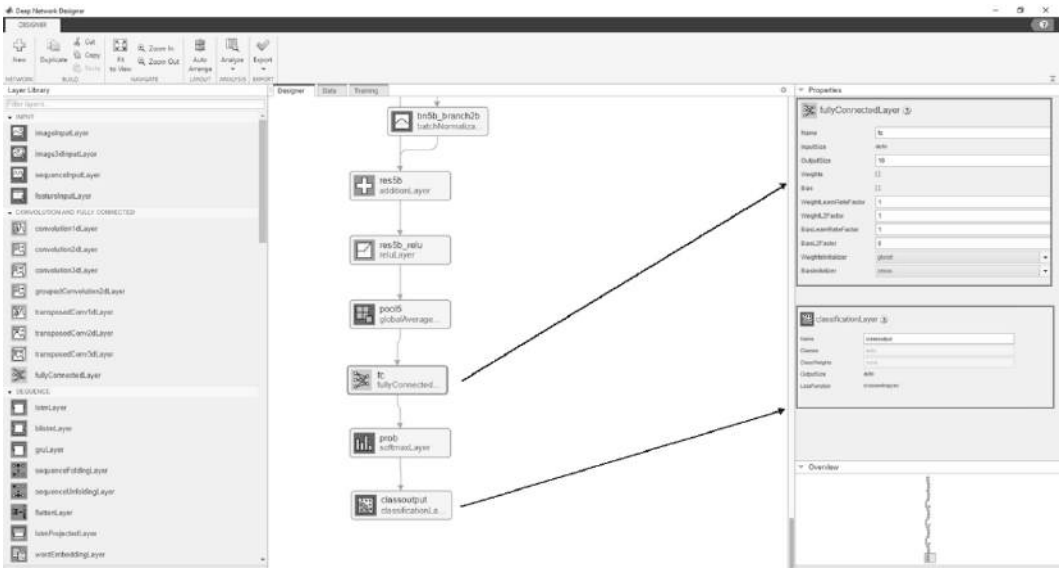


Figure 8.10 – Modifying the final layers of ResNet-18 to adapt it to the new classification

To check that the layer has been modified correctly, we can test it by clicking on the **Analyze** icon at the top of the **Designer** tab.

8. Now that we are ready to train the model, we can move to the **Training** tab of the Deep Network Designer app. We can check the training option by clicking on the **Training option** icon at the top of the **Training** tab. After that, we can push the **Train** button at the top of the tab. The training process will start; we will check its progress in the **Training Progress** window (*Figure 8.11*):

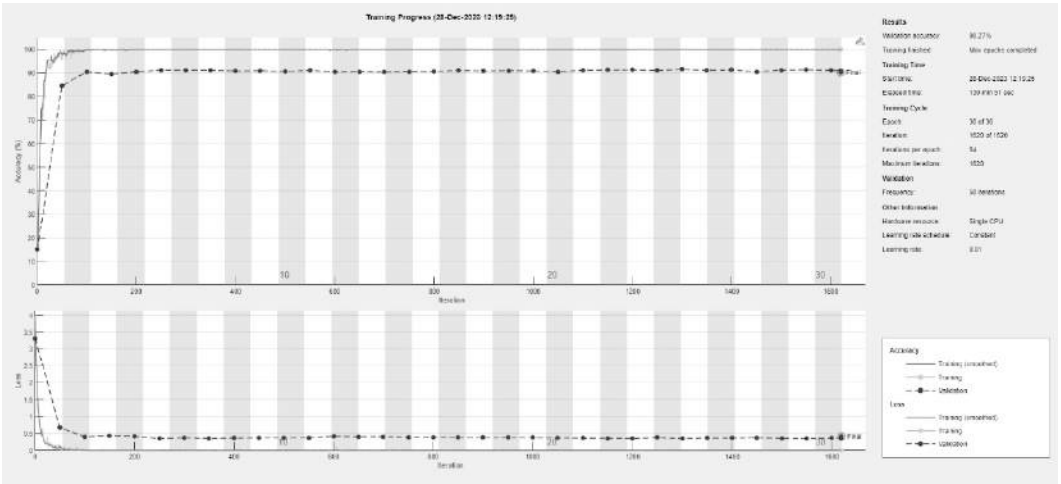


Figure 8.11 – The Training Progress window

At the end of the training process, we will be able to verify the performance of the model by reading the accuracy value that was obtained in the validation procedure. As we can see, we obtained an accuracy of 90.27%.

Now, let's try to collect some useful information on how to interpret the results that have been obtained from a model based on machine learning.

Interpreting and explaining machine learning models

Interpreting and explaining machine learning models is essential for understanding their predictions and making them more transparent and trustworthy, especially in applications where interpretability is critical. This is an ongoing process that requires collaboration between data scientists, domain experts, and stakeholders. The choice of interpretation techniques depends on the model type, problem domain, and level of transparency required for the application. It's important to strike a balance between model complexity and interpretability, depending on the specific use case.

Understanding saliency maps

Saliency maps are a visualization technique that's used in computer vision and deep learning to understand and interpret neural network predictions, particularly in image classification and object recognition tasks. Saliency maps help identify which regions of an input image or feature map are most relevant to a model's prediction. They are especially useful for gaining insights into why a neural network is making a particular decision.

Saliency maps are generated using gradient-based methods, typically backpropagation. The idea is to compute the gradients of the model's output concerning the input image's pixels. By calculating these gradients, you can identify which pixels in the input image have the most significant impact on the model's prediction. In other words, saliency maps highlight the regions that the model pays attention to when making a decision. Saliency maps are usually visualized as heatmaps overlaid on the original input image. In a heatmap, the intensity of color corresponds to the importance of each pixel. High-intensity areas indicate regions that strongly influence the model's output:

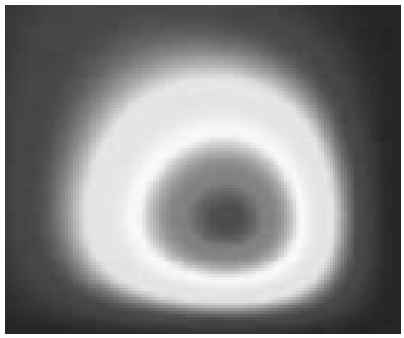


Figure 8.12 – A saliency map as a heatmap

Saliency maps provide interpretability to neural network predictions. By examining the saliency map, you can see which parts of the image resemble specific features and contribute to the decision. These maps can be used for model debugging and improvement. If the model's predictions appear incorrect, examining the saliency map can reveal whether the model is focusing on the right or wrong features. There are variations of saliency maps, such as class-specific saliency maps (highlighting features specific to a particular class) and gradient-based approaches such as guided backpropagation and SmoothGrad, which improve the interpretability of saliency maps.

It's important to note that saliency maps provide insights into a model's behavior but do not necessarily explain why a neural network made a particular decision in human-understandable terms. They are just one tool in the interpretability toolbox and are often used in conjunction with other techniques for a more comprehensive understanding of model decisions.

Understanding feature importance scores

Feature importance scores are a set of metrics or values that indicate the relative importance of different input features (also known as variables or attributes) in a machine learning model's prediction. These scores help data scientists and analysts understand which features have the most significant influence on the model's output. Feature importance scores are particularly valuable for feature selection, model interpretation, and debugging.

There are some common methods and techniques for calculating feature importance scores:

- **Gini importance**: In decision trees and random forests, Gini importance measures how often a feature is used for splitting data across the tree nodes. Higher values indicate more important features.
- **Mean decrease in impurity**: Similar to Gini importance, this metric calculates how much the impurity (or impurity reduction) decreases when a particular feature is used for splitting.
- **Coefficient magnitude**: In linear models, the magnitude (absolute value) of the coefficients represents the feature's importance. Larger coefficients indicate greater importance.
- **Permutation feature importance**: This method involves randomly permuting the values of a single feature while keeping other features constant and measuring how much the model's performance decreases. A significant drop in performance indicates an important feature.
- **Recursive feature elimination** (**RFE**): RFE is an iterative method that starts with all features and gradually removes the least important ones based on a model's performance. The order in which features are removed indicates their importance.
- **SHapley Additive exPlanations** (**SHAP**): SHAP values provide a unified measure of feature importance by considering all possible feature combinations. They can be applied to various models, including deep learning models.

The choice of feature importance calculation method depends on the machine learning algorithm used, the dataset, and the problem at hand. Different algorithms may provide different rankings of feature importance, so it's essential to consider multiple methods and use domain knowledge to interpret the results effectively. Feature importance scores help identify relevant features, reduce dimensionality, and improve model interpretability.

Discovering gradient-based attribution methods

Gradient-based attribution methods, also known as gradient-based attribution techniques, are approaches that are used to understand and attribute the contributions of individual features or input elements to the output of a machine learning model. These methods rely on gradients, which represent the sensitivity of the model's output to changes in input features. Here are some gradient-based attribution methods that are commonly used in machine learning:

- **Gradient saliency**: Saliency maps emphasize the most pertinent areas in an input image that influence a model's prediction. These maps are created by calculating the gradient of the model's output concerning the input image pixels. Regions with high gradients correspond to areas of significant relevance.
- **Integrated gradients**: Integrated gradients assigns attribution scores to each input feature by computing the cumulative integral of the gradient concerning the input along a path from a reference input (usually all zeros) to the actual input. This method provides a more comprehensive understanding of feature importance.
- **Guided backpropagation**: Guided backpropagation is a modified backpropagation algorithm that retains only positive gradients during backpropagation. This helps highlight the positive contributions of input features to predictions and suppresses negative contributions.
- **SmoothGrad**: SmoothGrad reduces noise in saliency maps by averaging gradients across multiple perturbed versions of the input and then visualizing the smoothed gradient values.
- **Layer-wise relevance propagation** (**LRP**): LRP is an attribution method that assigns relevance scores to each neuron in the network's hidden layers and propagates them backward to the input features. It provides fine-grained feature relevance information.
- **Deconvolutional networks** (**DeconvNets**): DeconvNets are designed to reverse the effects of convolutional layers in a neural network. They help visualize feature maps at different layers of the network to understand what each layer learns.
- **Gradient Class Activation Mapping** (**GradientCAM**): GradientCAM combines gradient information with class activation mapping techniques to highlight regions in an input image that are important for a specific class prediction.

Gradient-based attribution methods are valuable for model interpretability and debugging. They help identify which features or parts of the input are influential in driving the model's decisions. Choosing the right attribution method depends on the model architecture, dataset, and specific goals of interpretation. These methods provide insights into model behavior and can help build trust in AI systems.

Summary

In this chapter, we understood the basic concepts surrounding computer vision and how to implement a model for object recognition using MATLAB. We started by introducing image processing and computer vision. We learned how tools are available to process images and how computer vision is used for object recognition, motion detection, and pattern recognition. Then, we explored MATLAB tools for computer vision, and how the capabilities and functions provided by MATLAB create a robust environment for the development and prototyping of computer vision applications. Whether your focus is on tasks such as image analysis, object detection, 3D reconstruction, or any related application, MATLAB offers the necessary tools and features to support your work effectively.

After that, we learned how to build a MATLAB model for object recognition by using a CNN and the MNIST dataset. We understood how to import image data into a MATLAB workspace and how to use images to train a CNN. Then, we learned how to use pretrained deep learning models in MATLAB to improve the performance of the object recognition model. Finally, we introduced some tools for interpreting and explaining deep learning models.

In the next chapter, we will delve into fundamental concepts surrounding sequential data and explore the process of constructing a model to capture patterns within time series or any general sequence. We will learn the basic concepts of time series data, how to extract statistics from sequential data, and how to implement a model to predict the stock market data. Finally, we will understand oversampling, undersampling, and cost-sensitive learning.

9

Time Series Analysis and Forecasting with MATLAB

Time series data constitutes a sequence of measurements gathered over a certain period. These measurements, which are tied to a specific variable, occur at regular intervals. An essential characteristic of time series data lies in the significance of its order; the arrangement of observations on a timeline conveys meaningful patterns. Altering this order can completely reshape the data's meaning. Sequential data is a broader concept that encompasses any data presented in a sequential manner, which includes time series data. In this chapter, we will delve into the fundamental concepts surrounding sequential data, elucidating how to construct models that capture patterns within time series or any sequential data.

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

- Exploring the basic concepts of time series data
- Extracting statistics from sequential data
- Implementing a model to predict stock market data
- Dealing with imbalanced datasets in MATLAB

Technical requirements

In this chapter, we will introduce basic machine learning concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also require a working knowledge of MATLAB.

To work with the MATLAB code in this chapter, you will need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `Nile.csv`
- `TSLA.csv`
- `TeslaStockForecasting.m`

Exploring the basic concepts of time series data

A time series represents a chronological sequence of recorded observations, such as monthly revenue, daily stock prices, weekly interest rates, annual profits, and more. The primary objective of time series analysis is to examine the historical progression of a phenomenon over time to anticipate its future trajectory. This predictive insight is derived from the assumption that recurring patterns observed in the past will continue to manifest in the future. We will explore the concepts of predictive forecasting and various forecasting methodologies, providing a detailed description of both.

Understanding predictive forecasting

Predicting the trajectory of variables is immensely significant when it comes to formulating plans and policies for any endeavor. For instance, when planning a company's production strategy, it isn't sufficient to merely understand whether the demand for products or services is on the rise or declining. It's imperative to forecast the future trends in product demand, pricing dynamics, and the costs of raw materials. Collectively, these elements wield substantial influence over production activities.

Forecasts hold a pivotal position at the core of the entire decision-making procedure. When forecasts are imprecise or insufficient, there exists a substantial risk of rendering the conclusions drawn from intricate decision models ineffective. The phrase *forecasting process* encompasses a range of intricate activities, whether overt or covert, that ultimately culminate in the creation of a forecast.

The terms *forecast* and *prediction* are frequently used interchangeably, but it's beneficial to differentiate their meanings. Forecasting involves associating probabilities with future events or defining confidence intervals to estimate the range of values likely to occur in the future. On the contrary, prediction entails identifying the exact value a measurable quantity will take on in the future. Consequently, it's straightforward to connect forecasts with the predictions made by employing traditional inferential statistical tools to derive corresponding confidence intervals. The primary goal of all forecasting models is to determine an estimation of the anticipated value, coupled with an estimation of the potential error the forecasting model may generate.

The forecast horizon is a crucial factor that distinguishes the forecasting process significantly. Forecasts can pertain to the immediate future, spanning up to 12 months, and serving as a foundation for operational decision-making. This might involve predicting product demand for the next 2 months. Alternatively, forecasts may extend into the medium term, ranging from 12 to 24 months, supporting

decisions related to production planning. In the third scenario, forecasts are aimed at a more distant future, beyond the 24-month mark. In this case, they are crafted to underpin managerial decisions concerning company development plans. Across these three scenarios, each characterized by a short, medium, or long prediction horizon, the objectives of decision-makers seeking to utilize these forecasts differ significantly. Likewise, the level of precision and detail required for these respective forecasts also varies.

Introducing forecasting methodologies

Forecasting methodologies refer to the various techniques and approaches that are used to predict future values, trends, or events based on historical data and patterns. These methodologies are essential in a wide range of fields, including economics, finance, business, and meteorology, among others. Forecasting methodologies vary primarily depending on the characteristics and intended purposes of the decisions they support. Factors such as the time horizon's duration, the quality and consistency of historical data, and the specific attributes of the product being forecasted, such as its stage in the life cycle, play significant roles in determining the appropriate forecasting method.

In essence, forecasting methods can be categorized into two major groups: **quantitative methods** and **qualitative methods**.

Explaining quantitative forecasting methods

Quantitative methods are employed when there is sufficient quantitative data available, enabling predictions of future outcomes based on historical data. Typically, these methods find application in making short - to intermediate-term decisions. Here are some examples of quantitative prediction methods:

- **Time series analysis**: In this method, the phenomenon to be predicted is treated as an opaque entity, and the focus is not on identifying influencing factors. The objective of this approach is to recognize past patterns in the phenomenon's evolution and extend those patterns into the future to make predictions. In simpler terms, predictions are based on the historical behavior of the phenomenon over time rather than being linked to explanatory variables. For example, this approach is often used in analyzing sales trends, GDP trends, and similar data.
- **Regression analysis**: Regression models quantify the correlation between one or more predictor variables and a response variable. These methods operate under the assumption that the variable to be forecasted can be associated with one or more independent or explanatory variables. This method is suitable for predicting numerical outcomes and is commonly used in economics, finance, and social sciences. For instance, the demand for consumer goods in a household is believed to be influenced by factors such as income, the ages of household members, and other related variables.
- **Exponential smoothing**: Exponential smoothing techniques allocate exponentially decreasing weights to previous observations, prioritizing the significance of more recent data points. This helps in smoothing out noise and identifying trends in time series data.

- **Machine learning**: Machine learning algorithms, including decision trees, random forests, and neural networks, find application in forecasting tasks, especially when handling extensive datasets or intricate relationships. They can capture non-linear patterns and are increasingly employed in various forecasting applications.

These methods are applicable under the following conditions:

- Sufficient historical data is accessible regarding the phenomenon's previous patterns
- It is possible to quantify this historical information
- It can be reasonably assumed that the attributes characterizing past trends will persist into the future, allowing for accurate forecasting

In essence, quantitative methods are employed when there is an ample supply of quantifiable historical data.

Describing qualitative forecasting methodologies

Qualitative methods primarily rely on judgments, making them dependent on the viewpoints and assessments of consumers and experts. These methods come into play when quantitative data is scarce, yet ample qualitative information is available. Here are a few instances illustrating the application of qualitative methods:

- **Expert judgment**: Qualitative forecasting often relies on the expertise and opinions of subject matter experts. The Delphi method and expert panels are examples of techniques that gather input from knowledgeable individuals.
- **Market research**: Qualitative research methods, such as surveys, focus groups, and customer interviews, are used to gather information about consumer preferences, behaviors, and market trends. This data can be valuable for predicting future market conditions.
- **Scenario analysis**: Qualitative forecasting can involve the creation of multiple scenarios or narratives about possible future events. Decision-makers then consider these scenarios to make informed decisions.
- **Historical analogy**: This method involves making predictions based on similarities with past events or situations. It can be useful when dealing with unique or novel situations.
- **Expert systems**: Expert systems and AI-driven tools can assist in qualitative forecasting by aggregating and interpreting data from various sources to provide insights and predictions.

The choice between quantitative and qualitative methods for forecasting depends on various factors, including the nature of the data, the availability of historical information, the specific forecasting goals, and the level of subjectivity involved. In many cases, a combination of both quantitative and qualitative methods is used to improve the accuracy and robustness of forecasts, especially in complex decision-making contexts.

Time series analysis

A time series represents a collection of observations related to a phenomenon, recorded at consecutive time points or intervals. These intervals are typically, though not always, evenly spaced or of uniform duration. Instances of time series data include diverse elements such as trends in commodity prices, stock market indices, the BTP/BUND spread, and unemployment rates.

Unlike classical statistics, which frequently assumes that independent observations stem from a single random variable, time series analysis presupposes n observations derived from a multitude of dependent random variables. Consequently, the analysis of a time series entails a procedure aimed at deciphering the underlying generating process behind the observed data, as opposed to treating each observation as independent and identically distributed.

Based on the data collected, time series data can be divided into two main types:

- **Continuous time series**: Data is recorded continuously over a specific period without any gaps or interruptions. This type of time series is common in applications such as sensor data, where measurements are taken at very short intervals, often in real time. Examples include temperature readings recorded every second, stock prices updated every minute, or heart rate monitoring over continuous intervals.
- **Discrete time series**: Data is recorded at specific, discrete time intervals, which may not necessarily be evenly spaced. These intervals can be daily, weekly, monthly, annually, or any other predetermined period. Discrete time series data is often used in economic, financial, and social sciences to analyze trends and patterns over time. Examples include monthly sales figures, annual GDP growth rates, or weekly website traffic statistics.

The choice between continuous and discrete time series depends on the nature of the data being collected and the specific objectives of the analysis. Each type of time series has its own set of statistical techniques and tools that are used for modeling, forecasting, and extracting meaningful insights.

Time series data can also be categorized into two main types based on the underlying nature of the data-generating process:

- **Deterministic time series**: This type is characterized by a clear and predictable pattern or relationship between data points. The data values in a deterministic time series follow a specific mathematical function or rule. Deterministic patterns can be expressed as mathematical equations or formulas and may include periodic or seasonal variations, trends, or specific mathematical relationships. Deterministic time series often exhibit little or no randomness, making them highly predictable. Examples of this type of time series are a sine wave representing daily temperature variations over a year, a linear trend in stock prices, where prices consistently increase or decrease over time, and monthly sales figures influenced by a known seasonal pattern, such as holiday sales.

- **Stochastic time series**: This is characterized by randomness, and the data points are not governed by a specific deterministic rule. Instead, they are influenced by random or probabilistic factors. Stochastic time series exhibit inherent randomness, making it challenging to predict future values with certainty. They often involve elements of uncertainty, noise, and randomness that cannot be fully explained by a deterministic model. Statistical techniques, such as **Autoregressive Integrated Moving Average** (**ARIMA**) models or state-space models, are commonly used to analyze and forecast stochastic time series. Examples of this type of time series are stock prices, which are influenced by a multitude of unpredictable factors and exhibit daily fluctuations that cannot be precisely predicted, daily weather conditions, where variables such as rainfall or temperature can vary randomly, and daily website traffic, which is influenced by user behavior and external factors, resulting in random fluctuations.

In practice, many real-world time series data contains a combination of deterministic and stochastic components. Analyzing and modeling time series data effectively often requires identifying and separating these components to gain insights and make accurate forecasts or predictions. Deterministic models are useful when the underlying patterns are well defined and stable, while stochastic models are employed when randomness plays a significant role in the data.

Indeed, when dealing with time series data that exhibits both deterministic and stochastic components, it is common to represent the series as the sum of these two contributions. This approach allows for a more comprehensive understanding of the data and facilitates modeling and analysis. Mathematically, it can be expressed as follows:

$$Y_t = f(t) + w(t)$$

Here, we have the following:

- Y_t: This represents the observed time series data at time t
- $f(t)$: This represents the deterministic component or contribution at time t
- $w(t)$: This represents the stochastic (random) component or contribution at time t

By decomposing the time series into its deterministic and stochastic parts, analysts and researchers can do the following:

- **Analyze trends**: The deterministic component often includes trends, seasonality, and other structured patterns that can be analyzed separately to understand the underlying dynamics.
- **Model stochastic behavior**: The stochastic component captures the random fluctuations and noise in the data. Models such as **ARIMA** or **Generalized Autoregressive Conditional Heteroskedasticity** (**GARCH**) can be applied to this component for forecasting or risk assessment.

This decomposition approach enhances the ability to make forecasts, identify anomalies, and gain insights into the factors influencing the time series data.

Using time denoted as $t = 1 \ldots T$, we represent the sequence, y_t. Time serves as the critical parameter governing the sequence of events that must not be overlooked. Consequently, understanding the temporal dimension's position of observation becomes essential. Typically, this information is depicted as the pair of values (*t*, *yt*) on a Cartesian graph, creating a continuous line graph that conveys the impression of continuous detection of the phenomenon. This graphical representation is commonly known as a time series plot (*Figure 9.1*).

To understand how a series plot can be drawn, we can use the `Nile.csv` file in this book's GitHub repository. This file provides data on the annual flow of the Nile river at Aswan from 1871 to 1970, measured in units of 10^8 cubic meters. We can import this dataset into MATLAB using the following code:

```
Nile = readtable('Nile.csv');
```

To draw a time series plot, we can use the following command:

```
plot(Nile.time,Nile.Nile)
```

We used the dot format, where `time` and `Nile` are the names of the two features selected. The following time series plot will be drawn:

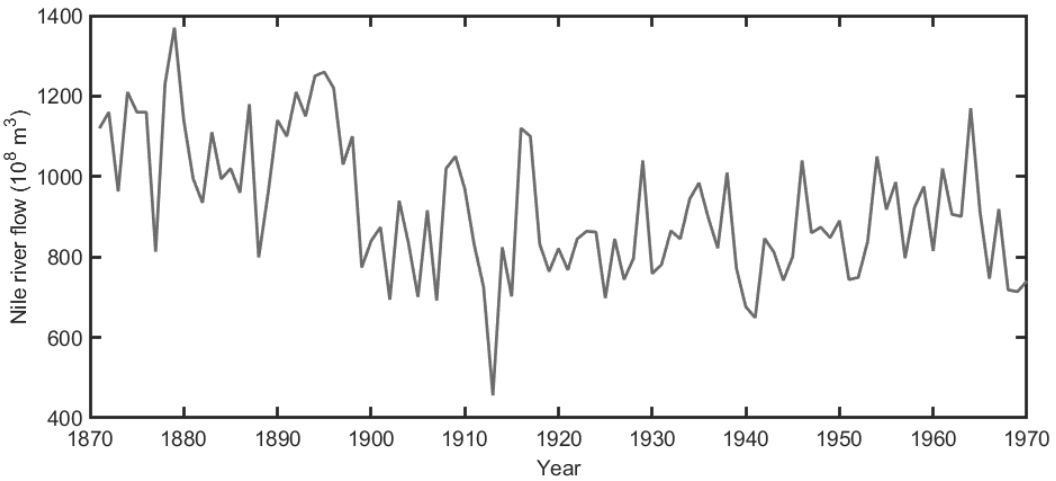


Figure 9.1 – Measurements of the annual flow of the Nile river at Aswan

As shown in *Figure 9.1*, there appears to be a significant change point in the data around 1898, suggesting a notable shift or change in the river's flow characteristics during that period.

A time series plot provides an immediate insight into trends, recurring patterns, and other systematic behaviors that evolve. In the previous graph, we can see annual data exhibiting a consistent declining trend spanning a considerable time frame. Notably, there is a recurring zigzag pattern due to the data being recorded monthly, which is indicative of seasonality. It's worth noting that the high peaks occur consistently during months when rainfall is anticipated.

Univariate analysis of time series aims to decipher the dynamic processes that underlie the series and predict future occurrences of the phenomenon. In this analysis, we focus exclusively on the data pairs (t, yt), where $t = 1, \ldots, T$. The crucial concept here is that both past and present data contain valuable information for forecasting the future development of the phenomenon.

However, it's essential to recognize that univariate analysis might be overly restrictive in some cases. Often, we possess information related to phenomena associated with the one under consideration, which should be suitably integrated to enhance the model's predictive capabilities. Nevertheless, univariate analysis serves as a valuable baseline, allowing us to validate more sophisticated modeling approaches.

In a time series plot, four distinct patterns can be identified concerning the progression of time:

- **Horizontal pattern**: In this scenario, the series fluctuates around a constant value, which is typically the series average. Such a series is referred to as being stationary on average. This pattern is commonly encountered in quality control, where processes are consistently maintained around an average value.
- **Seasonal pattern**: A seasonal pattern emerges when the series is influenced by recurring seasonal factors, such as monthly, semi-annual, or quarterly variations. Products such as ice cream, soft drinks, and electricity consumption often exhibit seasonal patterns. These series, which are influenced by seasonality, are also known as periodic series since the seasonal cycle repeats at fixed intervals. In the case of annual data, seasonality may not be evident.
- **Cyclic pattern**: The presence of a cyclic pattern is characterized by irregular increases and decreases in the series that do not follow a fixed period. This distinguishes cyclic fluctuations from seasonal ones. Furthermore, cyclic oscillations typically exhibit larger amplitudes compared to seasonal variations. In economic series, cyclic patterns are driven by economic expansions and contractions resulting from speculative phenomena.
- **Trend**: Trends are recognized by sustained, long-term increases or decreases in the series. For instance, the global population series exemplifies an increasing trend, while the monthly beer sales series may not display any discernible trend and instead possesses a stable horizontal background pattern.

Many time series exhibit a combination of these patterns, and it is precisely this kind of complexity that adds a high level of interest to the forecasting task. Forecasting methods must possess the capability to discern and replicate the various components of the series. This involves the assumption that past patterns will persist and continue to manifest their characteristic features in the future.

The classical approach to time series analysis involves dissecting the deterministic portion of the series into a set of signal components, which convey the structural information of the series while filtering out the negligible noise. In practical terms, our objective is to identify some of the previously mentioned patterns within the time series trend.

In time series analysis, three fundamental components are used to model and understand the underlying patterns in the data: trend, seasonal, and residual components. These components help decompose a time series into its constituent parts, making it easier to analyze and forecast the data accurately:

- **Trend component**: It represents the long-term, systematic, and often smooth movement or direction in the time series. It reflects the underlying, sustained changes or growth (increasing trend) or declines (decreasing trend) in the data over an extended period. The trend component aims to capture the overall behavior of the time series, making it essential for understanding its fundamental trajectory.
- **Seasonal component**: It accounts for the recurring patterns or fluctuations in the time series that follow a fixed and known period. These patterns repeat at regular intervals, such as daily, monthly, quarterly, or annually. Seasonality is often linked to external factors such as weather, holidays, or business cycles and tends to recur predictably over time.
- **Residual component**: This component, also known as the error or noise, represents the random fluctuations or unexplained variability in the time series data. It includes all the information not accounted for by the trend and seasonal components. Analyzing the residual component helps identify irregularities, outliers, or unexpected events in the data.

Mathematically, a time series (Y_t) can be expressed as the sum of these three components. Decomposing a time series into these components is a fundamental step in time series analysis. Once separated, analysts can model each component individually, apply appropriate statistical techniques, and make more accurate forecasts. This decomposition helps in understanding the underlying dynamics of the data and can provide valuable insights for decision-making and forecasting.

Let's see the different approaches to time series modeling and how it is possible to extract knowledge from these types of data.

Extracting statistics from sequential data

Time series data represents a sequence of measurements gathered over a certain period. These measurements are linked to a specific variable and are obtained at regular intervals. An essential characteristic of time series data is its inherent order, where the arrangement of observations along a timeline conveys significant information. Altering the sequence can completely change the data's meaning. Sequential data, on a broader scale, encompasses any data presented sequentially, including time series data.

Our primary goal is to develop models that capture the underlying patterns within time series data or any sequential data. These models are instrumental in describing essential aspects of the time series patterns. They enable us to explore how past data influences the future, examine correlations between datasets, make future predictions, or control variables based on specific metrics. To visually represent time series data, we commonly employ line charts or bar graphs. Time series data analysis finds extensive use in various domains, including finance, signal processing, weather forecasting, trajectory prediction, earthquake prediction, and any field dealing with temporal data.

In the realm of time series and sequential data analysis, the models we construct must consider data ordering and extract relationships among adjacent data points. Now, let's explore some MATLAB code examples for analyzing time series and sequential data.

Converting a dataset into a time series format in MATLAB

A time series represents a sequence of observations of a phenomenon collected at consecutive moments or time intervals, typically evenly spaced or of consistent duration, although not necessarily so. It's crucial to recognize that time plays a fundamental role in the analysis of a time series. To begin, we need to develop proficiency in handling data that portrays a prolonged observation of a specific phenomenon. Our initial step will involve learning how to transform a sequence of observations into time series data and then visualizing it.

In MATLAB, to adequately handle time series data, we can use the `timetable` object. A timetable is a specialized form of a table that's designed for handling time series data. Similar to regular tables, timetables store data variables organized by columns, allowing for various data types and sizes, so long as they share the same number of rows. In addition to this flexibility, timetables offer specific functions tailored for time-related operations, enabling alignment, combination, and calculations involving timestamped data across one or more timetables.

The row times in a timetable consist of datetime or duration values that serve as labels for individual rows. You can access and manipulate a timetable by row time and variable. To access specific elements within a timetable, you can use round parentheses, *()*, to retrieve a subtable or curly braces, *{}*, to extract their contents. Furthermore, you can reference variables and the row times vector using their respective names. To identify and annotate events within a timetable, you can associate an event table with it. Event tables contain a record of event times, corresponding event labels, and additional event-related details.

To begin with, we will see how it is possible to create a timetable from such data. Suppose we have stored data about the flow of a river, measured at different times, in a timetable. In addition to archiving, schedules provide functions to synchronize data with specified times. In addition, we can note the time to add more information about the measurement conditions and other variables to be measured. Therefore, we can associate a time with the variables present in the workspace. The values present in the variable containing the times become the times of the object lines. All the other input arguments become time-dependent variables. Let's learn how to format timetable data:

1. To begin, we must create the vector containing the variables, starting with the time vector:

```
TimeData = datetime({'01.01.1871';'01.01.1872';'01.01.1873';
'01.01.1874';'01.01.1875'});
TempData = [22.1;21.1;23.2;21.9;22.5];
RiverFlow = [1120;1160;963;1210;1160];
```

   First, we have the time of the measurements, then the value of the temperatures measured in this place at this time, and finally the values of the river flow measured in units of 10^8 cubic meters. To set the time format, we used the `datetime()` function: a datetime array contains information about the year, month, day, hour, minute, and second components associated with each recorded point in time, adhering to the proleptic ISO calendar system. Additionally, datetime arrays offer versatile formats for displaying output and parsing input text, support for storing fractional seconds with precision down to nanoseconds, and properties to handle considerations such as time zones, daylight saving time, and leap seconds.

2. The data that's used is corrected and added to the time vector in a format compatible with the type of data. Now, we can use the `timetable()` function to create the object:

```
NileRiverFlow = timetable(TimeData, TempData, RiverFlow);
```

3. A new timetable object was created in the MATLAB workspace. We can check the object's type as follows:

```
class(NileRiverFlow)
ans =
    'timetable'
```

4. To understand how the `timetable()` function handles the data, we will print the object that was created, as follows:

```
NileRiverFlow
NileRiverFlow =
  5×2 timetable
     TimeData       TempData    RiverFlow
    ____________    ________    _________
    01-Jan-1871       22.1        1120
    01-Jan-1872       21.1        1160
    01-Jan-1873       23.2         963
    01-Jan-1874       21.9        1210
    01-Jan-1875       22.5        1160
```

5. The timeline can also be constructed based on specific values of some parameters. For example, we can use the `RiverFlow` variable already available in the MATLAB workspace to associate each of its values with a specific time value defined in a specific timeline. For example, we can define a time step as follows:

```
NileRiverFlow2 = timetable(RiverFlow,'TimeStep',years(1),
'StartTime',years(1870));
```

We set `TimeStep` to one year and `StartTime` to 1870. By doing this, one value of `RiverFlow` is associated with a new year starting from 1870 with an increment of one year.

Understanding time series slicing

Slicing and dicing are two expressions that are used in the context of datasets, signifying the process of partitioning a substantial dataset into smaller segments or examining it from various perspectives to gain deeper insights. These terms draw an analogy from culinary terminology, describing two fundamental knife techniques that chefs must proficiently execute. Slicing entails cutting, whereas dicing involves cutting food into finely uniform sections, often performed consecutively. In data analysis, the concept of slice and dice typically encompasses the systematic breakdown of a large dataset into more manageable portions to extract additional information.

In MATLAB, a timetable is a specialized form of table that links a specific time with each row. You can efficiently extract time-related subsets of its data through various methods, such as identifying times within a specified range or matching recurring time intervals.

After learning how to create a timetable object from variables in the workspace, we can learn how to import and manipulate an external file into MATLAB as a timetable object:

1. We will use the `.csv` file we used in the *Exploring the basic concepts of time series data* section. I'm referring to the `Nile.csv` file; we can import this file using the following command:

```
NileRiverFlowData = readtimetable('Nile.csv');
```

Now, let's learn how to access this specific type of data. You can employ dot notation to retrieve the row times of a timetable. Furthermore, you can access specific variables by using dot notation individually or access all the data within a timetable using its name, as we just did.

2. To display the first two lines of the time series object, we can use the following command:

```
NileRiverFlowData(1:2,:)
```

The following data is printed:

```
NileRiverFlowData(1:2,:)
ans =
  2×1 timetable
       time       Nile
```

```
    ____________    ____
    01.01.1871      1120
    01.01.1872      1160
```

3. To access only the data of one variable, we can use dot notation, as follows:

```
NileRiverFlowData.Nile(1:2)
```

4. Retrieve the entirety of the timetable data as a matrix using the `NileRiverFlowData.Variables` syntax. This syntax relies on the second dimension name of the timetable and is functionally identical to accessing all contents through curly brace indexing:

```
NileData = NileRiverFlowData{:,:};
```

   However, it's important to note that the resulting matrix does not incorporate row times as the row times vector is considered metadata within the timetable and not treated as a variable. In cases where the timetable data cannot be effectively concatenated into a matrix, an error message will be generated.

5. To locate data within a particular range, you can utilize the `timerange` function, which establishes time-based subscripts for indexing. For example, you can set up a range that commences on January 1, 1880, and concludes on January 1, 1920. It's important to note that by default, the `timerange` function defines a half-open interval that includes the left endpoint but excludes the right endpoint:

```
TimeRange = timerange('01.01.1880','01.01.1920');
```

6. Now, we can slice a portion of the dataset defined from this range, as follows:

```
DataTimeRange = NileRiverFlowData(TimeRange,:);
```

The new dataset contains data starting from January 1, 1880, and ending on January 1, 1919.

Resampling time series data in MATLAB

Let's see how it's possible to resample time series data in MATLAB. This task involves changing the frequency or time intervals of the data while preserving the essential information. This can be useful for various purposes, such as aggregating data to a coarser time scale, interpolating data to a finer time scale, or aligning data from different sources with a common time grid. MATLAB provides several functions and methods for resampling time series data. Here's a basic example of how to do it.

We start by using the `retime()` function, as follows:

```
WeeklyData = retime(NileRiverFlowData,'weekly','spline');
```

This function resamples or consolidates data within a timetable while addressing issues related to duplicate or irregular time entries. The `retime()` function generates a timetable by incorporating variables from the initial data and enforces a consistent time step for the row times. This function resamples or consolidates data within the initial variables based on the specified method. `retime()` can be applied to interpolate data values from initial values at different timestamps, aggregate data into time intervals, eliminate rows with duplicate timestamps in the timetable, and transform an irregular timetable into a regular one by imposing regularly spaced row times, as defined by the new *time step*.

In this example, we resampled the dataset by adding the weekly data starting from the monthly, but there are different ways to resample data. We performed data value interpolation based on the adjacent rows. It is essential that the input timetable has sorted and distinct row times. There are four different types of interpolation available:

- `linear`: Linear interpolation
- `spline`: Piecewise cubic spline interpolation
- `pchip`: Shape-preserving piecewise cubic interpolation
- `makima`: Modified Akima cubic Hermite interpolation

There are the possibilities to aggregate the time series data using the same function (`retime()`). The `retime()` function offers aggregation options, including the mean. For example, we can compute the mean values monthly for the given data:

```
MonthlyData = retime(NileRiverFlowData,'monthly','mean');
```

In this way, we aggregated the previously resampled weekly data using the average as the aggregation criterion.

Moving average

A moving average is a commonly used method for analyzing time series data. It involves calculating the average of a set of data points within a specific window or time interval as it moves through the time series. This helps to smooth out fluctuations and highlight underlying trends in the data. There are different types of moving averages, such as the **simple moving average** (**SMA**) and the **exponential moving average** (**EMA**), each with its characteristics and use cases.

SMA is a basic and widely used method for analyzing time series data. It is a statistical calculation that helps smooth out fluctuations in data to identify trends or patterns. Here's how you calculate SMA:

1. Choose a specific period or window.
2. For each data point, take the average of the data points within that time window. This involves adding up the values over the chosen period and dividing by the number of data points in that window.

3. Move the window one time step forward and calculate the average again. Repeat this process for each data point.

The result is a series of averages that represent the trend in the data over the chosen period. SMAs are commonly used for various purposes, including identifying trends, smoothing out noise in time series data, and making forecasts. For example, if you are interested in calculating a 10-day SMA for daily stock prices, you would average the closing prices of the last 10 days to get a moving average value for each day. This moving average can help you identify the general direction of the stock's price trend over those 10 days.

In MATLAB, you can calculate an SMA for a time series using the `movmean()` function, which is available in the Statistics and Machine Learning Toolbox. Let's learn how to use the `movmean()` function with a practical example:

1. We'll start by defining the window size for the SMA:

   ```
   windowSize = 5;
   ```

   We defined a five-year window to calculate the moving average.

2. Now, we can calculate the moving average:

   ```
   SMA = movmean(NileRiverFlowData.Nile,windowSize);
   ```

   The `movmean()` function generates an array of local mean values for k-point neighborhoods. Each mean is computed by sliding a window with a length of *k* across adjacent elements of the array, *A*. If *k* is an odd number, the window centers around the current position's element. For even values of *k*, the window centers around the current and preceding elements. The window size adjusts automatically at the array's endpoints when there are insufficient elements to fill it. In such cases, the average is calculated based on the elements that fit within the window. The resulting array, denoted as *M*, maintains the same dimensions as array *A*. When *A* is a one-dimensional vector, `movmean()` processes it along the vector's length. For multidimensional arrays, `movmean()` operates along the first dimension of *A* without a size of 1.

The EMA is a variant of the moving average, placing greater emphasis on recent data points. This characteristic renders it more responsive to short-term price fluctuations when compared to the SMA. To calculate an EMA, follow these steps:

1. Choose a specific period, typically represented as *N*.
2. Calculate the multiplier, which is often referred to as the "smoothing factor" or "weighting multiplier." This is usually calculated as *2 / (N + 1)*.
3. Start with the EMA for the first data point, which is typically the SMA for the first *N* data points.

For the subsequent data points, calculate the EMA using the following formula:

$$EMA(current) = (Price(current) - EMA(previous)) * Multiplier + EMA(previous)$$

Here, we have the following:

- *EMA(current)* is the EMA for the current data point
- *Price(current)* is the price of the asset at the current time
- *EMA(previous)* is the EMA for the previous data point
- *Multiplier* is the smoothing factor calculated in *Step 2*

The EMA assigns more weight to recent data points, and as a result, it reacts more quickly to price changes than the SMA. It's commonly used in technical analysis to identify trends and potential reversal points in financial markets, such as stocks, currencies, and commodities. Traders and analysts often use different EMA periods to analyze short-term and long-term trends.

MATLAB, for instance, provides functions such as `movavg()` to calculate various types of moving averages, including the EMA. You can use this function to compute EMAs in MATLAB. Remember that the `movavg()` function requires the use of Financial Toolbox.

We start by defining the type of moving average, after which we define the window size and calculate the EMA:

```
type = 'exponential'
windowSize = 5;
EMA = movavg(NileRiverFlowData.Nile,type,windowSize)
```

The EMA also calculates an average over a specific time window, but it assigns more weight to recent data points. It gives exponentially decreasing weight to older data points. The EMA is more responsive to recent changes in the data. It reacts quickly to price changes, making it suitable for short-term trend analysis and capturing price reversals.

Exponential smoothing

Exponential smoothing is a time series forecasting method that is used to make predictions about future data points based on past observations. It is particularly useful for data with a trend or seasonal component. Exponential smoothing assigns exponentially decreasing weights to past data points, with the most recent observations receiving the highest weights. EMA and exponential smoothing are both techniques that are used in time series analysis to capture trends and patterns. However, they differ in their formulations and applications. EMA, a subtype of exponential smoothing, places higher emphasis on recent data points by assigning them greater weight in the calculation. The mathematical expression for the EMA explicitly includes a smoothing parameter (alpha), offering a straightforward approach to adjusting the impact of recent versus older observations. Conversely, exponential smoothing is a broader term that encompasses various smoothing techniques, with the EMA being a specific instance. While the EMA requires an initial value for the calculation, exponential smoothing, in general, demands an initial value that significantly influences the overall smoothing process.

There are several variations of exponential smoothing, including the following:

- **Simple exponential smoothing** (**SES**): This is used for univariate time series data without a trend or seasonality. It uses a single smoothing parameter (alpha) to assign weights to past observations. The forecast for the next period is a weighted average of the most recent observation and the previous forecast.
- **Holt's linear exponential smoothing**: This is used for data with a linear trend but no seasonality. It extends SES by introducing two smoothing parameters, alpha for the level and beta for the trend. It calculates the level and trend separately and uses them to make forecasts.
- **Holt-Winters exponential smoothing**: This is used for data with both a trend and seasonality. It extends Holt's linear exponential smoothing by adding a third smoothing parameter, gamma, for the seasonal component. It models and forecasts the level, trend, and seasonal components.

The basic idea behind exponential smoothing is to assign exponentially decreasing weights to past observations, giving more importance to recent data, and making the method adaptive to changes in the data pattern. Exponential smoothing is widely used in time series forecasting applications, such as sales forecasting, demand forecasting, and financial market predictions.

Double and triple smoothing are advanced techniques in time series analysis, building upon the foundation of exponential smoothing. Double smoothing incorporates an additional level of smoothing to address trends, introducing a trend-smoothing parameter alongside the data-smoothing parameter. This method enhances the model's ability to capture and forecast trends effectively. Triple smoothing, or the Holt-Winters method, goes a step further by incorporating seasonality components. It includes parameters for data smoothing, trend smoothing, and seasonality smoothing, making it particularly robust for time series data with consistent patterns. These techniques provide nuanced tools for capturing and forecasting complex trends and seasonal variations.

In MATLAB, you can implement various types of exponential smoothing using functions and libraries related to time series analysis and forecasting. These functions often provide tools to estimate the smoothing parameters and make forecasts based on your data. Let's take a closer look:

1. We start by defining the smoothing parameter (`alpha`):

```
alpha = 0.3;
```

2. We can adjust `alpha` based on our needs (`0 < alpha <= 1`). After that, we must initialize the forecast with the first data point:

```
forecast = NileRiverFlowData.Nile(1);
```

3. Finally, we must calculate the SES forecast:

```
for I = 2:length(NileRiverFlowData.Nile)
    forecast(i) = alpha * NileRiverFlowData.Nile (i) + (1--
alpha) * forecast(i-- 1);
```

```
end
```

Now, we can display the forecasts:

```
disp(forecast);
```

The `alpha` parameter controls the smoothing level, with lower values making the forecast more responsive to recent data and higher values making it smoother.

In this section, we analyzed some statistical methods for extracting information from a time series. Now, let's see a practical case in which we predict the performance of a company's shares using recurrent networks.

Implementing a model to predict the stock market

Predicting stock market movements is a complex and challenging endeavor. It involves analyzing various factors and data points to forecast the future direction of stock prices. Fundamental analysts examine a company's financial health, including its revenue, earnings, debt levels, and growth prospects. They also consider macroeconomic factors such as interest rates, inflation, and government policies that can impact the overall market. Technical analysts study historical price and volume data, looking for patterns and trends in stock charts. They use tools such as moving averages, support and resistance levels, and various technical indicators to make predictions. Identifying current market trends and understanding market cycles can provide insights into potential future movements. Bull markets, bear markets, and sideways markets can affect stock prices differently. Predictions are inherently uncertain, and risk management is crucial. Diversifying a portfolio, setting stop-loss orders, and using proper position sizing can help manage risks associated with predictions.

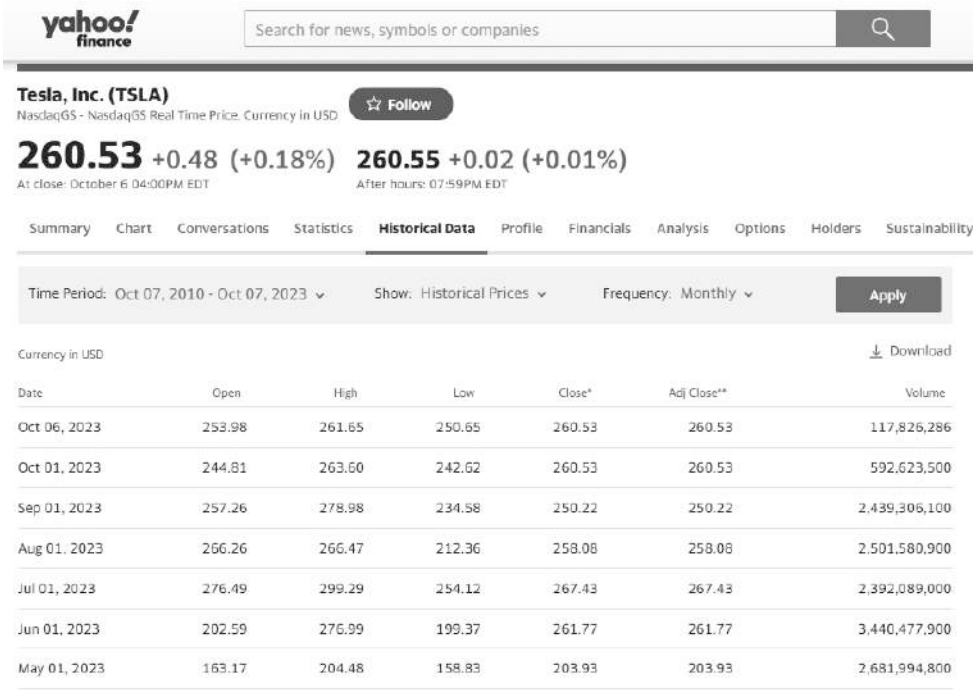
Some traders and investors use machine learning and **artificial intelligence** (**AI**) algorithms to analyze vast amounts of data and identify potential patterns or trends in the market. Stock market predictions can range from short-term (days or weeks) to long-term (years). The chosen will depend on the investment strategy and goals. It's important to remember that past stock market performance is not a guarantee of future results. Market conditions can change rapidly and unpredictably. Stock market predictions involve a combination of financial analysis, technical analysis, sentiment analysis, and consideration of various external factors. While some investors and traders may use predictions as part of their decision-making process, it's essential to approach them with caution and maintain a diversified portfolio to manage risk effectively.

In this section, we will delve into the application of the **Long short-term memory** (**LSTM**) model for forecasting the future stock price of a highly renowned corporation: Tesla Inc. This is an American **electric vehicle** (**EV**) and clean energy company founded in 2003 by Elon Musk, JB Straubel, Martin Eberhard, Marc Tarpenning, and Ian Wright. It is headquartered in Palo Alto, California.

To evaluate Tesla's stock price performance, we will utilize data encompassing the stock prices from November 6, 2010, to October 5, 2023, sourced from the NASDAQ GS stock quotes. It is also possible to adjust the timeframe if needed, beyond the default interval provided.

The data was acquired from the Yahoo! Finance website: `https://finance.yahoo.com/quote/TSLA`.

Figure 9.2 shows a screenshot of the Yahoo! Finance website:



Date	Open	High	Low	Close*	Adj Close**	Volume
Oct 06, 2023	253.98	261.65	250.65	260.53	260.53	117,826,286
Oct 01, 2023	244.81	263.60	242.62	260.53	260.53	592,623,500
Sep 01, 2023	257.26	278.98	234.58	250.22	250.22	2,439,306,100
Aug 01, 2023	266.26	266.47	212.36	258.08	258.08	2,501,580,900
Jul 01, 2023	276.49	299.29	254.12	267.43	267.43	2,392,089,000
Jun 01, 2023	202.59	276.99	199.37	261.77	261.77	3,440,477,900
May 01, 2023	163.17	204.48	158.83	203.93	203.93	2,681,994,800

Figure 9.2 – The Yahoo! Finance website

This CSV file includes the following attributes:

- **Date**: Date of quote
- **Open**: Open price
- **High**: High price
- **Low**: Low price
- **Close**: Close price adjusted for splits
- **Adj Close**: Close price adjusted for both dividends and splits
- **Volume**: Exchange volume

The information within the CSV file is labeled as `TSLA.csv`. To start, let's investigate the process of importing this data into the MATLAB workspace:

1. We will import the data in timetable format using the following command:

```
TeslaData = readtimetable('TSLA.csv');
```

   As always, we should be in the folder containing the `TSLA.csv` file or add this folder to the MATLAB path.

2. Before embarking on data prediction using the LSTM method, we will commence with an exploratory analysis to glean insights into the data distribution and extract preliminary knowledge. To glean preliminary insights from the imported dataset, we can utilize the `summary()` function:

```
summary(TeslaData)
```

   The following results are printed:

```
summary(TeslaData)
RowTimes:
    Date: 157×1 datetime
        Values:
            Min         2010-11-01
            Median      2017-05-01
            Max         2023-10-06
Variables:
    Open: 157×1 double
        Values:
            Min        1.4627
            Median      16.87
            Max         386.9
    High: 157×1 double
        Values:
            Min        1.6993
            Median       18.8
            Max         414.5
    Low: 157×1 double
        Values:
            Min        1.4033
            Median     15.798
            Max         326.2
    Close: 157×1 double
        Values:
            Min        1.5927
            Median     16.795
            Max        381.59
```

```
    AdjClose: 157×1 double
        Values:
            Min         1.5927
            Median      16.795
            Max         381.59
    Volume: 157×1 double
        Values:
            Min        1.1783e+08
            Median     1.8456e+09
            Max        7.0888e+09
```

A summary of the data was returned with some statistics on all the variables in the dataset. For example, we can read the number of records (`157`) and the min, median, and max values of all variables. Based on the preliminary analysis of the results, it is evident that Tesla's stock prices have undergone a significant transformation over the past 13 years. Specifically, the minimum value stands at $1.5927, while the maximum value reaches $381.59.

3. Upon reviewing the dataset's content, our next step involves conducting an initial visual exploratory analysis, wherein we will generate a graphical representation of the stock prices over the years:

```
plot(TeslaData.Date, TeslaData.Close)
xlabel('Time (years)')
ylabel('Stock price ($)')
```

The following chart will be drawn (*Figure 9.3*):

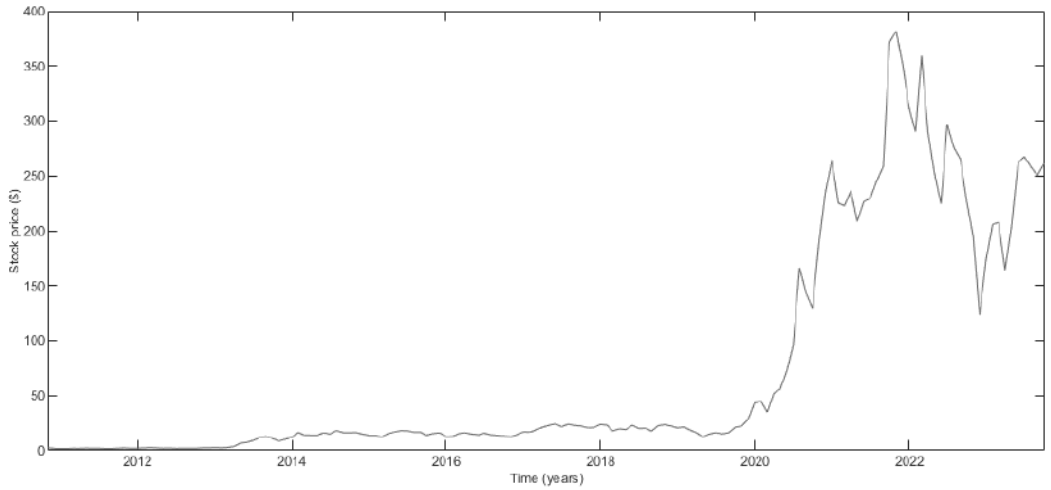


Figure 9.3 – Stock prices of Tesla Inc. from 2010 to 2023

Examining the preceding graph reveals a substantial increase in prices over time. Commencing in 2020, this upsurge follows an exponential trend. Let's delve deeper into understanding the fluctuations that the Tesla stock has documented over time.

4. Exploring the evolution of a phenomenon becomes intriguing not only by examining its time series graph but also by drawing comparisons between the intensity of the phenomenon at distinct time points. This involves calculating variations in intensity from one period to another. Additionally, analyzing the trend of changes in the phenomenon between contiguous periods can offer valuable insights.

   We represent a time series as *Y1*, ..., *Yt*, ..., *Yn*. As explained in the *Time series analysis* section, a time series reflects the chronological recording of experimental observations of a variable. This variable can encompass various elements, such as price trends, stock market indices, spreads, and unemployment rates. Essentially, it functions as a sequence of time-ordered data points from which we endeavor to extract insights for characterizing the observed phenomenon and predicting future values.

   To measure the variation between two distinct time points (referred to as t and $t + 1$), we can compute the following ratio:

$$\frac{Y_{t+1} - Y_t}{Y_t}$$

   This index, commonly known as "percentage change," represents a percentage ratio. It specifically denotes the percentage rate of variation in the phenomenon, Y, at time $t + 1$ compared to the preceding time, t. Utilizing the percentage change method provides a more accurate depiction of how data has evolved over a specified timeframe.

   This approach is applied not only in monitoring the prices of individual securities and major market indices but also in comparing the values of various currencies. When preparing balance sheets with comparative financial statements, it is typical to include specific asset prices at different time junctures, accompanied by the corresponding percentage changes during those periods.

5. To calculate percentage changes in MATLAB, we can use the following code:

```
PctCh=100*diff(TeslaData.Close(:,1))./TeslaData.
Close(1:end-1,1);
```

   The `diff()` function computes the differences between consecutive elements of an array along the first dimension where the size of that dimension is not equal to 1.

6. We can plot the results obtained as a bar plot to see how the differences are distributed on the time:

```
bar(TeslaData.Date(1:end-1,1),PctCh)
xlabel('Time (years)')
ylabel('Percentage changes ($)')
```

   Figure 9.4 shows the bar plot of percentage changes over the years:

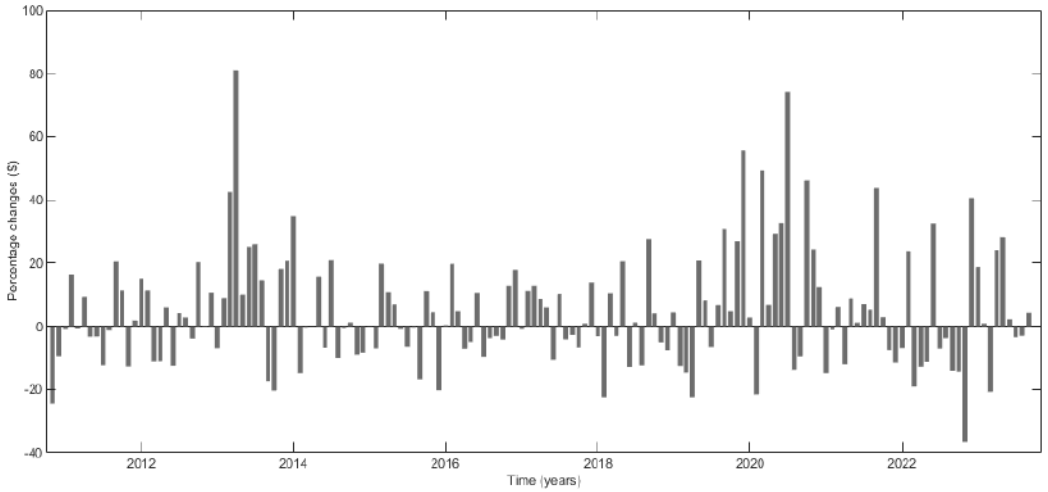


Figure 9.4 – Percentage changes over the years

Percentage changes in the stock market play a crucial role in providing valuable insights and serving various functions. For example, percentage changes allow investors and analysts to measure the performance of individual stocks, portfolios, or entire market indices over specific periods. This helps in assessing whether investments have grown or declined in value. Understanding percentage changes helps investors gauge the volatility and risk associated with a particular stock or asset. Stocks with larger percentage fluctuations are generally considered riskier investments.

7. After calculating the percentage changes, we can move on to the return concept; return refers to the profit or loss that an investor realizes from their investments in stocks. Returns are typically expressed as a percentage and represent the change in the value of an investment over a specific period.

8. Calculating the return from stock prices in MATLAB involves taking the percentage change in prices over a specific period. We have just loaded stock price data into MATLAB as a matrix. We checked that the data was organized with prices in chronological order. After that, we calculated the percentage change in stock prices from one period to the next. Now, we can calculate various types of returns, such as price return (capital gain) or total return (including income). For price return, simply sum the percentage changes, as follows:

```
PriceReturn = sum(PctCh);
```

The following result is obtained:

```
fprintf('Price Return: %.2f%%\n', PriceReturn);
Price Return: 698.80%
```

This is a great return from the Tesla stock market that's excellent for those who have held stocks for the last three years.

9. Our goal is to leverage the data within this dataset to forecast the Tesla stock price based on the information provided in the `.csv` file. To train the network, we require both input and output data. The input is defined by the data contained within the dataset. Consequently, we must generate our output. We will achieve this by assuming our objective is to forecast the Tesla stock price at time $t + 1$ based on the information available at time t. As a result, we will establish the following formula:

$$Input = TeslaData.Close(t)$$

$$Output = TeslaData.Close(t + 1)$$

This is a sequence-to-sequence regression problem, such as a deep learning approach where the goal is to predict a sequence of continuous values as an output, given a sequence of input data. It is commonly used for tasks such as time series forecasting, natural language processing, and any other problem where the output is a sequence of numerical values. In this context, the model learns to map an input sequence to an output sequence, and the length of the input and output sequences can vary.

10. A recurrent network possesses memory, and this memory is preserved by defining the time step. The time step determines how many steps into the past backpropagation are considered when computing gradients for weight updates during training. In this context, we establish *TimeStep = 1*. Let's prepare the data based on this assumption:

```
XData=TeslaData.Close(1:end-1);
YData=TeslaData.Close(2:end);
```

Within this code, `XData` represents the input variable, equivalent to data at time t, while `YData` signifies the output value for the subsequent period, namely data at time $t + 1$.

To address this type of problem, we can use an LSTM network. LSTM is a type of **recurrent neural network** (**RNN**) architecture that's designed to address the vanishing gradient problem that can occur when training traditional RNNs. LSTM models are especially effective for processing and making predictions on sequences of data. They are built around the concept of memory cells, which can store information over long sequences. These memory cells possess the capability to learn and remember patterns, rendering them well suited for tasks involving sequential data. Such tasks include time series analysis, natural language processing, and speech recognition.

The ideal lag value should be chosen based on the autocorrelation plot together with domain knowledge. To evaluate the performance of an LSTM model, long sequences (more lags) should be considered. Even an auto-regressive model would perform well for *lag=1*.

LSTM models have three fundamental gates that control the flow of information within the network:

- **Forget gate**: This is responsible for deciding which information from the previous cell state should be either forgotten or retained
- **Input gate**: This determines which new information should be stored in the cell state
- **Output gate**: This regulates the information to be output from the cell state

The cell state runs horizontally across the top of the LSTM and acts as a conveyor belt, allowing information to flow from one time step to another. The gates help regulate the flow of information into and out of the cell state.

11. We start by setting some parameters:

```
TrainLength = length(TeslaData.Close)-1;
numHiddenUnits = 200;
```

12. After that, we have to build the architecture of the deep network:

```
layers = [ ...
    sequenceInputLayer(TrainLength)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(50)
    dropoutLayer(0.5)
    fullyConnectedLayer(TrainLength)
    regressionLayer];
```

Six layers were used:

- `sequenceInputLayer`: A sequence input layer receives sequential data and performs data normalization before feeding it into a neural network.
- `lstmLayer`: An LSTM layer, as part of an RNN, specializes in capturing long-term dependencies within time series and sequential data. This layer facilitates additive interactions, contributing to enhanced gradient flow over extended sequences during the training process. Two parameters were passed: the number of hidden units and the output mode as a sequence. One argument was supplied: the quantity of hidden units, also referred to as the hidden size, denoted as a positive integer. This parameter determines the volume of information retained by the layer across time steps, encapsulated in the hidden state. The hidden state can encompass data from all preceding time steps, irrespective of the sequence length. Oversizing the number of hidden units may lead to overfitting the training data. Additionally, the output mode is specified with one of two values: `sequence`, which produces the entire sequence as output, and `last`, which generates output for the last time step of the sequence.
- `fullyConnectedLayer`: A fully connected layer multiplies the input with a weight matrix and subsequently incorporates a bias vector.

- `dropoutLayer`: This is used for regularization during training. It helps prevent overfitting by randomly setting a fraction of the input units to 0 during each forward and backward pass. This encourages the network to learn more robust and generalized features.
- `fullyConnectedLayer`: The last fully connected layer is often used for making predictions based on the features learned by the preceding layers in the network.
- `regressionLayer`: This is a layer that's used in deep learning for regression tasks. Regression tasks involve predicting continuous numerical values, as opposed to classification tasks, which involve predicting categories or classes. `regressionLayer` is specifically designed for regression problems, and it is commonly used in neural networks for tasks such as predicting stock prices, house prices, temperature forecasting, and various other numerical predictions. `regressionLayer` is responsible for calculating the loss between the predicted values generated by the neural network and the actual target values during training. It employs a loss function to gauge the dissimilarity between the predicted values and the ground truth values. Common loss functions employed in regression tasks encompass **mean squared error** (**MSE**), **mean absolute error** (**MAE**), and Huber loss, among others.

13. After that, we have to specify the training options:

```
maxEpochs = 600;
miniBatchSize = 20;
options = trainingOptions('adam', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'InitialLearnRate',0.01, ...
    'GradientThreshold',1, ...
    'Shuffle','never', ...
    'Plots','training-progress',...
    'Verbose',0);
```

As the optimization algorithm, we set the Adam optimizer, a popular optimization algorithm used in training deep neural networks. It is known for its ability to efficiently handle non-stationary objectives, high-dimensional parameter spaces, and noisy gradient information. Adam combines the benefits of both the RMSprop optimizer and momentum-based methods. We also set the number of training epochs: an epoch is one complete pass through the entire training dataset. Training may stop early based on validation performance to avoid overfitting. We set an appropriate learning rate for the optimizer. You may need to experiment with different learning rates to find the one that works best for your problem.

14. Now, we can train the network:

```
net = trainNetwork(XData,YData,layers,options);
```

The following results are printed (*Figure 9.5*):

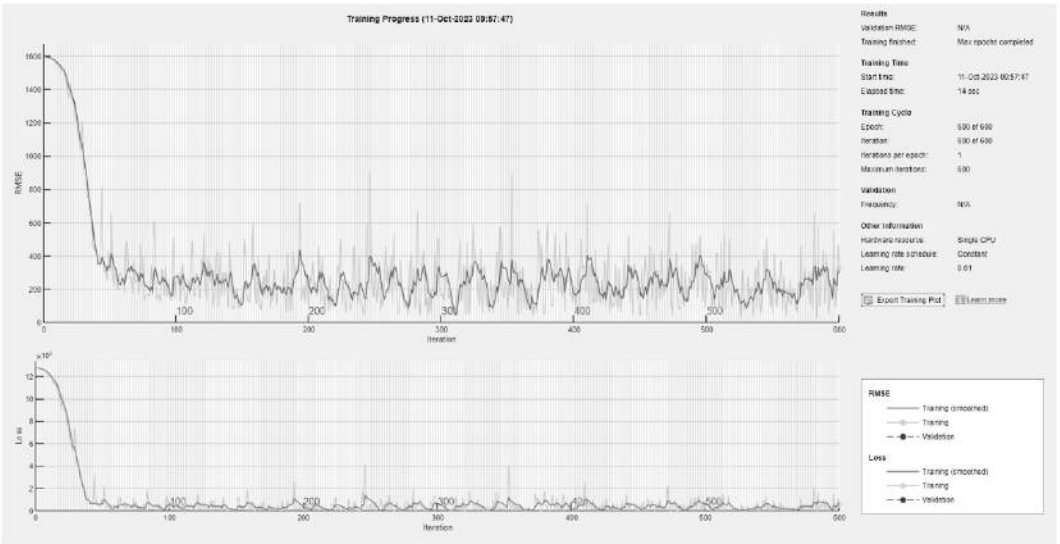


Figure 9.5 – LSTM training process

15. Now, we can use the network to predict the performance of stocks. Each value will be predicted starting from the previous one with a time step equal to 1:

```
YPred = predict(net,XData,'MiniBatchSize',1);
```

16. The `predict()` function forecasts outcomes by employing a well-trained deep learning neural network. To evaluate the performance of the prediction, we can use **root mean square error** (**RMSE**), which is calculated as follows:

```
RMSE = sqrt(mean((YData-YPred).^2));
```

RMSE is a widely utilized metric in statistics and machine learning for quantifying the average magnitude of errors or differences between predicted and actual values. It calculates the square root of the mean of the squared differences between predicted values and true values. RMSE serves as a means to evaluate the accuracy of a predictive model or the quality of predictions.

The following value is returned:

```
RMSE =
   13.9792
```

17. To appreciate the prediction capacity of the model based on LSTMs, let's draw a graph of the performance of the Tesla stock and compare it with that predicted by our model:

```
plot(TeslaData.Date(2:end), YData,TeslaData.Date(2:end),
YPred,'--')
xlabel('Time (years)')
ylabel('Stock price ($)')
legend('ActualData', 'PredictedData')
```

The following graph is printed (*Figure 9.6*):

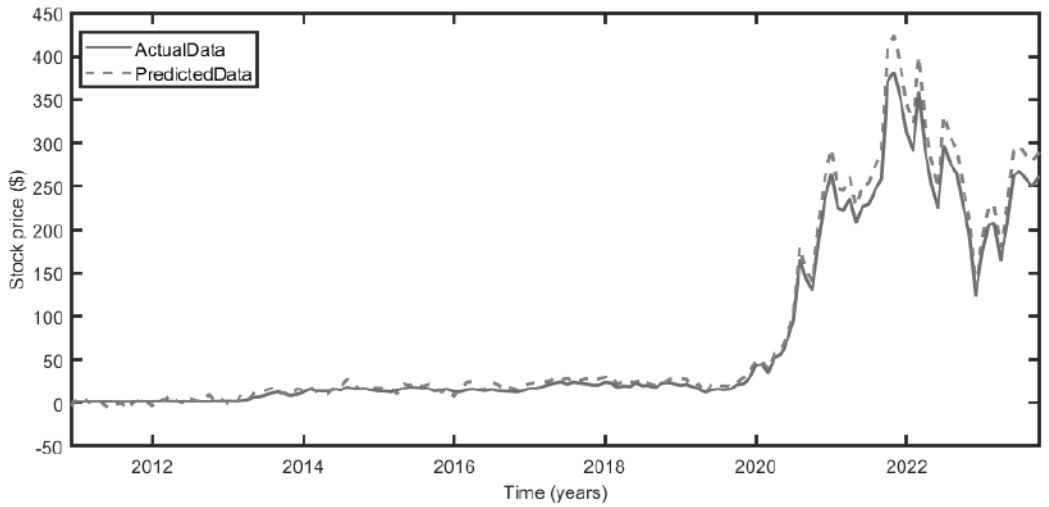


Figure 9.6 – Actual values versus predicted values

We can appreciate the goodness of the forecast in periods in which the situation does not undergo major variations. On the contrary, a certain deviation is noted in correspondence with the periods of greatest fluctuation of the security, demonstrating that it is necessary to consider further information for a correct prediction.

Now, let's see what we can do when we must work with unbalanced data.

Dealing with imbalanced datasets in MATLAB

Dealing with imbalanced datasets is a common challenge in machine learning, particularly in classification tasks where one class significantly outnumbers the other(s). Handling imbalanced datasets is crucial because models trained on such data may exhibit bias toward the majority class and perform poorly in predicting the minority class.

Understanding oversampling

Oversampling is a method that's employed to tackle class imbalance in a dataset by augmenting the number of instances belonging to the minority class. The aim is to balance the class distribution and prevent machine learning models from being biased toward the majority class. Oversampling is particularly useful when you have limited data for the minority class. There are several methods for oversampling, including the following:

- **Random oversampling**: In random oversampling, you randomly select and duplicate instances from the minority class until the class distribution is balanced. This method is straightforward but may lead to overfitting as it introduces duplicate data points.
- **Synthetic Minority Over-sampling Technique** (**SMOTE**): SMOTE is a popular oversampling method that creates synthetic samples for the minority class. It works by selecting a minority class instance and its *k* nearest neighbors. Then, it generates synthetic data points along the line segments connecting the chosen instance and its neighbors. SMOTE helps to prevent overfitting and can be more effective than random oversampling.
- **Adaptive Synthetic Sampling** (**ADASYN**): ADASYN is an extension of SMOTE that's designed to generate additional synthetic samples for minority class instances that pose greater difficulty in classification. It considers the distribution of the data and assigns different weights to different instances.
- **Borderline-SMOTE**: Borderline-SMOTE is a variation of SMOTE that specifically targets instances near the borderline between the minority and majority classes. It generates synthetic samples for these borderline instances to improve classification performance.
- **Minority Synthetic Over-sampling Technique** (**MSMOTE**): MSMOTE is an extension of SMOTE that takes into account the density of minority class instances. It generates synthetic samples based on the local density of the data points.
- **Random oversampling with noise**: This technique involves adding a small amount of random noise to the duplicated instances during random oversampling. This helps to reduce overfitting.
- **Informed oversampling**: Informed oversampling methods consider domain knowledge or specific characteristics of the data to guide the generation of synthetic samples.

The choice of oversampling method depends on the characteristics of your dataset and the problem you are trying to solve. Experimenting with different techniques and evaluating the impact on model performance using appropriate metrics (for example, F1 score, precision-recall curves, and so on) is often necessary to determine the most effective oversampling strategy for your specific use case. It's important to note that oversampling may not always be the best solution, and you should also consider other techniques, such as undersampling, cost-sensitive learning, and ensemble methods, to address class imbalance effectively.

Exploring undersampling

Undersampling is a technique that's used to address class imbalance in a dataset by reducing the number of instances in the majority class. The goal is to create a more balanced class distribution, which can help machine learning models avoid bias toward the majority class. Undersampling can be a useful approach when you have a large amount of data in the majority class and want to avoid the potential issues of overfitting that may arise with oversampling the minority class. Here are some key points to consider when exploring undersampling:

- **Random undersampling**: In random undersampling, you randomly select and remove instances from the majority class until the class distribution is balanced. This method is simple but may lead to the loss of valuable data.
- **Cluster-centroid undersampling**: Cluster-centroid undersampling involves clustering the majority class instances and selecting a representative centroid from each cluster to form a smaller dataset. This method retains diversity within the majority class and is less likely to remove informative data.
- **Tomek links**: Tomek links are pairs of instances from different classes that are close to each other but classified incorrectly. Removing the majority class instance of a Tomek link can help improve the boundary between classes.
- **Edited nearest neighbors** (**ENN**): ENN identifies and removes instances from the majority class that do not agree with their neighbors in terms of class labels. ENN can help in reducing noisy samples in the majority class.
- **One-sided selection**: One-sided selection combines both Tomek links and ENN. It uses Tomek links to identify problematic instances and then uses ENN to remove them.
- **NearMiss undersampling**: This method selects the majority class instances that are closest to minority class instances based on distance metrics. NearMiss algorithms ensure that the selected majority class instances are close to the minority class, thus helping in improving class separation.
- **Informed undersampling**: Informed undersampling methods consider domain knowledge or specific characteristics of the data to guide the selection of instances from the majority class.

When exploring undersampling, it's important to consider various aspects. First, choose the undersampling method that is most appropriate for your dataset. The method you choose should depend on the nature of your data, the problem you are trying to solve, and the potential impact on model performance. Also, undersampling may lead to a loss of information, especially if the majority class contains important data points. Be cautious when applying this technique, and consider whether you can obtain more data for the minority class or explore other methods such as oversampling, cost-sensitive learning, or ensemble methods. Remember to evaluate your model's performance using appropriate metrics, and experiment with different approaches to find the best solution for your specific use case. For instance, suppose you have a dataset with 10,000 credit card transactions, out of which only 100 are fraudulent. In this case, you might undersample the non-fraudulent transactions to create a more

balanced dataset. You could randomly select, say, 500 non-fraudulent transactions, resulting in a new dataset with 500 non-fraudulent and 100 fraudulent transactions.

Discovering cost-sensitive learning

Cost-sensitive learning is a machine learning approach that considers the varying costs of misclassifying different classes in a classification problem. In traditional machine learning, all misclassifications are treated equally. However, in many real-world applications, misclassifying one class may have more severe consequences or higher associated costs than misclassifying another class. Cost-sensitive learning aims to optimize the model to minimize these specific costs.

Cost-sensitive learning is related to but different from handling class imbalance. While class imbalance focuses on the unequal distribution of classes, cost-sensitive learning considers the associated costs of misclassification. Cost-sensitive learning often involves the use of cost matrices, which specify the costs of different types of misclassifications. These matrices assign different costs for false positives, false negatives, true positives, and true negatives.

In cost-sensitive learning, the decision threshold for classification can be adjusted to minimize the expected cost. This means that the model might be more or less conservative in its predictions, depending on the cost matrix. Various machine learning algorithms can be adapted to handle cost-sensitive learning. Some algorithms allow you to specify class-specific misclassification costs. For example, in decision trees or ensemble methods, you can assign weights to classes based on their costs.

When working with cost-sensitive learning, it's important to consider evaluation metrics that reflect the cost-effectiveness of the model, such as the weighted F1 score or cost-sensitive versions of precision and recall. Cost-sensitive learning is often applied to anomaly detection problems. In such cases, you want to detect rare and potentially costly events, such as fraud detection in financial transactions.

Keep in mind that cost-sensitive learning involves trade-offs. Reducing the cost of one type of error may increase the cost of another. Careful consideration and domain knowledge are required when designing the cost matrix. Cost-sensitive learning is a valuable approach in situations where misclassification costs are not uniform, and it can help optimize machine learning models for specific real-world applications. It's an essential technique for decision-making systems where the consequences of different errors can be significantly different.

Summary

In this chapter, we explored several key concepts in the field of data analysis and predictive modeling. We started by discussing the basics of time series data, which refers to data that is collected over a certain period and contains a sequential order. The extraction of statistics from such sequential data is then highlighted as an important step in analyzing and understanding patterns within the data.

This chapter also emphasized the implementation of a model to predict stock market data. This involves using various techniques and algorithms to analyze historical stock market data, identify patterns and trends, and make predictions about future stock prices.

Lastly, this chapter addressed the challenge of dealing with imbalanced datasets in MATLAB. An imbalanced dataset refers to a situation where the distribution of classes within the dataset is significantly skewed, making it difficult to train a model accurately. We discussed methods and strategies to handle imbalanced datasets within the MATLAB programming environment.

Overall, this chapter focused on the importance of understanding and analyzing time series data, extracting meaningful statistics, implementing predictive models for stock market data, and addressing the challenges of imbalanced datasets in MATLAB.

In the next chapter, we will understand the basic concepts of recommender systems, how to identify similar users in a dataset, and how to implement a practical case of recommender systems using MATLAB. Finally, we will understand model compression, pruning, and quantization for efficient inference on edge devices.

10
MATLAB Tools for Recommender Systems

A recommender system is a model that's designed to anticipate the preferences of a specific user. When applied to the domain of movies, it transforms into a movie recommendation engine. The process involves filtering items in a database by predicting the user's potential ratings and facilitating the connection of users with the most suitable content in the dataset. This holds significance because, in extensive catalogs, users might not discover all pertinent content. Effective recommendations enhance content consumption and major platforms such as Netflix heavily depend on them to maintain user engagement. In this chapter, we will learn the basic concepts of recommender systems and how to build a **network intrusion detection system** (**NIDS**) using MATLAB.

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

- Introducing the basic concepts of recommender systems
- Finding similar users in data
- Creating recommender systems for network intrusion detection using MATLAB
- Deploying machine learning models

Technical requirements

In this chapter, we will introduce basic machine learning concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. You will also required working knowledge of MATLAB.

To work with the MATLAB code in this chapter, you'll need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `CreditCardData.xlsx`
- `CreditCardFraudDet.m`
- `NDISdata.csv`
- `NDISEnsemble.m`

Introducing the basic concepts of recommender systems

A **recommender system** is a type of information filtering system that's designed to suggest items or content to users based on their preferences, historical behavior, or other relevant factors. These systems are widely used in various online platforms to help users discover products, services, content, and more. Recommender systems involve two primary entities: **users** and **items**. Users are individuals for whom recommendations are generated, and items are the products, content, or services to be recommended. These items can include movies, books, products, news articles, and more.

Recommender systems rely on data that captures the interaction between users and items. This interaction data can include user ratings, purchase history, clicks, views, likes, and any other form of user engagement with items.

There are different types of recommender systems:

- **Collaborative filtering** (**CF**): CF methods make recommendations based on the preferences and behavior of other users.
- **Content-based filtering**: This approach recommends items to users based on the attributes of the items and the user's historical preferences. It focuses on the content and descriptions of items.
- **Hybrid recommender systems**: These systems integrate various recommendation techniques to furnish recommendations that are both more accurate and diverse.

Understanding CF

CF is a popular technique that's used in recommender systems to make personalized recommendations to users based on their interactions and behaviors, as well as the behaviors of similar users. CF assumes that users who have interacted with items in a similar way in the past will have similar preferences in the future.

There are also two main types of CF:

- **User-based**: This type of CF recommends items to a user based on the preferences and behaviors of users who are like them. Compute a similarity score between the target user and all other users in the system. Common similarity metrics include cosine similarity or Pearson correlation. Identify a set of neighbor users who are most like the target user. Recommend items that the target user's neighbors have interacted with, but the target user has not.
- **Item-based**: This type of CF recommends items to a user based on the similarity of items they have interacted with in the past. It calculates the similarity between all pairs of items in the system based on user interactions. Common similarity metrics include cosine similarity and the Jaccard index. For a target user, it can identify the items they have interacted with. It can recommend items that are like the items the user has interacted with.

CF matrices are often sparse because users interact with only a small fraction of available items. Techniques such as matrix factorization can help address this issue by finding latent factors in the data. As the number of users and items grows, the computation of user-user or item-item similarity becomes more computationally expensive. Optimizations are required for scalability. CF can struggle to make recommendations for new users or items with little to no interaction history. Techniques such as content-based recommendations or hybrid models are often used to address this problem.

CF often relies on user behavior data, which raises privacy concerns. Privacy-preserving methods such as differential privacy may be employed to protect user information. Various metrics, including **mean absolute error** (**MAE**), **root mean squared error** (**RMSE**), and ranking-based metrics such as precision and recall, are used to evaluate the performance of CF algorithms.

CF has been widely used in recommendation systems in various domains, including e-commerce, movie and music recommendations, social networks, and more. While it is effective at capturing user preferences, it does have limitations, such as the cold start problem and the need for a sufficient volume of user interactions. Hybrid approaches that combine CF with other recommendation techniques can overcome some of these limitations.

Content-based filtering explained

Content-based filtering is a recommendation technique that's used in recommender systems to provide personalized recommendations to users based on the characteristics and features of the items and the user's preferences. Unlike CF, which relies on user-item interactions, content-based filtering focuses on the content of items and attempts to match them with user profiles.

Each item in the system is described by a set of features or attributes. These features can vary widely based on the domain but may include things such as genres, keywords, actors, directors (for movies), authors (for books), and more. For text-based content, natural language processing techniques may be used to extract keywords or topics. The system maintains a user profile or preference vector for each user, which reflects their preferences for different features or attributes. This user profile is built based on the items the user has interacted with or explicitly rated.

To generate recommendations for a user, the system calculates a similarity score between the user's profile and the features of items. The similarity score is often computed using techniques such as cosine similarity, **Term Frequency-Inverse Document Frequency** (**TF-IDF**), or other distance metrics. Items with the highest similarity scores are recommended to the user. These are the items that are most in line with the user's historical preferences and interests.

Content-based filtering is effective at providing personalized recommendations because it considers the individual user's preferences based on item features. It doesn't rely on user-to-user or item-to-item comparisons, making it suitable for new or less active users who might not have much interaction history. The quality of recommendations heavily depends on the accuracy and richness of item descriptions and features. Note that ensuring high-quality metadata is crucial.

While content-based filtering is good at recommending items such as those a user has interacted with before, it may not introduce as much serendipity or novelty in recommendations compared to CF. It can also help address the cold start problem for new items if there is sufficient information available about their features. Many recommender systems combine content-based filtering with CF or other recommendation techniques to improve recommendation quality and address the limitations of each approach.

Content-based filtering is commonly used in various domains such as news recommendation, music recommendation, and e-commerce for suggesting products. When implemented correctly, it can offer valuable recommendations tailored to individual user preferences and needs.

Hybrid recommender systems

Hybrid recommender systems are recommendation systems that combine multiple recommendation techniques to provide more accurate and diverse recommendations. These systems aim to leverage the strengths of different recommendation approaches, such as CF, content-based filtering, and more, while mitigating their weaknesses. Hybrid recommender systems are commonly used in various applications to improve recommendation quality and address the limitations of individual methods.

In weighted hybrid systems, different recommendation techniques are assigned weights that determine their influence on the final recommendations. For example, you might assign a higher weight to CF if historical user interactions are more critical, and a lower weight to content-based filtering for item features. Recommendations are generated by combining the scores from each technique, weighted by their importance.

In switching hybrid systems, the recommendation approach is dynamically chosen based on certain conditions or user characteristics. For instance, if a user has a substantial interaction history, CF might be used, but if they are a new user with minimal history, content-based filtering might be employed. In this approach, the features or scores generated by different recommendation techniques are combined to create a unified feature vector for items or users. This combined feature vector is then used to generate recommendations using any single recommendation method.

Cascade hybrids use the output of one recommendation technique as input to another. For example, content-based filtering may be used to generate an initial set of recommendations. These recommendations can be further refined using CF to improve accuracy.

In meta-level hybrid systems, different recommendation methods are applied independently, and their outputs are combined using a meta-learner or meta-classifier. The meta-learner takes the outputs of individual recommendation methods as inputs and provides the final recommendations. Machine learning algorithms such as decision trees, neural networks, or ensemble methods such as stacking can be used as meta-learners.

In the realm of data analysis and user profiling, the process of finding similar users holds immense significance. Let's delve into the techniques and methodologies that are employed for identifying and categorizing users with similar patterns and behaviors within a dataset.

Finding similar users in data

Fraud has consistently been a pervasive issue in various forms, but the emergence of new technological tools, such as **virtual intelligence** (**VI**), has expanded the avenues for fraudulent activities. In today's world, the use of credit and debit cards has become the standard for making purchases, and as a result, fraud associated with these payment methods is on the rise. The repercussions of such fraud extend beyond impacting just merchants and banks, who are often left shouldering the financial burden.

When a customer falls victim to fraud, they may find themselves burdened with higher interest rates imposed by the bank as they could be categorized as a higher risk profile. Additionally, fraud incidents can tarnish a merchant's reputation and image. If a customer experiences fraud during a transaction, it can erode their trust in the seller, potentially driving them to seek alternatives from competitors for future purchases.

Given a set of credit card transaction data, fraud recognition is the process of identifying whether a new transaction belongs to the class of fraudulent or legitimate transactions. Such a system should not only detect fraudulent transactions but should do so in a cost-effective manner.

Examining a transaction with the goal of classification can be carried out through distinct methods, and two specific approaches are suggested:

- User-level analysis
- Single transaction analysis

These two approaches represent unique strategies, each having its own set of advantages and limitations. Each approach entails certain assumptions, varying in their degree of validity. Regardless, both approaches can yield valuable outcomes. There are situations where the choice between these methods is determined by the nature of the available data as it may not permit an alternative approach.

Analyzing transactions at the individual transaction level refers to an approach where the classification of a transaction is determined by its relationship with all the transactions within the dataset. Various machine learning algorithms can be employed for this purpose. For instance, consider the **k-nearest neighbors** (**k-NN**) algorithm, which classifies a transaction based on its similarity to other transactions in the dataset. If a transaction closely resembles known fraudulent transactions, there's a likelihood it will be categorized as fraudulent, and conversely, it would be deemed a legitimate transaction. This mechanism facilitates the identification of data patterns by leveraging similarities among objects of the same class.

In a nearest neighbors algorithm, a record is categorized by examining all the data in the training set and assigning it the same class as the nearest element. The underlying assumption is that in a multidimensional space, if two records are "close," they likely belong to the same class. To gauge this proximity, it's important to employ a distance metric. One example is the Euclidean distance or, more broadly, the Minkowski metric, as we introduced in *Chapter 4*, *Clustering Analysis and Dimensionality Reduction*.

The nearest neighbor rule results in a space division known as **Voronoi tessellation**. Each element in the training set delineates a region in which patterns will be classified into the same class. To enhance the robustness of this mechanism, one approach is to classify a record by considering the k closest records. The record can then be assigned to the class that has the largest representation among the selected examples. To reduce sensitivity to the choice of k, each record can contribute to the classification based on a weighted scheme determined by its distance from the element to be classified. For this process to work effectively, the attributes must possess consistent value scales, which typically necessitates prior normalization during the data preprocessing phase. Let's learn how to build a credit card fraud detection system:

1. As usual, we will start by importing the data into the MATLAB workspace. For this, we will utilize the credit card fraud detection dataset, which comprises anonymized credit card transactions classified as fraudulent or legitimate:

   ```
   CreditCardData = readtable('CreditCardData.xlsx');
   ```

   The dataset exclusively consists of numerical input variables, derived from a PCA transformation. Among the features, `V1` to `V28` represent the principal components obtained through PCA. The `Class` feature serves as the response variable and holds a value of `1` in cases of fraud and `0` otherwise. To understand how the data is distributed, we can count the occurrences in the `class` column:

   ```
   NumOcc = groupcounts(CreditCardData.Class)
   NumOcc =
           3992
            492
   ```

The `groupcounts()` function provides the distinct combinations of grouping variables for the table or timetable denoted as `T`. It also offers the count of members within each group and the corresponding data percentage, ranging from `0` to `100`. Groups are established based on rows in the variables contained within `groupvars` that share identical unique value combinations. Each entry in the result table corresponds to an individual group.

The dataset is unbalanced on one of the binary classes, we should take this into account when evaluating the results. Classifying imbalanced data is a common challenge in machine learning when one class significantly outnumbers the other(s). The primary class (majority class) often dominates, making it challenging for the model to correctly predict the minority class.

To check how the data is distributed, we can draw a boxplot of the features:

```
boxchart(CreditCardData{:,1:28})
xlabel('features')
ylabel('values')
```

The following graph will be output (*Figure 10.1*):

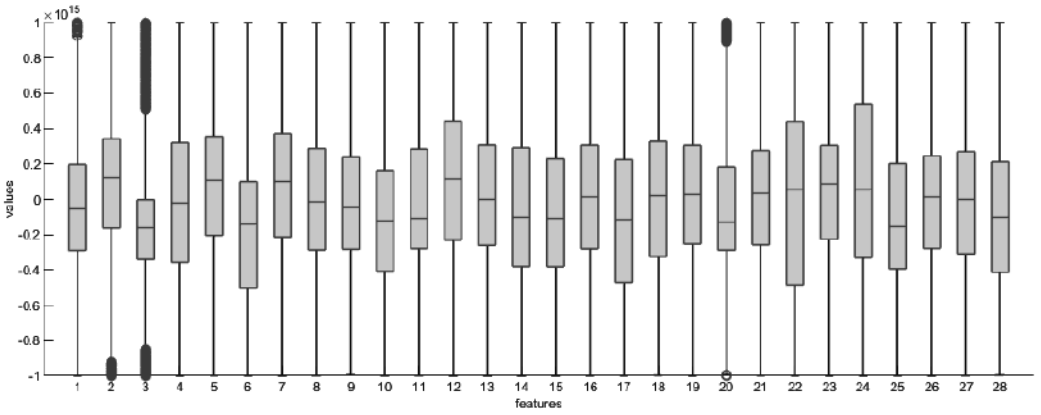


Figure 10.1 – Boxplot of the features

From the analysis of *Figure 10.1*, we can see that the variables are skewed. We can also see that some of them present potential outliers. It is recommended to conduct data scaling. Keep in mind that it's a best practice to standardize or normalize the data before training a machine learning model. Through scaling, the data's units are standardized, making it simpler to compare data from various sources or locations.

2. Data scaling, also known as data normalization or standardization, is a preprocessing technique in machine learning and data analysis that involves transforming the data into a standardized range or distribution. The primary purpose of data scaling is to make different features or variables in your dataset comparable and to help machine learning algorithms perform better.

The choice between scaling and standardization depends on the specific requirements of your dataset and the machine learning algorithm you plan to use. In general, it is advisable to apply data scaling to your features to prevent certain variables from dominating the learning process, especially in algorithms that are sensitive to feature scales, such as kNN or support vector machines.

Data scaling helps in improving the convergence of optimization algorithms, makes features more interpretable, and can also enhance the performance of some machine learning models. However, it's important to note that not all algorithms require data scaling, and there are cases where the natural scale of the data is meaningful and should not be altered. The decision to scale the data should be made with careful consideration of your specific problem and the characteristics of your dataset.

We will perform a standardization with a range of `-1` to `1`:

```
DataScaled = rescale(CreditCardData{:,1:28},-1,1);
```

Standardization with a range of `-1` to `1`, also known as min-max scaling with a specific range, is a method of data scaling that transforms your data to fit within the range of `-1` to `1`. This approach is useful when you want to normalize your data while maintaining the possibility of negative values.

To check how the data is distributed after we performed data scaling, we can draw a boxplot of the features:

```
boxchart(DataScaled)
xlabel('features')
ylabel('values')
```

The following graph is shown (*Figure 10.2*):

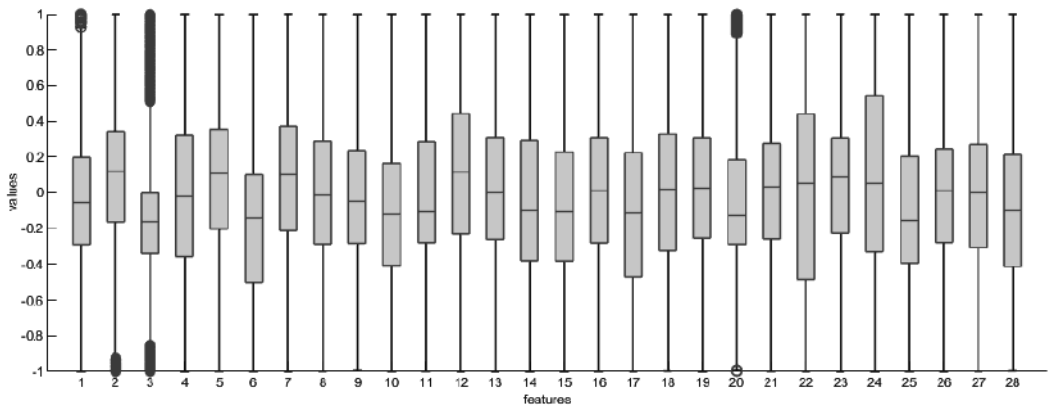


Figure 10.2 – Boxplot of the scaled data

Now, it is clear that the data has been scaled so that we have the same existence intervals.

3. Before training the model based on machine learning, it is necessary to carry out data splitting. Data splitting refers to the process of dividing a dataset into separate subsets for training, testing, and validation purposes. We will perform train-test splitting, which involves dividing the dataset into two parts, typically with a 70-30 or 80-20 split. The larger portion is used for training the model, while the smaller portion is used for testing its performance:

```
n = length(CreditCardData.Class);
SplitData = cvpartition(n,'Holdout',0.3);
TrainIndex = training(SplitData);
TrainData = DataScaled(TrainIndex,:);
TestIndex = test(SplitData);
TestData = DataScaled(TestIndex,:);
```

We initiated the process by obtaining the number of observations within our dataset using the `length()` function. This function returns the size of the largest dimension in the array, `X`. In the context of vectors, this size corresponds to the total number of elements. Subsequently, we employed the `cvpartition()` function to create a random partition for the dataset. This partition serves as the foundation for constructing essential training and test subsets, which are instrumental in evaluating a statistical model. To extract the training data index and the test data index from the original dataset, we utilized the `training` and `test` object functions. These indices were then applied to extract the corresponding data subsets.

4. k-NN is a versatile algorithm that's used not only for classification and regression tasks but also in recommender systems. It can be adapted for building CF-based recommender systems. We will use the `fitcknn()` function, as follows:

```
WeightedKNNModel = fitcknn(...
    TrainData(:,1:28), ...
    TrainData(:,29), ...
    'Distance', 'cosine', ...
    'Exponent', [], ...
    'NumNeighbors', 10, ...
    'DistanceWeight', 'SquaredInverse');
```

The following arguments were passed:

- **Sample data**: `TrainData(:,1:28)`
- **Response variable**: `TrainData(:,29)`
- **Distance metric**: `cosine` (the expression represents the complement of the cosine of the angle included between observations, treating them as vectors):
  - Minkowski distance exponent
  - Number of nearest neighbors to find: `10`
  - Distance weighting function: `SquaredInverse` (`Weight` is 1/distance)

We applied weighted k-NN, a variation of the k-NN algorithm that assigns different weights to the neighbors when making predictions or classifications. In traditional k-NN, each neighbor has an equal influence on the final decision, but in weighted k-NN, the influence of each neighbor is adjusted based on certain factors, typically the proximity or similarity between the neighbors and the query point. This allows for more accurate and context-aware predictions.

We start by choosing a distance metric to measure the similarity or dissimilarity between data points. We used the `cosine` metric, also known as cosine similarity, which is a similarity measure that's used to determine the similarity between two non-zero vectors in a high-dimensional space. It is widely used in various fields, including information retrieval, natural language processing, and recommendation systems. The `cosine` metric measures the cosine of the angle between two vectors, and it provides a value between -1 and 1, where 1 indicates that the vectors are identical, 0 means that they are orthogonal (completely dissimilar), and -1 implies they are opposed.

The algorithm then determines the value of *k*, which represents the number of nearest neighbors to consider when making a prediction. The choice of k depends on your dataset and the specific problem you're trying to solve. You must compute the distances between the query point (the point you want to classify or predict) and all other data points in your dataset, then calculate weights for each neighbor based on their proximity or similarity to the query point. Common methods for assigning weights include inverse distance, where closer neighbors have higher weights, or similarity-based weighting, where more similar neighbors are given higher weights. You can also choose the k neighbors with the highest weights. These neighbors will have a more significant impact on the final prediction. For classification tasks, assign class labels to the query point based on the majority class among the selected neighbors, with the weights influencing the voting process.

Here are the advantages of using weighted k-NN:

- Weighted k-NN can provide more accurate predictions because it considers the influence of each neighbor on the final decision
- It allows for better handling of imbalanced datasets, where some neighbors may be more informative than others
- Weighted k-NN is particularly useful when the neighbors are not equally informative

Weighted k-NN is a flexible and widely used machine learning technique that can be applied to a variety of problems, including classification, regression, and recommendation systems. It allows you to adapt the algorithm to the specific characteristics of your data and the problem you are trying to solve.

5. After training the model, we must evaluate its performance. We will start by using the trained model to predict data labels:

```
PredData = predict(WeightedKNNModel,TestData(:,1:28));
```

Here, we employed the `predict()` function, which furnishes the anticipated response values produced by the generalized linear regression model.

Now, we can assess the model's accuracy, which gauges the extent to which a predictive model's forecasts match the real or observed values. It serves as an indicator of the model's performance in accurately predicting outcomes:

```
accuracy = sum(PredData == TestData(:,29)) /
length(TestData(:,29));
fprintf('Accuracy: %.2f%%\n', accuracy * 100);
```

The following result is printed:

```
Accuracy: 97.03%
```

The result is excellent, confirming that the choice of the algorithm and training parameters were correct.

In the ever-evolving landscape of cybersecurity, the development of effective tools for network intrusion detection is paramount. The next section will explore how to utilize MATLAB to create advanced recommender systems tailored for enhancing network intrusion detection capabilities.

Creating recommender systems for network intrusion detection using MATLAB

A NIDS serves as a security mechanism that's employed to identify and prevent unauthorized access, malicious activities, and potential threats within a computer network. It involves monitoring network traffic and analyzing it to identify any suspicious or abnormal behaviors. The main objective of network intrusion detection is to protect the network from various types of attacks, such as **denial-of-service** (**DoS**) attacks, malware infections, data leakage, unauthorized access, and other cyber threats.

There are two primary methods of network intrusion detection:

- **Signature-based detection**: This method involves comparing network traffic patterns with a database of known signatures or patterns of known attacks. If a match is found, an alert is generated to notify the network administrator.
- **Anomaly-based detection**: This method focuses on identifying abnormal or suspicious network behavior that deviates from the normal patterns. It uses machine learning algorithms to analyze network traffic and detect any anomalies that could potentially indicate an ongoing or imminent attack.

A NIDS can be implemented at different levels within a network architecture, such as at the perimeter, on individual hosts, or within specific network segments. They collect and analyze network packets, logs, and other network data to identify and alert the system administrators about potential intrusions.

The following are some common techniques that are used in network intrusion detection:

- **Packet analysis**: For examining individual network packets to identify specific attack signatures or anomalies
- **Protocol analysis**: For analyzing network protocols to detect any abnormal or unauthorized activities
- **Traffic analysis**: For monitoring network traffic patterns to identify any sudden spikes or unusual patterns
- **Behavior analysis**: For analyzing user or host behavior to detect any unusual or malicious activities

NIDS play a crucial role in safeguarding computer networks against unauthorized access and potential threats. They help identify and respond to security incidents promptly, minimizing any potential damage or data breaches.

Recommender system for NIDS

In this example, we will adopt a new approach to identifying and preventing network attacks: network intrusion detection using a recommender system. Traditional NIDS rely on fixed rules and signatures to detect known attack patterns. However, attackers are continually evolving their techniques, making it challenging for these systems to keep up. By incorporating a recommender system into the network intrusion detection process, it becomes possible to leverage machine learning and data mining techniques to enhance detection capabilities. The primary objective is to utilize historical network traffic data to build a model that can predict whether a particular network event is normal or malicious.

Here is an overview of how network intrusion detection using a recommender system works:

- **Data collection**: Network traffic data is collected and stored for analysis. This data consists of various network parameters, such as IP addresses, port numbers, protocols, packet sizes, and more.
- **Feature extraction**: Relevant features are extracted from the collected data. These features can include traffic volume, connection duration, packet headers, and other characteristics that provide insights into network behavior.
- **Data preprocessing**: The gathered data undergoes preprocessing to eliminate noise, address missing values, and normalize the features. This step ensures that the data is in a suitable format for analysis.
- **Training the recommender system**: The preprocessed data is used to train a recommender system algorithm, such as CF, matrix factorization, or association rule mining. This algorithm learns the patterns and relationships within the data.

- **Building a recommendation model**: The trained algorithm generates a recommendation model based on the network traffic data. This model can identify the normal network behavior and detect any deviations from it.
- **Real-time monitoring**: The recommendation model is then used to monitor incoming network traffic in real time. It analyzes the network events and predicts whether they are normal or potentially malicious.
- **Alert generation**: When the recommendation model detects a potentially malicious network event, it triggers an alert to notify network administrators. The alert can include information about the detected attack type and recommended countermeasures.
- **Continuous learning**: The recommendation model can continuously update itself over time by incorporating new data and adjusting its detection capabilities. This ensures that the system remains effective against emerging threats.

Overall, network intrusion detection using a recommender system offers a more dynamic and adaptive approach to identifying network attacks. It leverages machine learning techniques to learn from historical data and make intelligent predictions about the network's security status. This can enhance the accuracy and efficiency of network security operations by reducing false positives and detecting emerging attack patterns.

NIDS using a recommender system in MATLAB

In this practical example, we will build a NIDS-adopting recommender system using **ensemble methods**. Ensemble methods are techniques that combine multiple individual models to form a more powerful and accurate predictor. These individual models, also known as base models or weak models, can be of any type, such as decision trees, support vector machines, or neural networks. By combining the predictions of these base models, ensemble methods aim to improve the overall performance and generalization ability of the model. There are several popular ensemble methods, including the following:

- **Bagging**: Bagging, short for **bootstrap aggregating**, involves training multiple base models on different subsets of the training data, with replacement. The final prediction is made by averaging or voting the predictions of the individual models.
- **Boosting**: Boosting algorithms train base models sequentially, with each subsequent model focusing more on the samples that were previously misclassified. The predictions of multiple models are combined using weighted voting or averaging to make the final prediction.
- **Random forest**: Random forest is an ensemble method that amalgamates bagging and decision trees. Numerous decision trees, each trained on a random subset of the features, are integrated through majority voting to formulate the final prediction.

- **Adaptive boosting** (**AdaBoost**): AdaBoost is a boosting algorithm that assigns weights to the training samples based on their classification errors. Weak models are trained iteratively, with each subsequent model focusing more on the misclassified samples. The final prediction is made by combining the predictions of multiple models using weighted voting.
- **Gradient boosting**: Gradient boosting is another boosting algorithm that sequentially trains weak models, with each subsequent model minimizing the loss function using gradient descent. The predictions of multiple models are combined using weighted averaging to make the final prediction.

Ensemble methods have proven to be effective in improving the performance and robustness of predictive models in various domains, including classification, regression, and anomaly detection. They are widely used in machine learning and data mining applications.

As usual, we start by importing the dataset into the MATLAB environment. We will use the Network Intrusion Detection dataset, which is available in the Kaggle dataset repository (`https://www.kaggle.com/datasets/sampadab17/network-intrusion-detection`). The dataset contains a diverse range of simulated intrusions in a military network setting. The objective was to create a realistic environment resembling a US Air Force LAN, which was then exposed to multiple attacks. Each connection in this environment represents a sequence of TCP packets, with data flowing between a source IP address and a target IP address under a predefined protocol. Every connection is classified as either normal or as an attack, with a specific attack type assigned to it. A connection record contains approximately 100 bytes of data. For each TCP/IP connection, a total of 41 features, both quantitative and qualitative, are collected from both normal and attack data. These features include three qualitative features and 38 quantitative features. The class variable in the dataset has two categories – normal and anomalous:

1. We start by importing the dataset into the MATLAB environment:

```
NIDSTrainData = readtable('NDISdata.csv');
```

   We can extract some information about the features using the `summary()` function:

```
summary(NIDSData)
```

   This function prints a summary of the table, displaying the properties description of the variables and some statistics such as min, median, and max for numeric features.

2. As we did in the *Finding similar users in data* section, we have to split the data into two subsets: train and test. We will use the `cvpartition()` function for this, as follows:

```
n = length(NIDSData.class);
SplitData = cvpartition(n,'Holdout',0.3);
TrainIndex = training(SplitData);
TrainData = NIDSData(TrainIndex,:);
TestIndex = test(SplitData);
TestData = NIDSData(TestIndex,:);
```

We began by using the `length()` function to determine the number of observations in our dataset. This function gives us the size of the largest dimension in the array, `X`, which in the case of vectors represents the total number of elements. Next, we utilized the `cvpartition()` function to randomly split the dataset into training and test subsets. This partition forms the basis for evaluating a statistical model. We extracted the indices for the training and test data using the training and test object functions and then used these indices to obtain the respective data subsets.

3. Now, it's time to train the algorithm for NDIS. To do this, we will use an algorithm based on ensemble methods:

```
predictors = TrainData(:,1:41);
response = TrainData.class;
```

We started by setting the data used for the training and divided the subset into predictors and response variables. These terms are used in statistical analysis to describe the relationship between independent variables (predictors) and dependent variables (response). Predictors are variables that are used to predict, explain, or account for the variation in the response variable. They can be continuous or categorical and may have various levels or values. The response variable, on the other hand, is the outcome or variable that is being predicted or explained by the predictors. It is also known as the dependent variable. In our case, it is the label of the network intrusion detection classification (anomaly, normal).

4. Now, we can train the model:

```
NDISEnsModel = fitcensemble(...
    predictors, ...
    response, ...
    'Method', 'AdaBoostM1');
```

As a classification method, we used `AdaBoostM1`. `AdaBoostM1`, also known as adaptive boosting, is a machine learning algorithm that can be used for both classification and regression problems. It is particularly effective in handling complex datasets and improving weak learning algorithms. The fundamental concept behind `AdaBoostM1` is to amalgamate multiple weak classifiers to form a robust classifier. A weak classifier is a basic model that performs marginally better than random guessing. `AdaBoostM1` accomplishes this by iteratively training weak classifiers on varied weighted versions of the dataset. The main idea behind `AdaBoostM1` is to combine multiple weak classifiers to create a strong classifier. A weak classifier is a simple model that performs slightly better than random guessing. `AdaBoostM1` achieves this by iteratively training weak classifiers on different weighted versions of the dataset.

In each iteration, the algorithm assigns increased weights to instances misclassified in the previous iteration. This compels the weak classifiers to prioritize the most challenging instances, enhancing their focus on difficult cases. These weighted weak classifiers are then combined using a weighted majority voting scheme to make final predictions. In classification tasks, the final prediction is obtained by assigning a class label based on the weighted majority vote of the weak classifiers. In regression tasks, the final prediction is obtained by averaging the weighted predictions of the weak classifiers.

`AdaBoostM1` has several advantages. It is robust to overfitting and can handle datasets with many features. It can also handle imbalanced datasets by adjusting the weights of the instances. Additionally, `AdaBoostM1` can be easily parallelized, making it suitable for distributed computing environments. However, `AdaBoostM1` may be sensitive to noisy data and outliers, as it assigns higher weights to misclassified instances. It also requires careful selection of weak classifiers, as too complex or too weak classifiers may not yield good results. Overall, `AdaBoostM1` is a powerful algorithm that has been widely used in many applications, including face detection, object recognition, and financial forecasting.

Let's take a look at the model that was trained:

```
NDISEnsModel
NDISEnsModel =
  ClassificationEnsemble
            PredictorNames: {1×41 cell}
              ResponseName: 'Y'
     CategoricalPredictors: [2 3 4]
                ClassNames: {'anomaly'  'normal'}
           ScoreTransform: 'none'
          NumObservations: 17635
               NumTrained: 100
                   Method: 'AdaBoostM1'
             LearnerNames: {'Tree'}
      ReasonForTermination: 'Terminated normally after completing
the requested number of training cycles.'
                  FitInfo: [100×1 double]
       FitInfoDescription: {2×1 cell}
```

Some information about the model is shown in the preceding code.

5. After training the algorithm, it is time to evaluate its performance:

```
PredData = predict(NDISEnsModel,TestData(:,1:41));
```

The `predict()` function was utilized to obtain the expected response values generated by the generalized linear regression model. By evaluating the accuracy of the model, we can measure how well the predictive model's predictions align with the actual observed values. This assessment serves as a performance indicator in accurately forecasting outcomes:

```
accuracy = sum(strcmpi(PredData,TestData{:,42})) /
height(TestData(:,42));
fprintf('Accuracy: %.2f%%\n', accuracy * 100);
```

To compare predicted data with the actual data, we used the `strcmpi()` function, which compares two strings, disregarding any discrepancies in letter case. It returns *1* (true) if the two are the same, and *0* (false) otherwise. To determine if the texts are identical, both their size and content are considered, except for case sensitivity. The output variable is of the logical type. The input arguments can comprise various combinations of string arrays, character vectors, and cell arrays of character vectors.

The following accuracy was obtained after running the preceding code:

```
Accuracy: 99.79%
```

These are amazing results that demonstrate that ensemble methods are very effective in classifying network intrusion.

Embarking on the final frontier of the data science journey, the deployment of machine learning models marks the critical phase where theoretical prowess transforms into real-world impact. The next section delves into the intricacies and best practices of deploying machine learning models, ensuring their seamless integration into operational environments.

Deploying machine learning models

Deploying machine learning models refers to the process of making a trained model available for making predictions on new, unseen data. It involves taking the trained model and integrating it into a production environment where it can receive input data, perform predictions, and return the results. The trained model needs to be organized and packaged into a format suitable for deployment. This may involve exporting the model into a file format that can be easily loaded and used by other systems. An **application programming interface** (**API**) is typically created to expose the machine learning model's functionality. The API acts as the interface that other systems or applications can use to send data and receive predictions from the model.

If the model is expected to handle many concurrent requests, the deployment environment may need to be scaled to accommodate the increased load. This may involve setting up clusters of servers or using cloud-based infrastructure. Once the model has been deployed, it is important to monitor its performance and behavior to ensure it continues to provide accurate predictions. Monitoring can involve tracking metrics such as response time, throughput, or error rates to identify any issues or performance degradation. Machine learning models often need updates or retraining as new data becomes available. Therefore, it is important to have processes in place for continuous integration and delivery to easily deploy new versions of the model and ensure it stays up to date.

Understanding model compression

Model compression refers to the process of diminishing the size of a machine learning model without significantly compromising its performance. The need for model compression arises in scenarios where the size or computational requirements of a model become critical, such as when deploying models on resource-constrained devices such as smartphones or edge devices.

There are several techniques for model compression:

- **Pruning**: Pruning involves removing unnecessary connections or parameters from the model. Typically, connections with small weights are pruned based on a certain threshold, resulting in a sparser model with fewer parameters.
- **Quantization**: Quantization is the process of reducing the precision of the weights and activations in a model. For example, instead of using 32-bit floating-point values, weights can be represented using 8-bit integers. This reduces memory requirements and improves computational efficiency.
- **Knowledge distillation**: Knowledge distillation involves training a smaller, "student" model to mimic the predictions of a larger, "teacher" model. The teacher model's knowledge is transferred to the student model, allowing for a compact representation of the original model while maintaining performance.
- **Low-rank approximation**: Low-rank approximation techniques aim to approximate the weights of a model using low-rank matrices or tensors. This reduces the number of parameters required to represent the model, leading to a smaller memory footprint and faster computations.
- **Compact architectures**: This involves designing or using compact architectures, such as MobileNet or SqueezeNet, that are specifically built to be lightweight and efficient.

These compression techniques enable models to be deployed on devices with limited resources or used in scenarios with strict latency or memory constraints. However, it is important to consider trade-offs as model compression may lead to slight performance degradation in terms of accuracy or inference speed.

Discovering model pruning techniques

Model pruning is a machine learning technique that aims to decrease the size and complexity of a trained model by eliminating unnecessary or redundant parameters, connections, or nodes. The objective of model pruning is to enhance the efficiency and computational performance of the model without significantly compromising accuracy.

There are several methods of model pruning, including the following:

- **Weight pruning**: In weight pruning, individual weights in the model are set to zero or removed entirely based on their magnitude. This reduces the number of parameters and can result in a sparser model.

- **Neuron pruning**: Neuron pruning entails the removal of entire neurons from the model, guided by their contribution to the overall performance. Neurons with low activation or minimal impact on the output are pruned.
- **Filter pruning**: This technique is commonly used in **convolutional neural networks** (**CNNs**). Filters are groups of feature detectors that are applied across an image during the convolution operation. Filter pruning involves removing unnecessary filters that do not contribute significantly to the model's accuracy.
- **Structured pruning**: Structured pruning involves removing entire layers or blocks of the model, rather than individual weights or neurons. This method often results in more efficient implementations since removing entire layers reduces both computational complexity and memory requirements.

Model pruning can be performed during training or as a post-training optimization step. It is often combined with other techniques such as regularization methods or quantization to further improve the efficiency of the pruned model. Pruning can provide significant benefits in terms of model size, speed, and memory usage, making it a useful technique for deploying models on resource-constrained devices or in real-time applications.

In MATLAB, you can perform model pruning using various techniques and tools such as Neural Network Toolbox. The following steps outline a general approach to model pruning in MATLAB:

1. Load or create your initial neural network model using Neural Network Toolbox in MATLAB.
2. Train the neural network model to achieve a reasonably good performance on a given task or dataset.
3. Use the pruning algorithm or technique of your choice to identify and remove unnecessary weights, connections, or nodes from the trained model. Some commonly used pruning techniques include magnitude-based pruning, sensitivity-based pruning, and weight-decay pruning.
4. Evaluate the pruned model's performance by testing it on a validation or test dataset. Make sure that the pruning process does not significantly degrade the model's performance.
5. Fine-tune or retrain the pruned model, if necessary, to recover any performance degradation due to pruning. Adjust the learning rate or other training parameters to optimize the pruned model's performance.
6. Repeat *steps 3* to *5* until the desired level of model compression or performance trade-off is achieved.

Remember that model pruning is a dynamic and iterative process, and it requires careful consideration of the trade-off between model size, performance, and computational requirements.

Introducing quantization for efficient inference on edge devices

Quantization is a technique that's used in model compression to reduce the memory footprint and computational requirements of deep neural networks. It involves converting the weights and activations of a network from floating-point representation into lower precision representation, such as 8-bit or even lower.

Quantization reduces the precision of the model, which can lead to a loss in model accuracy. However, it has been observed that many neural networks are robust to quantization and can still achieve similar performance with reduced precision representation. This is especially true for deep neural networks with large numbers of parameters.

There are different approaches to quantization in model compression:

- **Weight-only quantization**: In this approach, only the weights of the neural network are quantized, while the activations remain in the original precision. This reduces the memory requirements significantly as weights typically consume the majority of the memory in a neural network.
- **Full quantization**: In this approach, both the weights and activations are quantized to lower precision. This provides further reduction in memory requirements and computational complexity but can result in a larger loss in model accuracy compared to weight-only quantization.
- **Dynamic quantization**: Dynamic quantization techniques adaptively adjust the precision of weights and activations during inference based on the input data. This allows for more flexibility in the precision used, leading to potential accuracy improvements compared to static quantization.
- **Quantization-aware training**: Instead of quantizing the model after training, quantization-aware training incorporates the quantization process during the training phase itself. This ensures that the model learns to be robust to quantization and results in better accuracy when quantized.

Quantization is a widely used technique in model compression and is particularly useful when deploying deep neural networks on resource-constrained devices such as edge devices or mobile phones. It allows for efficient deployment of models with reduced memory requirements and improved computational efficiency, without sacrificing significant model accuracy.

Getting started with knowledge distillation

Knowledge distillation is a technique that's used in model compression that refers to the process of reducing the size and complexity of a machine learning model without significant loss in performance. In knowledge distillation, a large "teacher" model is trained on a dataset and used as a reference to train a smaller "student" model. The goal is to transfer the knowledge, or the learned representations, from the teacher model to the student model.

Knowledge transfer is achieved by training the student model to mimic the outputs of the teacher model. Typically, this involves using the teacher model's soft targets or logits instead of the hard labels during training. Soft targets refer to the probabilities assigned to each class by the teacher model, which can provide more nuanced information compared to the one-hot encoded hard labels.

During the training process, the student model tries to minimize the difference between its predictions and the soft targets provided by the teacher model. This enables the student model to learn from the richer information provided by the teacher, improving its understanding of the data and increasing its performance. Knowledge distillation can lead to significant model compression as the smaller student model can capture the knowledge of the larger teacher model, often with fewer parameters. Additionally, the student model can be more efficient in terms of inference time and resource usage, making it suitable for deployment on resource-constrained platforms.

In summary, knowledge distillation is a technique that's used in model compression to transfer knowledge from a larger teacher model to a smaller student model, thereby compressing the model size while maintaining performance.

Learning low-rank approximation

Low-rank approximation is a technique that's used in model compression to reduce the size of a given model. It involves approximating a high-dimensional weight matrix or tensor by a lower-rank approximation, which significantly reduces the number of parameters needed to represent the model. In low-rank approximation, a factorization of the weight matrix is performed to decompose it into two or more smaller matrices or tensors. The rank of the approximation determines the number of smaller matrices or tensors used. By choosing a lower rank, the resulting approximation will have fewer parameters, making it more compact and efficient.

Low-rank approximation can be applied to various types of models, including neural networks, deep learning models, and other machine learning algorithms. It is especially useful for reducing the computational and memory requirements of large models. This enables their deployment on resource-constrained devices, such as mobile phones or embedded systems. One common approach for low-rank approximation is **singular value decomposition** (**SVD**), which decomposes a matrix into three matrices representing the left singular vectors, singular values, and right singular vectors. Selecting a subset of the singular values and their corresponding singular vectors allows for the creation of a low-rank approximation.

Other techniques for low-rank approximation include Tucker decomposition, which decomposes a tensor into smaller tensors, and tensor-train decomposition, which represents a tensor as a series of matrix products. These techniques can be applied to higher-order tensors typically found in deep learning models.

Overall, low-rank approximation is a powerful technique for model compression, enabling the reduction of model size without sacrificing too much performance. It allows for the efficient deployment of models on resource-constrained devices and faster inference times.

Summary

In this chapter, we learned the basic concepts of recommender systems, starting with the definition of these systems and then understanding how the problem is approached. We analyzed the different types of recommender systems: CF, content-based filtering, and hybrid recommender systems.

Next, we saw how to use similarities in the data to identify possible fraudulent uses of credit cards. To do this, we trained a model based on the nearest neighbor algorithm but using a modified version of the traditional k-NN algorithm, where neighbors are given varying weights during the prediction or classification process.

Then, we saw how to implement a NIDS based on ensemble methods in MATLAB. Specifically, we adopted an AdaBoost algorithm to identify intrusions in a LAN network.

Finally, we introduced the techniques of deploying machine learning models regarding model compression. We analyzed the most popular model compression techniques, including pruning, quantization, knowledge distillation, and low-rank approximation.

In the next chapter, we will learn the basic concepts of anomaly detection and fault diagnosis systems. We will understand how to identify anomaly functioning using deep learning, as well as how to implement a fault diagnosis system in MATLAB. Finally, we will discover dropout, L1 and L2 regularization, and early stopping.

11
Anomaly Detection in MATLAB

Throughout the life cycle of a physical system, the occurrence of failures or malfunctions poses a potential threat to its normal functioning. To safeguard against critical interruptions, it becomes imperative to implement an anomaly detection system within the facility. Termed as a **fault diagnosis system**, this mechanism is designed to identify potential malfunctions within the monitored system. The pursuit of fault detection stands as a pivotal and defining phase in maintenance interventions, demanding a systematic and deterministic approach to comprehensively analyze all conceivable causes that might have led to the malfunction.

In this chapter, we will learn the basic concepts of anomaly detection systems and how to implement an anomaly detection system in MATLAB.

We're going to cover the following main topics:

- Introducing anomaly detection and fault diagnosis systems
- Using **machine learning** (**ML**) to identify anomalous functioning
- Building a fault diagnosis system using MATLAB
- Understanding advanced regularization techniques

Technical requirements

In this chapter, we will introduce basic ML concepts. To understand these topics, a basic knowledge of algebra and mathematical modeling is needed. A working knowledge of the MATLAB environment is also required.

To work with the MATLAB code in this chapter, you need the following files (available on GitHub at `https://github.com/PacktPublishing/MATLAB-for-Machine-Learning-second-edition`):

- `GearboxAccData.xlsx`
- `AnomalyDetectGearBox.m`

- `DroneFaultDiagnosis.xlsx`
- `UAVFaultDiagnosis.m`

Introducing anomaly detection and fault diagnosis systems

Anomaly detection and fault diagnosis systems are crucial components of various industries, particularly in areas where safety, reliability, and efficiency are of utmost importance, such as manufacturing, healthcare, finance, and cybersecurity. These systems aim to identify unusual or unexpected patterns, behaviors, or conditions in data, processes, or systems that may indicate the presence of faults, defects, or anomalies.

Delving into the realm of anomaly detection, this section provides a comprehensive overview, unraveling the key principles and methodologies employed in identifying deviations from the norm within diverse systems and datasets.

Anomaly detection overview

Anomaly detection is a technique used in data analysis and ML to identify data points or patterns that deviate significantly from the expected or normal behavior within a dataset. Anomalies, also known as outliers, are data points that do not conform to most of the data and may indicate errors, fraud, unusual events, or other important information. Anomaly detection has various applications across different domains, such as cybersecurity, industrial **quality control** (**QC**), finance, healthcare, and more.

We can start to get an overview of different types of anomalies to understand what is intended with this term, we will list some types of anomalies:

- **Point anomalies**: These are individual data points that are considered anomalies, such as a single fraudulent transaction in a credit card dataset.
- **Contextual anomalies**: These are anomalies that are context-dependent. A data point might not be an anomaly on its own but is unusual in a particular context or time, such as a sudden spike in web traffic during a holiday sale.
- **Collective anomalies**: These are anomalies that are identified by examining a group of data points collectively. These anomalies involve patterns or relationships between data points.

There are several methods for addressing anomaly detection problems, ranging from simple statistical techniques to complex ML algorithms. The choice of method depends on the nature of the data and the specific problem you are trying to solve. Here, we are listing the most used ones:

- **Statistical methods**: Statistical techniques such as z-scores, percentiles, and boxplots can be used to identify anomalies based on deviations from the mean or median of the data distribution.

- **ML**: Supervised, unsupervised, and semi-supervised ML algorithms can be used for anomaly detection. Some popular methods include Isolation Forest, **One-Class Support Vector Machine** (**One-Class SVM**), **autoencoders** (**AEs**), and k-means clustering.
- **Time series analysis**: Specialized techniques are used for detecting anomalies in time series data, such as **autoregressive** (**AR**) models, exponential smoothing, and **moving averages** (**MAs**).
- **Density estimation**: Methods such as **kernel density estimation** (**KDE**) and **Gaussian Mixture Models** (**GMMs**) are used to estimate the probability density function of the data and identify anomalies as low-density regions.
- **Deep learning (DL)**: **Neural networks** (**NNs**), especially **deep AEs** (**DAEs**) and **recurrent NNs** (**RNNs**), are used for anomaly detection in high-dimensional data or sequences.
- **Ensemble methods**: Combining multiple anomaly detection models can improve overall performance and robustness.

In addressing anomaly detection problems, we have to face some challenges. For example, determining an appropriate threshold for defining anomalies can be challenging. Imbalanced datasets, where anomalies are rare, can make model training and evaluation tricky. Handling high-dimensional data and noisy datasets can also be challenging.

Anomaly detection is a valuable tool for identifying rare but potentially important events or patterns in large datasets. The choice of method depends on the specific domain, data characteristics, and the nature of anomalies that need to be detected.

Fault diagnosis systems explained

Diagnostics refers to a procedure that translates information obtained from measuring parameters and collecting data about a machine into insights about its current or potential failures. This process encompasses a combination of analytical and synthetic activities, utilizing physical measurements and machine-specific characteristics to derive valuable information regarding the machine's condition and its future trends. This information is crucial for assessing both short-term and long-term reliability.

The adoption of fault diagnosis techniques is becoming increasingly important to ensure high levels of safety and dependability in automated and autonomous systems. Indeed, over recent years, the global scientific community has invested significant efforts in developing systematic approaches for diagnosing failures in a wide array of systems. The primary objective of a fault diagnosis system is to continuously monitor a system during its operation to achieve three key goals: detecting the presence of faults (fault detection), pinpointing the specific location of these faults (fault isolation), and understanding how these faults evolve over time (fault identification).

Usually, the output of a fault diagnosis system provides a group of sensitive variables that are influenced by the type of fault, with some modifications when the system experiences a failure. Subsequently, the system extracts and processes information embedded within these fault occurrences to carry out the tasks of detecting, isolating, and identifying faults. The techniques employed for fault diagnostics can be categorized into three fundamental groups:

- **Model-based**: This approach relies on precise mathematical models, enabling efficient fault detection and diagnosis. These models are constructed to depict the actual degradation processes of the components under scrutiny. This involves describing, in terms of the laws of physics, how operational conditions influence the performance and lifespan of assets. Key variables encompass various thermal, mechanical, chemical, and electrical parameters. Expressing their impact on machinery's health is a challenging endeavor, demanding a high degree of domain expertise and modeling skills from those creating such solutions. Once the model is established, it becomes essential to have sensors that can provide data regarding the relevant quantities identified during the analysis and modeling phase. This data serves as input for the model. The primary advantage of this approach lies in its descriptiveness. It allows for a detailed analysis of the causes behind each output it generates because it's rooted in a physical representation of the process. This, in turn, enables validation and certification. However, its accuracy heavily relies on the quality of analysis and modeling conducted by domain experts. Conversely, drawbacks include its complexity and high implementation costs, along with its high specificity to the system, limiting possibilities for reuse and expansion.
- **Knowledge-based**: This approach also leans on domain experts, aiming to model the expertise and behavior of these experts directly. The objective is to formalize the knowledge they possess, enabling its automated replication and application. Expert systems are essentially software programs that leverage knowledge repositories collected from proficient individuals in various fields. These systems then employ inference and reasoning mechanisms to emulate human thought processes, offering support and solutions for practical issues. Two prevalent techniques for implementing this type of model are rule-based mechanisms and fuzzy logic. Rule-based approaches are valued for their simplicity of implementation and interpretability. However, they may fall short in expressing complex conditions and may suffer from a combinatorial explosion when dealing with many rules. On the other hand, the use of fuzzy logic permits describing the system's state through more imprecise and less rigid inputs, simplifying the formalization and model description process and making it more intuitive. As with model-based methods, the effectiveness of expert systems heavily hinges on the quality and level of detail achieved by the model, resulting in highly specific solutions.
- **Data-based methods**: The data-based approach applies statistical and ML techniques to the data gathered from machines, with the goal of discerning the condition of components. The aim is to acquire comprehensive real-time information about the machinery, typically through sensors and production and maintenance activity logs, and to correlate this information with the individual components' degradation levels or the system's performance. Presently, this approach

is most prevalent in practical applications. This is attributed to several advantages it offers over the other two methods. Data-driven methods require substantial data volumes for effectiveness. However, given the accessibility of modern interconnected sensors, this requirement is generally easy to fulfill. In contrast to the other approaches, they have the significant advantage of not mandating in-depth domain-specific expertise, thereby reducing experts' impact on the model's final performance. While expert insights can facilitate the selection of input variables, their influence is less pronounced compared to other methods. Additionally, ML and data mining techniques may uncover relationships between input parameters and system states that even experts might not know beforehand. ML-based algorithms can be harnessed to build predictive models that yield domain-specific knowledge.

Embarking on the intersection of machinery and intelligence, this section explores a cutting-edge approach to fault diagnosis by harnessing the power of ML. Unraveling the potential of algorithms and models, we delve into how these technological advancements enable a more precise and efficient identification of faults within complex systems.

Approaching fault diagnosis using ML

ML algorithms autonomously derive knowledge from data through the inputs they receive, eliminating the need for explicit developer instructions. In such models, the machine independently identifies the patterns required to achieve the desired outcome, a hallmark of **artificial intelligence** (**AI**). During the learning process that characterizes these algorithms, the system is supplied with a training dataset, enabling it to estimate the relationships between input and output data. These relationships constitute the parameters of the model that the system infers.

The selection of a particular ML model hinges heavily on the intended goal of the system. The nature of the problem is modeled differently based on the objective, and two primary approaches can be identified: diagnostics and prognostics. A diagnostic system is designed to pinpoint and identify faults when they occur. This involves continuous monitoring of a system, alerting when deviations from the expected behavior are detected, specifying which component is affected by the anomaly, and categorizing the type of anomaly.

In contrast, prognostics seeks to predict whether a fault is imminent or to estimate the probability of its occurrence. As a proactive analysis, prognostics can contribute significantly to cost reduction in terms of interventions, but it represents a more intricate objective to accomplish. Another viable option is the simultaneous application of diagnostic and prognostic solutions to the same system. This combination offers two valuable advantages: diagnostics can provide support in cases where prognostics falls short. This situation is inevitable because some failures do not follow predictable patterns, and even when certain failures are predictable with a high degree of accuracy, they may not be identifiable in every instance. The information gleaned from diagnostic applications can serve as supplementary input for predictive systems, enabling the creation of more sophisticated and precise models.

Fault diagnosis problems can be addressed using the following two types of approaches depending on the type of data:

- **Regression**: Regression can serve as a modeling technique for prognostic challenges. This entails estimating the remaining useful life of a component in continuous time units. In this scenario, the training dataset should exclusively encompass data from components that have experienced failures, facilitating the labeling of inputs starting from the failure event backward in time. This approach to the problem still adheres to the **supervised learning** (**SL**) framework, where input data is linked to continuous output values in this context.
- **Binary classification**: The simplest approach is to frame fault detection as a binary classification problem, where each input representing the system's state must be categorized into one of two exclusive values. In the context of diagnostic problem-solving, this entails determining whether the machine is operating correctly or incorrectly, categorizing all possible states into these two classes. This process aligns with SL, as the input data is paired with labels that represent the model's output. In such cases, the system learns to discern relationships between the input data and the corresponding labels. In the case of prognostics, the task shifts toward assessing whether the machine could potentially fail within a specified time frame. The distinction between these two meanings arises solely from the differing interpretations of the labels. This implies that the same model can be employed to address both issues. The differentiation lies in how the dataset is labeled during the model training phase.
- **Multiclass classification**: The multiclass variant represents an extension of binary classification, introducing a broader range of available labels for selection. Nevertheless, each input is still associated with only one label. In the diagnostic scenario, the extension follows a straightforward approach, involving the determination of whether the machine is operating correctly or not. In the latter case, it's about pinpointing the specific anomaly state from various possibilities. When considering prognostics applications, the problem shifts toward determining the time interval before the machine's potential failure. Here, the labels denote different intervals representing proximity to a failure event.

For a precise diagnosis of faults in intricate machinery, it is imperative to gather information by collecting data, employing advanced signal processing algorithms for data analysis, and subsequently extracting the appropriate features to efficiently identify and classify faults. Data can be acquired through various methods, primarily via measurements of physical attributes that describe the machine's state during operation. These measurements are obtained by specialized sensors that convert physical properties into electrical values, often referred to as sensor data.

It's worth noting that the choice of parameters depends greatly on the specific system under scrutiny, with examples including noise, vibrations, pressure, temperature, and humidity. Typically, in modern automated industrial systems, the necessary diagnostic data is readily available. In cases where it's not, adding extra sensors becomes the initial step in establishing an effective fault identification strategy. Additionally, data can also be gathered by linking the machine's static operating conditions to specific time points, incorporating factors such as material codes, machine production speed, or the type of product being manufactured. This type of data is categorized as statistical data.

Finally, data may encompass the history of significant events and actions related to a machine and its components, which is often referred to as log data. This can include records of faults, repairs, replacements, and other pertinent activities.

With the foundational concepts of fault diagnosis explored, it's now time to practically address a simulation problem.

Using ML to identify anomalous functioning

A gearbox, also known as a gear mechanism or transmission, is a mechanical device designed to transmit mechanical power from one component to another while altering the speed, torque, and direction of rotation. Gearboxes consist of a set of gears with different sizes and configurations that mesh to provide specific mechanical advantages based on the desired output.

In an automotive context, a gearbox, often referred to as a transmission or gearshift, is a critical component that plays a central role in controlling the power output of the engine and the vehicle's speed. Anomaly detection in a gearbox involves identifying irregularities or deviations from the normal behavior of the gearbox components and their associated systems. Detecting anomalies in a car's gearbox is crucial for ensuring the vehicle's safety, reliability, and overall performance.

To promptly identify anomalies in a gearbox, we can use sensors such as those measuring temperature, pressure, speed, and fluid levels to continuously collect data. These sensors provide valuable information about the gearbox's condition. Raw sensor data often contains noise and outliers. Data preprocessing techniques, such as filtering and data cleaning, are applied to ensure the quality and reliability of the data. Relevant features are extracted from the preprocessed data. These features may include gear engagement patterns, rotational speeds, and temperature fluctuations. Anomaly detection typically begins by establishing a baseline model of the normal behavior of the gearbox. This is done by analyzing historical data to understand typical operating conditions and performance.

Various anomaly detection techniques are applied to identify deviations from the established normal behavior. Some common techniques include the following:

- **Statistical methods**: These methods flag data points that significantly deviate from the mean or follow a different distribution
- **ML algorithms**: These models can learn to recognize patterns in the data and detect anomalies
- **Time series analysis**: For gearbox data collected over time, time series methods such as **Autoregressive Integrated Moving Average** (**ARIMA**) or **long short-term memory** (**LSTM**) NNs can be used to detect anomalies

Anomalies are detected based on predefined thresholds or through statistical analysis of the data. These thresholds are determined during the model training phase and may be adjusted over time to optimize detection. When an anomaly is detected, the system can generate alerts for the vehicle operator or service technician. These alerts may include information about the specific anomaly, its potential impact, and recommended actions for maintenance or repair.

Anomaly detection is an ongoing process. The gearbox's condition is continuously monitored, and data is analyzed in real time to ensure prompt detection of any emerging issues. Anomaly detection in car gearboxes is essential for predictive maintenance, preventing unexpected failures, and optimizing the lifespan and performance of the vehicle. It is a critical component of modern automotive diagnostics and maintenance systems, especially in commercial and industrial settings where vehicle reliability is of utmost importance.

Navigating the landscape of anomaly detection, this section focuses on the application of logistic regression as a powerful tool. By delving into the intricacies of this statistical model, we uncover its efficacy in identifying anomalies within datasets, shedding light on its nuances with a practical implementation.

Anomaly detection using logistic regression

In this instance, we will employ data captured by accelerometers that recorded vibrations from a gearbox under two distinct operating conditions: one representing a healthy state and the other indicating a broken state. To ensure comprehensive coverage, the sensors were positioned in opposing directions to capture any operational variances. These datasets will serve as the foundation for training various algorithms aimed at classifying the gearbox's operational conditions.

Let's go in deep to describe the algorithm:

1. We start by importing the dataset into the MATLAB environment. The dataset provides measurements of vibrations recorded by two sensors, along with the associated classification of engine operation: `0` denotes a broken state, and `1` indicates a healthy state:

```
data = readtable('GearboxAccData.xlsx');
```

2. A table with `20000` rows and `3` columns was imported.
3. To perform a first visual examination of the data, let's utilize graphs for assistance. For instance, we can create boxplots illustrating the data distribution as detected by the two sensors:

```
subplot(1,2,1)
boxchart(data.state,data.Vib1)
xlabel('state')
ylabel('Vib1')
subplot(1,2,2)
boxchart(data.state,data.Vib2)
xlabel('state')
ylabel('Vib2')
```

The chart seen in *Figure 11.1* will be drawn:

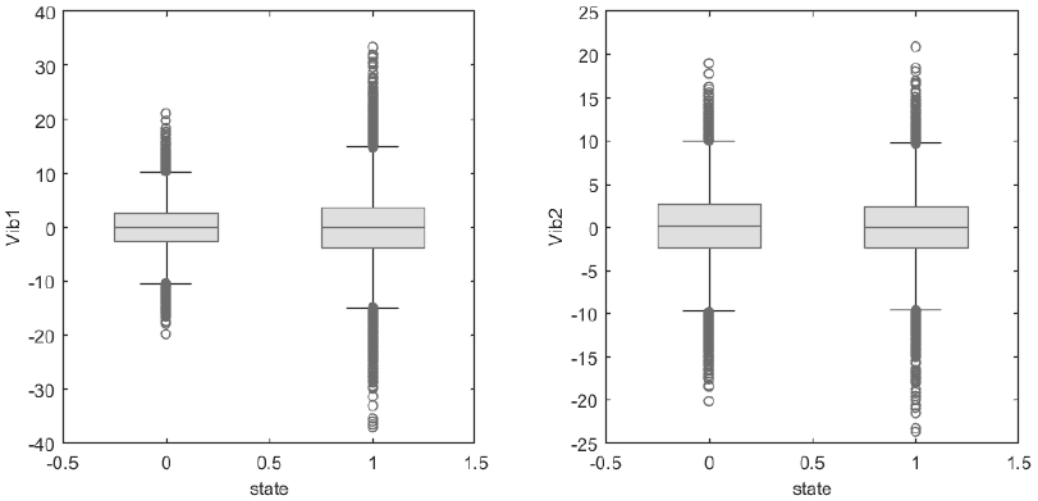


Figure 11.1 – Boxplot of the data

A boxplot serves as a visual representation employed to characterize the distribution of a sample through basic dispersion and positional measures. Upon scrutinizing *Figure 11.1*, it becomes evident that the data distributions from the two sensors vary. Sensor `Vib1` appears to exhibit a greater divergence in vibration values between malfunctioning and normal conditions. This implies that faults are likely manifesting at the location of this sensor.

4. After visually analyzing the data at our disposal, we need to split the data so that we can use it to train an anomaly detection model. Data splitting is a fundamental step in the process of building and evaluating ML models. It involves dividing your dataset into multiple subsets, typically including a training set, a validation set, and a test set. Each of these subsets serves a specific purpose in the ML workflow. We will divide the data into two subsets: a training set and a test set. The training set is the largest portion of your data and is used to train the ML model. It's the dataset the model learns from and uses to update its parameters. The test set is used to evaluate the final, trained model's performance. It provides an unbiased estimate of how well the model will perform on unseen data. The test set should not be used during model development or training to avoid data leakage. Let's see how to make a data split:

```
n = length(data.state);
SplitData = cvpartition(n,'Holdout',0.3);
TrainIndex = training(SplitData);
TrainData = data(TrainIndex,:);
TestIndex = test(SplitData);
TestData = data(TestIndex,:);
```

We first extracted the number of observations present in our dataset using the `length()` function: this function provides the size of the most extensive dimension in the *X* array. When dealing with vectors, this size corresponds to the total number of elements. After that, we used the `cvpartition()` function to establish a random partition for a dataset. We can utilize this partition to create training and test subsets, which are crucial for validating a statistical model. We used the two object functions, `training` and `test`, to extract the training data index and the test data index from the initial dataset. After that, we applied these indices to extract the data.

5. Now, we can build the logistic regression model. Logistic regression is a statistical method used for binary classification, but it can be extended for multiclass classification too. It's a type of regression analysis that models the probability of a binary outcome, often denoted as `0` or `1`. Logistic regression is widely used in various fields, including medicine, social sciences, economics, and ML. Logistic regression is used to predict the probability of an observation belonging to one of two classes or categories. It's not suitable for problems with more than two classes; for such cases, multinomial logistic regression or other techniques are used. The logistic regression model uses a `sigmoid` *(logistic)* function to map input features to a probability between `0` and `1`. Logistic regression is a powerful and interpretable method for binary classification tasks. It serves as a foundational technique in ML and statistics and is especially useful when you need to understand the relationship between input features and the probability of a binary outcome. Let's go to train the model:

```
LogRegModel = fitglm(TrainData{:,1:2},TrainData{:,3},
'Distribution','binomial','link', 'logit')
```

We used the `fitglm()` function for creating a logistic regression model: this function provides a generalized linear model fitted to the variables in the table or dataset array. By default, it designates the last variable as the response variable. We set a binomial distribution of the response variable and `logit` as a link function. The link function $f(\mu) = log(\mu/(1-\mu))$ is associated with logistic regression. It transforms the linear combination of predictors into a probability range (`0` to `1`). In this formula, \(\mu\) represents the probability of the dependent variable being `1`. The logarithmic transformation maps the odds of an event occurring in a linear space, facilitating modeling in the context of binary outcomes. As the logistic function's range is (`0`, `1`), the log-odds ratio spans the entire real number line. This link function is crucial in logistic regression, aiding in the estimation of probabilities and prediction of binary outcomes.

The distribution refers to the probability distribution that describes the variability of the response variable (dependent variable) in the model. In other words, it specifies the form of the likelihood function that the model assumes for the response variable. The choice of distribution is based on the nature of the response variable and the assumptions of the model. The link function defines the relationship between the linear predictor and the expected value of the response variable. We model the linear predictor as a linear combination of the predictors' coefficients. The link function transforms this linear predictor into the expected value of the response variable.

The following model was returned from the training process:

```
LogRegModel =
Generalized linear regression model:
    logit(y) ~ 1 + x1 + x2
    Distribution = Binomial
Estimated Coefficients:
          Estimate        SE         tStat      pValue
        __________    _________    ________    _______
(Intercept)0.00028422   0.016903   0.016814    0.98658
x1          0.00041526  0.0028803    0.14417    0.88537
x2          0.00086682  0.004003     0.21654    0.82857
14000 observations, 13997 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 0.0608, p-value = 0.97
```

6. After training the model, we can use it to identify anomalies, starting from data that the algorithm has never seen before. To do this, we will use the test dataset:

```
PredY = predict(LogRegModel, TestData{:,1:2});
```

We used the `predict()` function; this function provides the forecasted response values generated by the generalized linear regression model. After that, we have to also obtain binary predictions based on a threshold:

```
threshold = 0.5;
PredBin = PredY >= threshold;
```

7. In this way, all the >= values of the threshold are transformed into 1 and all the > values of the threshold are transformed into 0.

8. We now can calculate the accuracy of the model. This term refers to the degree to which a predictive model's predictions align with the actual or observed values. It is a measure of how well the model performs in making correct predictions:

```
accuracy = sum(PredBin == TestData{:,3}) /
length(TestData{:,3});
fprintf('Accuracy: %.2f%%\n', accuracy * 100);
```

9. The following result is printed:

```
Accuracy: 48.82%
```

10. Certainly not a high value of accuracy, which tells us that half of the predictions are wrong. A non-linear model to handle non-linearity in the data is needed. Logistic regression being a linear model fails to capture the non-linear relationship between dependent and independent variables. Surely, we can increase the accuracy by adopting another algorithm, as we will see in the next section.

Let's move on to improving accuracy using the Random Forest algorithm next.

Improving accuracy using the Random Forest algorithm

Random Forest is a powerful ensemble learning algorithm used for both classification and regression tasks. It is an extension of the **Bootstrap Aggregating** (**Bagging**) algorithm that builds multiple decision trees during training and combines their predictions to improve accuracy and reduce overfitting. Random Forest begins by creating multiple random samples (with replacements) from the original dataset. Each of these samples is called a **bootstrap sample**.

For each bootstrap sample, a decision tree is constructed. These trees are often referred to as **base learners** or **weak learners**. Decision trees are grown with the following characteristics:

- **Random feature selection**: At each node of the tree, a random subset of features is considered for splitting, reducing the correlation between trees
- **Limited depth**: Trees are often grown to a certain depth to prevent overfitting
- **No pruning**: The Random Forest algorithm typically does not prune trees

For classification tasks, each tree in the Random Forest algorithmvotes for the class it predicts. The class that receives the most votes is the final prediction. For regression tasks, each tree in the Random Forest predicts a numeric value. The final prediction is the average of these individual tree predictions. The majority vote (for classification) or averaging (for regression) results in the ensemble decision.

Random Forest includes a mechanism to estimate the model's accuracy without the need for a separate test set. This is known as the **out-of-bag** (**OOB**) error. For each data point, it estimates the error rate using only the trees that did not use this point during training. Random Forest can provide an estimate of the importance of each feature in the dataset. This is done by measuring the decrease in prediction accuracy when the values of a particular feature are randomly shuffled.

The following list provides key characteristics and advantages of using Random Forest:

- Random Forest is known for its strong generalization performance and resistance to overfitting
- It can handle both categorical and numerical data
- The random feature selection and ensemble nature make it robust to noisy data and outliers
- Random Forest can be used for feature selection, as it provides a feature importance ranking
- It works well "out of the box" without much tuning of hyperparameters
- It can be used for both classification and regression tasks

Overall, Random Forest is a popular and effective ML algorithm used in a wide range of applications, from image classification and text analysis to financial modeling and healthcare.

To implement an automated system to detect anomalies in gearbox data, we will use the same data already used in the *Anomaly detection using logistic regression* section. Refer to the following steps:

1. Random Forest involves bootstrapping, where we create multiple random samples (with replacement) from the dataset. We can do this to create different subsets of your data. We start by defining some parameters as follows:

```
numTrees = 100;
numSamples = size(TrainData, 1);
sampleSize = round(0.6 * numSamples);  % Adjust as needed
```

2. We defined the number of trees in the forest (`numTrees`), the number of observations that we will use in the training (`numSamples`), and the sample size (`sampleSize`).

3. We will pass `forest` to initialize the forest, setting the data as a `cell` structure:

```
forest = cell(numTrees, 1);
```

Now, we will use a `for` loop to repeatedly execute a block of code a specific number of times or iterate over a sequence. This will be necessary to create a random sample with a replacement first and to then train a decision tree on the sample:

```
for i = 1:numTrees
    sampleIndices = randi(numSamples, sampleSize, 1);
    X_sample = TrainData{:,1:2}(sampleIndices, :);
    y_sample = TrainData{:,3}(sampleIndices);
    tree = fitctree(X_sample, y_sample);
    forest{i} = tree;
end
```

The `fitctree()` function trains a binary decision tree for multiclass classification. It returns a trained binary classification decision tree using the input variables found in the table and the output responses. The binary tree forms a split at branching nodes using values from a column in the table.

The process involves randomly selecting samples to simulate bootstrap sampling, yet the Random Forest classifier typically chooses `\( \sqrt{p} \)` features for each decision tree. In the context of two features, every decision tree would ideally operate with a single feature when introduced into the `for` loop. Consequently, substantial performance enhancement isn't anticipated in this scenario.

4. Now, we can use the model to test the model performance:

```
numTestSamples = size(TestData, 1);
RPredY = zeros(numTestSamples, numTrees);
```

5. We calculated the number of observations in the test data and we initialized the prediction using a matrix of zeros.

6. We can now make the predictions:

```
for i = 1:numTrees
    tree = forest{i};
    RPredY(:, i) = predict(tree, TestData{:,1:2});
end
```

Once again, we used the `predict()` function to make predictions on the dataset never seen before by the algorithm.

7. Finally, we will calculate the performance of the model using the accuracy:

```
FinalRPredY = mode(RPredY, 2);
accuracy = sum(FinalRPredY == TestData{:,3}) /
length(TestData{:,3});
```

8. We used the `mode()` function, which evaluates the most frequent values in an array. This function provides the mode of elements along a specified dimension. In our case, this function returns results in a column vector that holds the most common value for each row.

9. The following accuracy is returned:

```
accuracy
accuracy =
    0.5765
```

We obtained the improvement requested.

In the next section, we will see how to implement a fault diagnosis system using the acoustic emission data of an **unmanned aerial vehicle** (**UAV**).

Building a fault diagnosis system using MATLAB

Technological advancements have given rise to highly capable aircraft that can autonomously manage flight operations. These aircraft belong to the category known as UAVs, meaning they can fly without human pilots. UAVs offer numerous advantages, including significantly reduced operating costs compared to traditional piloted aircraft, the ability to operate in environments unsuitable for human presence, and the capacity for timely aerial surveillance, such as during natural disasters.

Initially employed primarily for military purposes, UAVs were used in dull missions involving monotonous and lengthy surveillance and reconnaissance, as well as in dirty missions that posed risks to human pilots' safety. They also played a crucial role in dangerous missions where human lives were at risk. However, today, they are considered the future of modern aeronautics. The vast potential of UAV technology, its successes in military operations, and advancements in micro- and nanotechnologies have spurred both industries and universities to develop increasingly modern and reliable UAV systems. These systems can be utilized across a wide spectrum of civilian and military missions.

The widespread proliferation of UAVs worldwide has brought to light uncharted issues. While the convenience of using these devices is evident, they also introduce a host of challenges and potential hazards associated with this technology. Among concerns that have arisen is the safety of UAV flights—what would be the consequences if a UAV were to crash to the ground? An automatic system for identifying a defect on the propeller of a UAV, therefore, represents a tool that can guarantee us an acceptable level of safety.

In this example, we will develop an algorithm for fault diagnosis of a UAV propeller from acoustic emission measurements:

1. We start importing the dataset into the MATLAB workspace:

```
UAVData = readtable('DroneFaultDiagnosis.xlsx');
```

2. There are 847 observations, 32 features, 31 predictors, and 1 binary response. We can explore the data using a boxplot:

```
boxplot(UAVData{:,1:31})
xlabel('Features')
ylabel('Leq (dBA)')
```

The following plot will display:

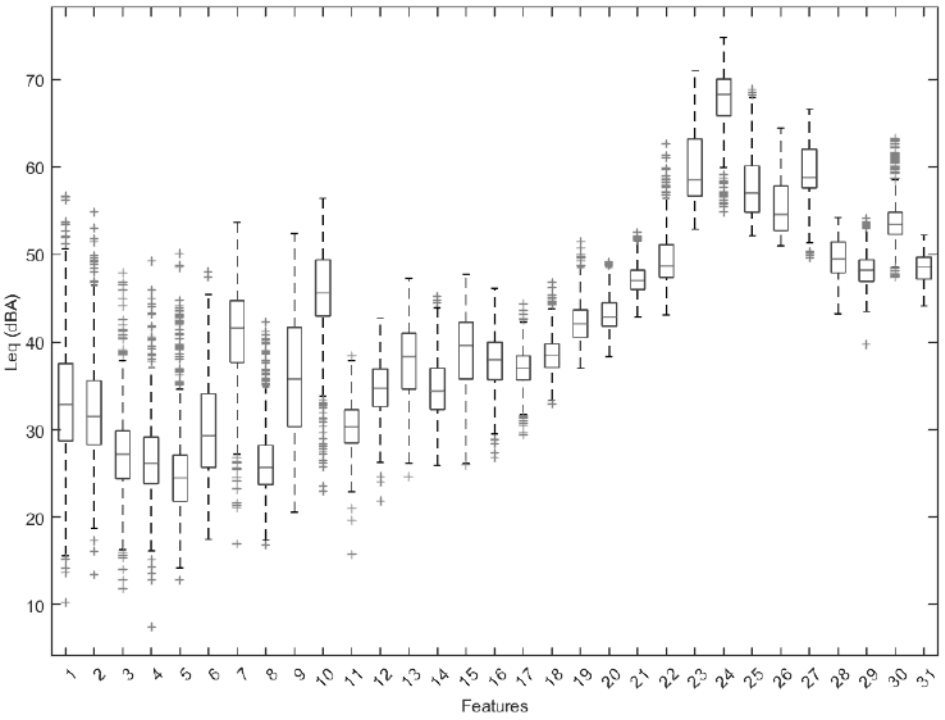


Figure 11.2 – Boxplot of the predictors

We can see that more variability in the data is shown for the low frequencies on the left of the graph (*Figure 11.2*). This tells us that it is the low frequencies that will allow us to identify damage to the UAV's propeller. Low frequencies are valuable for identifying damage to a UAV's propeller due to their unique acoustic signatures. When a propeller is damaged, it often produces distinctive low-frequency vibrations or sounds that are not easily discernible in higher frequency ranges. These low-frequency signals can be indicative of structural issues, such as cracks, imbalances, or deformities, which might not be as prominent in higher frequency ranges. By focusing on low frequencies, you enhance sensitivity to subtle but crucial vibrations associated with propeller damage, providing a more effective means of early detection and preventive maintenance for UAVs.

3. Now, we have to partition data for the model:

```
n= height(UAVData)
RandSplitInd=randperm(n);
DivNum=round(0.7*n);
TrainData=UAVData(RandSplitInd(1:DivNum),:);
TestData=UAVData(RandSplitInd(DivNum+1:end),:);
```

We first extracted the number of observations of the data using the `height()` function, which returns the number of table rows. Then, we used `randperm()`; this function generates a row vector that represents a random permutation of integers from 1 to *n*, ensuring that no elements are repeated. After that, we decided to divide the data into 70% for the training set and the remaining 30% for the test set. Finally, we used the indices obtained to split the dataset into two subsets to train and test the model.

4. To detect faults in the UAV propeller, we will use SVMs for *binary classification*. SVMs represent a category of **supervised ML** (**SML**) models applied in tasks such as classification and regression analysis. They are powerful and versatile algorithms that excel in various applications, including image classification, text classification, and outlier detection. SVMs are primarily designed for binary classification problems, where the goal is to separate data points into two distinct classes. The classifier finds a hyperplane that best separates the data, maximizing the margin between the classes. The margin is the distance between the hyperplane and the nearest data points from each class. SVMs aim to maximize this margin because a larger margin implies better generalization of unseen data and improved robustness.
5. **Support Vector Classification** (**SVC**) and **Support Vector Regression** (**SVR**) are both types of SVMs. SVC is employed for classification tasks, aiming to separate data into distinct categories, while SVR is used for regression, predicting continuous outcomes. In both cases, the SVMs work by finding the optimal hyperplane that maximizes the margin between different classes or fits the data points for regression. They are effective in high-dimensional spaces and versatile in handling complex relationships in data, making them valuable tools in various ML applications.
6. Support vectors are data points closest to the decision boundary (hyperplane). These points are critical in defining the margin and the position of the hyperplane. SVMs derive their name

from these support vectors. In a two-dimensional space, the hyperplane is a straight line that separates the data into two classes. In higher dimensions, it's a hyperplane. The goal is to find the hyperplane that maximizes the margin while correctly classifying as many data points as possible. SVMs can be applied to non-linear problems by transforming the input data into a higher-dimensional space, where a linear separation is possible. Various kernel functions, such as the polynomial kernel, **radial basis function** (**RBF**) kernel, and sigmoid kernel, are used to achieve this. SVMs are known for their ability to provide robust models with strong generalization capabilities, particularly when dealing with high-dimensional data. SVMs are widely used in ML and have shown effectiveness in a variety of applications. While they excel in many scenarios, selecting the appropriate kernel and tuning hyperparameters can be critical for their success. Let's apply SVM model to identify a fault in UAV blades.

```
SVMModel = fitcsvm(TrainData{:,1:31}, TrainData{:,32},
'KernelFunction', 'linear', 'BoxConstraint', 1);
```

7. The `fitcsvm()` function trains an SVM model for classification, which can be used for one-class or two-class (binary) problems, on a predictor dataset with low or moderate dimensions.

   The following parameters are passed:

   - `TrainData{:,1:31}`: The training data for the model is represented as a table, where each row represents an observation, and each column represents a predictor variable. Multicolumn variables and non-character cell arrays are not supported.
   - `TrainData{:,32}`: The name of the response variable should be provided as a character vector or string scalar.
   - `KernelFunction`, `linear`: The choice of kernel function used to calculate the elements of the `Gram` matrix is specified as a pair using `KernelFunction` followed by the name of the kernel function, provided as a comma-separated pair. The following kernel functions are available: `gaussian, linear, and polynomial`.
   - `BoxConstraint`, `1`: This parameter controls the regularization strength. It determines how much error the SVM model is willing to tolerate to achieve a wider margin between the classes. A larger `BoxConstraint` value allows more tolerance for misclassifications, while a smaller value enforces a stricter fit to the training data.

8. Now, we can use the SVM model to perform a fault diagnosis on the acoustic emission of a UAV. We will use the test dataset, which contains data that has not yet been seen by the algorithm:

```
PredY = predict(SVMModel, TestData{:,1:31});
accuracy = sum(PredY == TestData{:,32}) / length(PredY);
```

9. First, we used the `predict()` function to detect a fault in the UAV propeller, and then we calculated the accuracy of the model.

The following result is returned:

```
accuracy
accuracy =
    0.9055
```

91% accuracy is a great result. This is evidence that an SVM is a good choice to predict a possible fault in the UAV propeller.

In the next part of this chapter, we will introduce some methods to prevent overfitting and improve the performance of algorithms.

Understanding advanced regularization techniques

Advanced regularization techniques are methods used in ML and statistical modeling to prevent overfitting and improve the generalization performance of models. Overfitting occurs when a model fits the training data too closely, capturing noise and irrelevant patterns, which leads to poor performance on unseen data. Regularization techniques introduce constraints or penalties to the model's parameters during training to encourage simpler, more generalized models.

Understanding dropout

Dropout is a regularization technique used in NNs, particularly **deep NNs** (**DNNs**), to prevent overfitting. Overfitting occurs when an NN learns to fit the training data too closely, capturing noise and memorizing specific examples rather than generalizing from the data. Dropout is a simple yet effective method for improving a model's generalization performance.

During the training phase, at each forward and backward pass, dropout randomly "drops out" (deactivates) a portion of the neurons (units) in the NN. This means that during the forward pass, the output of some neurons is set to zero with a certain probability, effectively excluding them from the computation. The dropout probability is a hyperparameter and determines the probability that a neuron will be dropped out. A common value is around 0.5, which means that each neuron has a 50% chance of being dropped out during training.

The dropout process introduces variability during training because, for each forward and backward pass, a different set of neurons is active, and the network effectively trains on different subnetworks. This variability forces the network to learn more robust features and prevents it from relying too heavily on any neuron or feature. During the inference or testing phase (when you use the trained model for predictions), dropout is typically turned off. However, the output of each neuron is scaled by a factor of *1/(1 - p)*, where *p* is the dropout probability used during training. This scaling helps maintain the expected values of neuron activations, ensuring that the model's behavior is consistent with what it learned during training.

The key benefits of dropout are as follows:

- **Regularization**: Dropout acts as a regularization technique by preventing the NN from overfitting the training data. Training on various subnetworks encourages the network to generalize better to unseen data.
- **Reducing co-adaptation**: Dropout discourages neurons from relying too heavily on their neighboring neurons. This helps prevent the co-adaptation of neurons, where certain neurons become specialized to predict the output of other neurons. As a result, dropout can lead to more independent and informative features.
- **Ensembling effect**: The dropout process is akin to training multiple models with different architectures, which is similar to an ensemble of models. This ensembling effect can lead to improved model performance.

In practice, dropout is widely used in DL and is especially effective when dealing with large and complex NNs. It's a simple yet powerful technique for improving model generalization and reducing overfitting, making it an essential tool for training robust DL models.

Venturing into the intricacies of model regularization, the next section delves into the distinct realms of L1 and L2 regularization. Unraveling the nuances of these techniques, we explore how they contribute to enhancing the robustness and generalization capabilities of ML models.

Exploring L1 and L2 regularization

L1 and L2 regularization are two common techniques used in ML and statistical modeling to prevent overfitting by adding penalty terms to the loss function. They encourage the model to have smaller weights and, in the case of L1, can lead to feature selection.

Let's introduce two typical methods:

- **L1 regularization (Lasso)**: L1 regularization, also known as Lasso regularization, introduces a penalty term to the loss function, proportionate to the absolute values of the model's coefficients (weights). Mathematically, it can be represented as follows:

  $$Loss\ with\ L1\ regularization = Original\ Loss + \lambda * \Sigma|w_i|$$

  Here, λ (lambda) is a hyperparameter that controls the strength of regularization.

  L1 regularization has the effect of shrinking some of the model's weights to exactly zero. As a result, it performs feature selection by effectively removing less important features from the model. *L1* regularization is especially useful when you suspect that only a subset of your features is relevant for making predictions. It helps you create simpler and more interpretable models by eliminating irrelevant features.

- **L2 regularization (Ridge)**: L2 regularization, also known as Ridge regularization, adds a penalty term to the loss function that is proportional to the square of the model's coefficients (weights). Mathematically, it can be represented as follows:

  $$Loss\ with\ L2\ regularization = Original\ Loss + \lambda * \Sigma(w_i\hat{}2)$$

  Again, λ is the hyperparameter controlling the strength of regularization.

  L2 regularization encourages all the model's weights to be small but rarely forces them to be exactly zero. It prevents any one feature from having an overwhelmingly large weight and has the effect of making the model more stable and numerically well conditioned. *L2* regularization is often used when you want to prevent multicollinearity (high correlation between features) and reduce the impact of outliers. It's suitable when you believe that all features are relevant but should not be overly influential.

In the following list, we summarize the key differences between *L1* and *L2* regularization:

- *L1* encourages sparsity by setting some weights to exactly zero, leading to feature selection, while *L2* mainly shrinks weights but rarely eliminates them entirely
- *L1* is suitable for situations where you suspect that only a subset of features is relevant, whereas *L2* is more appropriate when you want to prevent multicollinearity and ensure all features contribute to the model
- A combination of both *L1* and *L2* regularization is known as elastic net, which balances **feature selection** (**L1**) and **feature grouping** (**L2**)

The choice between *L1* and *L2* regularization depends on the specific problem, the nature of the data, and your understanding of the underlying relationships between features. In some cases, a combination of both *L1* and *L2* regularization may provide the best results by offering a balance between sparsity and stability.

Introducing early stopping

Early stopping is a regularization technique commonly used in ML, particularly in training DNNs, to prevent overfitting and optimize model performance. It involves monitoring the model's performance on a validation dataset during the training process and stopping the training when a certain criterion is met.

During the training process, your dataset is typically divided into two parts: the training dataset and the validation dataset. The training dataset is used to update the model's parameters, while the validation dataset is used to monitor the model's performance. To implement early stopping, you need to define a performance criterion. Common choices for the criterion include validation loss or validation accuracy, depending on the specific ML task (for example, regression or classification). You aim to minimize the validation loss or maximize the validation accuracy.

As the model trains, you periodically evaluate its performance on the validation dataset by calculating the validation loss or accuracy.

You typically monitor the validation performance at regular intervals or after a fixed number of training epochs. You set a threshold, or a rule based on the validation performance, to decide when to stop the training. This could involve tracking whether the validation loss is decreasing or if the validation accuracy has reached a plateau. The most common condition is to stop training when validation performance starts to degrade or stagnate. In other words, you look for the point where the validation loss begins to increase or the validation accuracy starts to decrease. When the early stopping condition is met, training is halted. The model's parameters at the point when training stops are typically considered the final model.

What are the benefits of using early stopping? There are a few:

- **Prevents overfitting**: Early stopping prevents the model from overfitting the training data because it terminates the training process when the model starts to fit the noise in the data
- **Saves training time**: It can significantly reduce training time because the model doesn't continue to train unnecessarily once optimal or near-optimal performance is achieved
- **Enables better generalization**: The model's performance is more likely to generalize well to new, unseen data since it's not overfitting the training data

However, it's essential to be cautious when using early stopping: choosing the right hyperparameters, such as the early stopping threshold or the frequency of validation checks, is critical. The wrong choices can lead to premature stopping or excessive training. It's also important to use a separate validation dataset that the model has not seen during training. Data leakage can occur if the same data is used both for training and validation.

Early stopping is a valuable tool for training ML models, but it should be used with careful consideration of the problem and the specifics of the data. It helps to achieve better model generalization and reduces the risk of overfitting.

Summary

In this chapter, we saw how to implement an automatic fault diagnosis system in MATLAB. We started by introducing the essential concepts of anomaly detection and fault diagnosis. Then, we saw how to implement a system for identifying anomalous operations in MATLAB. We used vibrational data from a gearbox to train a model based on logistic regression. Subsequently, we used the same data, but this time using a model based on Random Forest to improve the performance of the predictive model.

In the next section, we implemented a model for identifying a fault in UAV propellers based on acoustic emission. We used a classification model based on an SVM.

In the final section, we introduced the most popular methods for regularizing algorithms to improve model performance.

In conclusion, this book serves as a comprehensive guide and invaluable resource for both beginners and seasoned practitioners navigating the dynamic landscape of machine learning. The book not only equips readers with a solid foundation in key concepts and methodologies but also empowers them with practical skills to implement and deploy machine learning models using MATLAB. As we embark on the ever-evolving journey of technological advancement, this book stands as a beacon, illuminating the path to understanding and mastering the intricate realm of machine learning. Whether you are a student, researcher, or industry professional, the knowledge gained from these pages will undoubtedly propel you towards harnessing the full potential of machine learning in your endeavors. May this book be a catalyst for innovation and discovery, inspiring readers to contribute to the exciting and transformative field of machine learning.

Index

A

accuracy
  improving, with Random Forest algorithm 326- 328
activation function 141, 143
  exponential linear unit (ELU) function 142
  hyperbolic tangent (Tanh) function 142
  rectified linear unit (ReLU) function 142
  sigmoid function 142
  softmax function 142
  step function 142
Adam optimization
  exploring 175
Adaptive Boosting (AdaBoost) 232, 306
adaptive moment estimation (Adam) 174
Adaptive Synthetic Sampling (ADASYN) 289
adjusted R-squared 81
advanced data preprocessing techniques 52
  correlation analysis 54-58
  data normalization for feature scaling 53
advanced optimization techniques 171
  Adam optimization, exploring 175
  exploring 171
  second-order optimization methods 175, 176
  stochastic gradient descent (SGD) 172-174
advanced regularization techniques 332
  dropout 332, 333
  early stopping 334, 335
  L1 and L2 regularization, exploring 333, 334
anomaly detection system 316
  overview 316, 317
  with logistic regression 322-326
anomaly functioning
  identifying, with ML 321, 322
application programming interface (API) 309
artificial intelligence (AI) 138, 278, 319
artificial neural network (ANN) 137, 138, 179, 249
activation function 141, 143
  architecture 143, 144, 184
  basic concepts 138, 139
  perceptron 140, 141
  used, for building clustering application 166-171
  used, for discovering pattern recognition 157-166
  using, for data fitting 148, 155-157

ANN model
testing, in MATLAB 144
training 144, 145
training, in MATLAB 144
ASCII-delimited files
reading 36-38
attention mechanisms 204
audio data handling 43, 44
autoencoders (AEs) 123, 183, 317
automatic code generation 214
automatic processing 216
autoregressive (AR) models 317
Autoregressive Integrated Moving Average (ARIMA) 266, 321

B

bagging 229, 230
Bag of words (BoW) 218
base learners 326
BERT model 183
best matching unit (BMU) 169
bias 228
binary classification 75, 320
black-box models 203
boosting 231, 232
boosting, techniques
Adaptive Boosting (AdaBoost) 232
eXtreme Gradient Boosting (XGBoost) 233
gradient boosting 233
bootstrap aggregating 305
Bootstrap Aggregating (Bagging) algorithm 326
bootstrap method 95
bootstrapping 229
bootstrap sample 326
borderline-SMOTE 289
Brown Corpus 217

C

capsule networks 183
categorical predictor
multiple linear regression with 82-86
centroid 100
chatbots 211
classification methods
with MATLAB 64
clustering 98
centroids and centers, searching 100, 101
concepts and methods 98
grouping 101, 102
similarity, measuring 98-100
clustering algorithms
families 101
clustering application
building, with ANNs 166-171
collaborative filtering (CF) 294
types 295
computer vision (CV) 43, 179, 236, 241
approaches 240, 241
explaining 240
MATLAB tools, exploring 241-243
Computer Vision Toolbox 23, 24
confusion matrix 162
Connectionist Temporal Classification (CTC) 203
content-based filtering 294-296
content creation 213
convolutional layer, CNN 185-187
convolutional neural networks (CNNs) 21, 124, 179, 182-85, 240, 311
aggregation or pooling neurons 189
building, in MATLAB 189-195
convolutional layer 185-187
FC layer 188
fully connected (FC) architecture 185

pooling layer 187
processing neurons 189
ReLUs 188
coreference resolution 212
corpora 216, 217
Brown Corpus 217
Linguistic Data Consortium (LDC) Corpora 218
New York Times Corpus 217
Project Gutenberg Corpus 218
correlation analysis 54
correlation coefficients, calculating 56
correlation matrix, visualizing 56
performing 55, 56
p-values, calculating 57
scatter plots, visualizing 58
cosine distance 100
cost-sensitive learning
discovering 291
creative writing 214
cross-entropy performance function 159
cross-validation 71
C-support vector machine (C-SVM) 73
customer segmentation 107

D

data
exporting, from MATLAB 39-41
grouping, with similarity measures 115, 116
importing, into MATLAB 30-32
data cleaning 45-47
data type conversion and validation 45
data validation and error checking 45
duplicates, handling 45
inconsistent or incorrect data, handling 45
missing values, handling 45
outliers, removing 45
data fitting 148
example 149
with ANNs 148, 155-157
data normalization 53
Min-Max scaling 53
Z-score standardization 53
dataset
converting, into dataset into a time series format in MATLAB 270-272
data sharing 39
data-to-text generation 213
data wrangling 44
data aggregation 45
data cleaning 44
data integration 45
data reduction 45
data reshaping 45
data transformation 44
decision trees 65, 66
elements 65
exploring, in MATLAB 66-72
for decision-making 64
deconvolutional networks (DeconvNets) 259
deep AEs (DAEs) 317
deep learning (DL) 179, 180, 317
automated feature extraction 180
DNN, training 181
techniques 211
Deep Learning Toolbox 21, 22
deep NNs (DNNs) 179, 180, 332
training 181, 182
dendrites 140
dendrogram 101
denial-of-service (DoS) attacks 303
density-based clustering 100
dependency parsing 212
design of experiments (DOE) 21
diagnostics 317

dialogue generation 213
dimensionality reduction, techniques
approaches 120
challenging, reasons 120
discovering 119, 120
feature extraction algorithms, exploring 122-124
feature selection methods 120-122
dissimilarity 99
DL architectures 201
LSTM networks, analyzing 202, 203
RNNs 201
transformer models 204
DL models
autoencoders (AEs) 183
capsule networks 183
CNNs 182
exploding 182
generative adversarial networks (GANs) 183
graph NNs (GNNs) 183
long short-term memory (LSTM) networks 182
Neural Turing machines (NTMs) 183
recurrent NNs (RNNs) 182
transformer networks 183
dots per inch (dpi) 237
dropout 181, 332, 333
benefits 333

E

early stopping 334, 335
benefits 335
ecology and environmental science 107
edited nearest neighbors (ENN) 290
electric vehicle (EV) 278
ensemble learning 228
bagging 229, 230
boosting 231, 232
random forest 230, 231
ensemble methods 317
using 305
epsilon-SVM 73
Euclidean distance 99
expectation-maximization (EM) algorithm 108
explanatory role 78
exploratory data analysis (EDA) 39, 48
correlation analysis 48
data distribution 48
data summary 48
hypothesis testing 48
multivariate analysis 48
outlier detection 48
practical cases 48-51
exploratory statistics 47
exploratory visualization 51, 52
bar() function 52
boxplot() function 52
contour() function 52
heatmap() function 52
histogram() function 52
imagesc() function 52
line() function 52
pairplot() function 52
plot() function 51
scatter() function 52
exponential linear unit (ELU) function 142
exponential moving average (EMA) 274
exponential smoothing 276
variations 277
eXtreme Gradient Boosting (XGBoost) 233

F

fault diagnosis system 315-319
approaching, with ML 319-321
building, with MATLAB 328-332
issues, addressing approaches 320
techniques 318, 319
FC layer 188
feature engineering (FE) 180
feature extraction 239
MATLAB, using 124
feature extraction algorithms
exploring 122-124
feature grouping (L2) 334
feature importance scores
approaching 258
methods and techniques, for calculating 258
feature map 186
feature selection
MATLAB, using 124
stepwise regression for 124-129
feature selection (L1) 334
feature selection methods 120, 121
categories 121, 122
purposes 121
feature spaces 74
feedback loop 201
feedforward NNs (FNNs) 201
first quartile (Q1) 50
forecasting methodologies 263
forget gates 203
F-statistic versus constant model 81
fully connected (FC) 249
function fitting 148

G

Gaussian Mixture Models (GMMs) 101, 317
gaussian or RBF 77
Generalized Autoregressive Conditional Heteroskedasticity (GARCH) 266
generative adversarial networks (GANs) 21, 183
generative pre-trained transformer (GPT) 214
gradient-based attribution methods
discovering 259, 260
gradient boosting 233
gradient boosting techniques 228
ensemble learning 228
Gradient Class Activation Mapping (GradientCAM) 259
graphical user interface (GUI) tool 252
graph NNs (GNNs) 183

H

handwriting recognition (HWR) 244-250
Hessian matrix 175
hidden Markov models (HMMs) 203
hierarchical clustering 100-107
types 102
histogram of oriented gradients (HOG) 124, 243
human visual system (HVS) 235
hybrid recommender systems 294-297
hyperbolic tangent (Tanh) function 142
hyperparameters 181
hyperplane 72

I

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 22
image processing 107, 236-239
methods 238
images
displaying 42
loading 42
manipulating 42
saving 42
working with 42, 43
imbalanced datasets, in MATLAB
dealing with 288
oversampling 289
undersampling, exploring 290, 291
Import Tool 32
exploring 32, 33
independent component analysis (ICA) 123
information extraction 212
interquartile range (IQR) 45

K

kernel density estimation (KDE) 317
kernel function 74
k-fold cross-validation 91-94
k-means algorithm 108, 109
k-means clustering 100
using, in MATLAB 109-115
k-medoids
applying, in MATLAB 116-119
versus k-means 115
k-nearest neighbors (k-NN) algorithm 298
knowledge distillation 312, 313
Kohonen maps 169

L

L1 and L2 regularization
exploring 333, 334
methods 333
L1 norm 181
L1 regularization 181
L1 regularization (Lasso) 333
L2 norm 181
L2 regularization 181
L2 regularization (Ridge) 334
language generation 211
language modeling 213
language models 211
latent Dirichlet allocation (LDA) 24
latent semantic analysis (LSA) 24
layer-wise relevance propagation (LRP) 259
learning processes
types, discovering 8, 9
learning processes, types
reinforcement learning 11, 12
semi-supervised learning 12
supervised learning 9, 10
transfer learning 13
unsupervised learning 10
leave-one-out cross-validation (LOOCV)
exploring 94
Levenberg-Marquardt (LM) backpropagation 149
working 149-155
linear 74, 77
linear discriminant analysis (LDA) 122
linear regression 78, 79
linear regression model
in MATLAB 79-82
Linguistic Data Consortium (LDC) Corpora 218

load() function 34, 35
using, to import files 34
local binary patterns (LBP) 124
logistic regression
used, for anomaly detection system 322-326
long short-term memory (LSTM) 284, 321
long short-term memory (LSTM) networks 21, 182
low-rank approximation 313
LSTM model
using, to label sentences 222-227

M

Machine learning (ML) 3, 4, 179, 209
analysis, of logical reasoning 4-6
real-life applications 25, 26
strategy typologies 6, 7
techniques, using 13
used, for approaching fault diagnosis system 319-321
used, for identifying anomaly functioning 321, 322
Machine learning (ML), techniques
paradigm, selecting 14, 15
machine translation 213
Manhattan distance 100
market research 107
MATLAB
advanced techniques, using for model evaluation and selection 90
classification methods with 64
CNN, building 189-195
decision trees, exploring 66-72
k-means clustering, using in 109-115
k-medoids, applying 116-119
linear regression model 79-82
NIDS-adopting recommender system, building with ensemble methods 305-309
partitioning-based clustering algorithms 108
predictions, with regression analysis 82
pre-trained deep learning models, training and fine-tuning 250
time series data, resampling 273, 274
tools, exploring for computer vision 241-243
used, for building fault diagnosis system 328-332
used, for creating recommender systems for network intrusion detection 303, 304
used, for testing artificial neural networks (ANNs) model 144
used, for training artificial neural networks (ANNs) model 144
using, for feature selection and feature extraction 124
MATLAB Deep Network Designer app 252-257
MATLAB model
building, for object recognition 243, 244
MATLAB toolboxes, for ML
Computer Vision Toolbox 23, 24
Deep Learning Toolbox 21, 22
exploring 19, 20
Reinforcement Learning Toolbox 22, 23
Statistics and Machine Learning Toolbox 20, 21
Text Analytics Toolbox 24, 25
Matrix Laboratory 20
max pooling layer 187
mean absolute error (MAE) 87, 286, 295
mean squared error (MSE) 87, 152, 239, 286
median absolute deviations (MADs) 47
medical reports 214

Minkowski distance 100
Min-Max scaling 53
Minority Synthetic Over-sampling Technique (MSMOTE) 289
ML applications 25
issues, types 25
ML models
building, step-by-step guide 16-19
deploying 309
feature importance scores, approaching 258, 259
gradient-based attribution methods, discovering 259, 260
interpreting and explaining 257
knowledge distillation 312, 313
low-rank approximation 313
model compression 310
model pruning techniques, discovering 310, 311
quantization, for efficient inference on edge devices 312
saliency maps 257, 258
model
implementing, to predict stock market 278-288
model compression 310
techniques 310
model evaluation and selection, in MATLAB
advanced techniques, using 90
model performance
evaluating 86
outlier effects, reducing 87-90
model pruning techniques
discovering 310
methods 310
Modified National Institute of Standards and Technology (MNIST) 244
morphology analysis 214
movie review sentiment analysis 221
moving averages (MAs) 317
multiclass classification 320
multiple linear regression
with categorical predictor 82-86
multivariate algorithms 65

N

named entity recognition (NER) 212
NaN code 46
natural language analysis (NLA) 212
coreference resolution 212
dependency parsing 212
information extraction 212
language modeling 213
named entity recognition (NER) 212
parsing ambiguity 213
semantic understanding 212
sentiment analysis 212
syntax analysis 212
text classification 212
natural language generation (NLG) 212, 213
automatic code generation 214
content creation 213
data-to-text generation 213
dialogue generation 213
machine translation 213
medical reports 214
personalized messaging 214
storytelling and creative writing 214
text summarization 213
natural language processing (NLP) 182, 209-211
chatbots and virtual assistants 211
language generation 211

language models 211
speech recognition 211
tasks, analyzing 214
text understanding 211
Natural Language Processing Toolbox 218
network intrusion detection
MATLAB, used for creating recommender systems 303, 304
methods 303
techniques 304
network intrusion detection system (NIDS) 293
recommender systems 304, 305
neural network, design 147
test set 147
training set 147
validation set 147
neural network, layers
hidden layer 143
input layer 143
output layer 143
neural networks (NNs) 317
Neural Network Toolbox 145-147
command-line operations 145
customize toolbox 145
graphical user interfaces (GUI) 145
toolbox function, modifying 145
Neural Network Training window
algorithms 160
neural network 160
plots 160
progress 160
Neural Turing machines (NTMs) 183
Newton-like methods 175
New York Times Corpus 217
NIDS-adopting recommender system
building, in MATLAB 305-309
nominal part (NP) 215
non-negative matrix factorization (NMF) 120, 123
null hypothesis 82

O

object recognition 190
MATLAB model, building 243, 244
One-Class Support Vector Machine (One-Class SVM) 317
Open Neural Network Exchange (ONNX) 22
optical character recognition (OCR) 244
out-of-bag (OOB) 326
overfitting 239
oversampling 289
methods 289

P

parsing ambiguity 213
partitioning around medoids (PAM) 118
partitioning-based clustering algorithms
with MATLAB 108
part-of-speech tagging 215
pattern recognition 157
discovering, with ANNs 157-166
perceptron
working 140, 141
personalized messaging 214
polynomial 74
functions 77
pooling layer, CNN 187
benefits 187
pragmatic analysis 215
pragmatic competence 216
predictions, with regression analysis
in MATLAB 82

predictive forecasting 262, 263
predictive role 78
pre-trained deep learning models
training and fine-tuning, in MATLAB 250
Principal component analysis (PCA) 20, 48, 120, 122, 129-135
Project Gutenberg Corpus 218
p-value 81, 82

Q

qualitative forecasting methodologies
describing 264
quantitative forecasting methods
conditions 264
examples 263
explaining 263, 264
quantization
for efficient inference on edge devices 312
quantization, in model compression
approaches 312

R

radial basis function (RBF) 74, 331
random feature sets
for partition choice 231
Random Forest algorithm 230, 231
used, for improving accuracy 326-328
random oversampling 289
random sampling
of training data 231
receiver operating characteristic (ROC) 164
recommender system
concepts 294
types 294
recommender systems
collaborative filtering (CF) 294
content-based filtering 295, 296
creating, for network intrusion detection using MATLAB 303, 304
for NIDS 304, 305
hybrid recommender systems 296, 297
rectified linear unit (ReLU) function 142, 188
recurrent neural network (RNN) 182, 201, 214, 222, 284, 285, 317
architecture 202
recursive feature elimination (RFE) 258
regression 320
regression analysis
types, exploring 78
regularization techniques 181
dropout 181
L1 regularization (sparsity) 181
L2 regularization 181
L2 regularization (weight decay) 181
reinforcement learning 11, 12
Reinforcement Learning Toolbox 22, 23
principles 23
ResNet pretrained network 251, 252
features 251
resubstitution error 71
root mean square error (RMSE) 81, 287, 295
R-squared 81

S

saliency maps 257, 258
scaled conjugate gradient (SCG) 158
Scale-Invariant Feature Transform (SIFT) 243

second-order optimization methods 175, 176
self-organizing map (SOM) 168
semantic analysis 215
semantic understanding 212
semi-supervised learning 12
sentence tokenization 219
sentence tokenizers 216
sentiment analysis (SA) 183, 212, 220
lexicon-based methods 220
ML methods 221
rule-based methods 220
statistical methods 221
SHapley Additive exPlanations (SHAP) 258
sigmoid 74
sigmoid function 142
significance level 82
silhouette plot 113
similarity measures 99
used, for grouping data 115, 116
simple exponential smoothing (SES) 277
simple moving average (SMA) 274
simultaneous localization and mapping (SLAM) 23
singular value decomposition (SVD) 120, 313
softmax function 142
speech recognition 211
Speeded-Up Robust Features (SURF) 243
Squared Euclidean 109
Statistics and Machine Learning Toolbox 20, 21
step function 142
stepwise regression
for feature selection 124-129
stochastic gradient descent (SGD) 158, 172-174
stochastic gradient descent with momentum (SGDM) optimizer 195
stock market
model, implementing to predict 278-288
storytelling 214
strategy typologies 6, 7
strong learner 228
Structural Similarity Index Measure (SSIM) 239
supervised classification
with SVMs 75-78
supervised learning (SL) 9, 10, 182
framework 320
supervised ML (SML) models 330
Support Vector Classification (SVC) 330
support vector machines (SVMs) 20, 64, 72-74
using, for supervised classification 75-78
Support Vector Regression (SVR) 330
syntax analysis 212, 215
Synthetic Minority Over-sampling Technique (SMOTE) 289

T

Term Frequency-Inverse Document Frequency (TF-IDF) 296
testing set 74
text analysis 107
Text Analytics Toolbox 24, 25, 218
text classification 212
text summarization 213
text understanding 211
third quartile (Q3) 50
time series
slicing and dicing 272, 273

time series data
analysis 265-269
components 269
concepts, exploring 262
distinct patterns 268
exponential smoothing 276-278
forecasting methodologies 263
moving average 274-276
predictive forecasting 262, 263
resampling, in MATLAB 273, 274
statistics, extracting from sequential data 269, 270
types 265
time series format, in MATLAB
dataset, converting into 270-272
token 219
tokenization
challenges, to address 219
train() function 153
training set 74
train-validation-test split 16
transfer learning 13
approaches 13
transformer models 204
applying 204
transformer networks 183
t-test 81

U

undersampling
cost-sensitive learning, discovering 291
exploring 290, 291
univariate rules 65
unmanned aerial vehicle (UAV) 328
unsupervised learning (UL) 10, 183
users, in data
searching 297-303

V

vanishing gradient problem 182
variance 228
verbal part (VP) 215
virtual assistants 211
virtual intelligence (VI) 297
voronoi tessellation 298

W

wavelet transform 123
weak learners 228, 326
Web Services Toolbox 218
Word2vec 218
word embedding 24
Bag of words (BoW) 218
Word2vec 218
words 216-218

Z

Z-score standardization 53

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

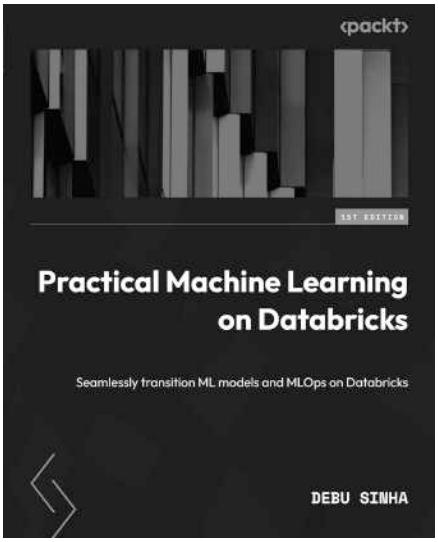
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

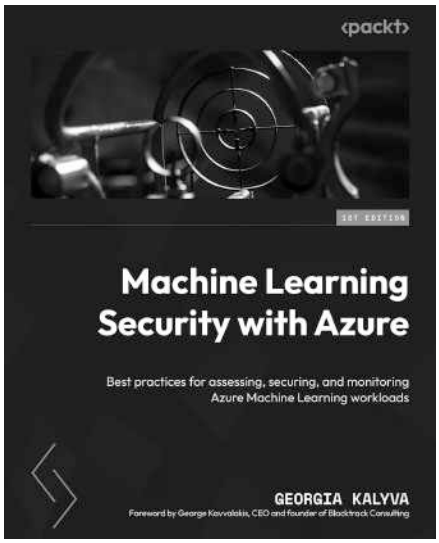


Practical Machine Learning on Databricks

Debu Sinha

ISBN: 978-1-80181-203-0

- Transition smoothly from DIY setups to databricks
- Master AutoML for quick ML experiment setup
- Automate model retraining and deployment
- Leverage databricks feature store for data prep
- Use MLflow for effective experiment tracking
- Gain practical insights for scalable ML solutions
- Find out how to handle model drifts in production environments



Machine Learning Security with Azure

Georgia Kalyva

ISBN: 978-1-80512-048-3

- Explore the Azure Machine Learning project life cycle and services
- Assess the vulnerability of your ML assets using the Zero Trust model
- Explore essential controls to ensure data governance and compliance in Azure
- Understand different methods to secure your data, models, and infrastructure against attacks
- Find out how to detect and remediate past or ongoing attacks
- Explore methods to recover from a security breach
- Monitor and maintain your security posture with the right tools and best practices

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *MATLAB for Machine Learning*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835087695`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly